Thoughts Beyond C++

First published: 2024-12-30
Last updated: 2025-01-13
Author: Adam Sawicki - see About the author

Short address of this document: https://asawicki.info/articles/thoughts_beyond_cpp.html

Introduction

This document gathers thoughts about C++ programming language - what things are good, what things are bad in it, and what a new, better programming language could look like. It is written from the perspective of a programmer working with games, game engines, and real-time graphics. References to other programming languages also appear, including Python, C#, Rust, and GPU programming (shader) languages: HLSL, GLSL.

I believe a new programming language could and should be developed to supersede C++ at least in some applications, like game development and other software that is complex, but requires high performance at the same time. C++ is popular in those applications because it is the best we have right now, but at the same time it has many flaws and it is unfixable due to backward compatibility and the general direction it is heading.

This document is less about a specific language syntax, and more about generic ideas for designing a language that is:

- more convenient to use,
- safer - with less opportunities to introduce bugs,
- more efficient - giving the compiler opportunity to optimize the code better.

The document is divided into 2 main parts, describing “the good parts” and “the bad parts” of C++, but all the sections really discuss pros and cons of various aspects of the language, compare it to other languages, and suggest what a better syntax could look like. I don’t have all the answers. In many cases I propose some design decisions, including proposed language syntax, but more often I just point to the problem of C++ or talk about higher-level ideas.

Disclaimers

In many places the document says “I think…”, “in my opinion”, but in reality it is entirely my personal “wishlist”. This is just a random programmer from Eastern Europe having an opinion.

Note also that all the opinions in this document are my own and do not reflect that of my employer.

If you decide to read it, you do it at your own responsibility of losing time and getting irritated if you disagree with something. Comments are welcome. The document will be updated.

This document contains some strong opinions, like what things I consider “good” or “bad”. I reserve the right to have limited knowledge in many topics, to have my opinions biased due to my personal experiences and career path, to be wrong with some of these things, and to change my mind in the future.

The good parts

Things that C++ does right:

Level of abstraction

Why C++ stays dominant language in so many applications despite all its flaws? I think this is because C++ is at the medium level of abstraction. On one hand, it gives low-level, explicit control over memory allocation and computations. On the other hand, it provides high-level features like classes, templates, strings, containers, etc. that allow convenient development of complex systems.

Languages that are higher level: scripting languages like Python, JavaScript, and languages that require some interpreter/JIT/runtime like Java, C# have many advantages, like easy portability between platforms, more safety (garbage collector, bounds checking etc.). However, they are not suitable for many applications due to their requirement of having the runtime installed on the client machine, due to size of the program compiled to the intermediate form, slow startup time, runtime memory requirements, unpredictable performance (when garbage collector kicks in), and, most importantly, lower overall runtime performance.

Why not just use C?

C, on the other hand, is a lower-level language. Many developers, first learning all the capabilities of “modern C++”, after a period of fascination with elaborate template trickery, go back to plain C or close-to-C-looking C++ code and consider it as a higher level of experience and seniority. I don’t agree with this. I think the problems they eventually noticed in C++ are exactly this - problems with C++, not proof that all high-level languages are bad or that C is great.

I consider C a very low level language, just a small overlay on top of assembly. It provides:

- structs,
- conditions and loops instead of jumps,

- some type safety (although limited - think about `void*` and `malloc`, about unsafety of macros, or arguments passed to `printf` and `scanf`).

After all, pointers, function calls, and returns already exist in the assembly. I wouldn't call high level any language that doesn't provide built-in strings, vectors, lists, and other containers.

Some developers praise C for its explicitly so that no complex and time-consuming work can be hidden under an innocent looking operator when doing `a+b` or in a destructor called when an object goes out of scope with `}`. I also disagree with that. I think that the right way to solve this is not to be *explicit*, but to be *transparent* - like showing generated assembly next to our source code. For an idea how to do it, see section Integrated development environment.

Those who were programming graphics in the early 2000s remember that the first shaders were written in assembly. Then, High Level Shader Language (HLSL) appeared. Nowadays, we can only write shaders in high level languages. We just look at the generated assembly code (ISA), we go back to the HLSL code and we optimize without too much whining that things happen behind our back and we don't have manual control over every assembly instruction.

Having to write everything so explicitly results in any complex system written in C requiring 10x too much boilerplate code and having 10x too many bugs made on the way - crashes, memory leaks, security vulnerabilities. I believe that developing complex and high quality systems requires high level languages.

Why not just use Rust?

Rust is a great programming language in many aspects. Among all the new native languages that can supersede C++, I think it has the biggest chance. Some companies already use it for professional software development. When I was learning Rust, I found many language features very well designed. It solves many problems of C++ that I describe in this document.

However, after some more time spent with it, I eventually concluded that it isn't convenient in developing larger software projects and I am not alone with this opinion. Fighting with the borrow checker to make the program compile successfully is too much hassle. To me, the design of Rust seems too obsessed with this one specific problem of object lifetime and ownership. It is an important problem, but the cure shouldn't be worse than the disease. It is easier to reason about object lifetimes when we write an algorithm that executes begin to end, much more difficult if we write a GUI application with windows and widgets persistent on the screen or a game with various objects persistent on a scene.

To understand my point, imagine we get fixated on another class of bugs - mixing different units of measurement, like adding seconds to milliseconds, kilometers to miles, bits versus bytes etc. We design a language where we ask developers to annotate every numeric constant and variable with a physical unit, and the compiler that checks them diligently so we can only compile the code if we correctly divide the distance by the time to calculate the speed etc. It

would surely help avoiding certain class of bugs, but it wouldn't be convenient to use. This is what I think about borrow checking in Rust.

Rust is also known to compile slower than C++, which is quite an achievement…

Following articles that explain various problems with Rust very well:

- Leaving Rust gamedev after 3 years by LogLog Games
- I spent 18 months rebuilding my algorithmic trading platform in Rust. I'm filled with regret. by
- Austin Starks

Why not just use modern C++?

What if we use the latest and greatest features of C++11/14/17/20/23/26 and ban using certain legacy features? What if we use modules instead of header files, `std::unique_ptr` instead of `new` and `delete` etc.?

This would solve many problems described in this document, but some problems are unfixable in C++ - partially due to the need for backward compatibility to decades of old C++ and all the way back to C, and partially due to the general approach and direction that the development of the language is heading.

There are attempts to improve C++ beyond what the language committee is doing, while still ensuring compatibility with the base language, like Safe C++ by Sean Baxter and Christian Mazakas. However, if you look at the chapter 1.1 of their document, you will see that even the United States Government is calling for migrating away from C++ and adopting some other languages rather than hoping that C++ can be fixed.

Syntax

Should the scopes be declared with `{}` or `begin end`? Variables declared like `int i` or `var i: int`? Should we recommend `camelCase`, `PascalCase`, or `snake_case`? I think the syntax of the language is not the most important topic.

When designing a new language, decisions about the syntax should be well justified, but at the end, there is still room for a choice among equally good options, which is a matter of taste. Then, I think the following criteria should be considered:

- Familiarity of the syntax. If we want to replace C/C++, we should prefer a similar syntax, so definitely `{}` not `begin end`.
- Ease of parsing. The fact that problems like the most vexing parse exist in C/C++ is a failure of this language that we should avoid, so having a keyword like `var`, `type`, and `function` (or `func`, or `fn`, whatever we choose) may actually be a good idea.

A syntax might look like this, for example:

```
function Add(a: int32, b: int32): int32
{
    var c: int32 = a + b;
    return c;
}
```

Why not syntax similar to Python?

Python syntax with scopes defined by indentation has an advantage of being very concise and looking like a pseudocode written by humans for humans. On the other hand:

- It looks unlike any other popular language.
- Python is a scripting language far from the domain of C++ and other native languages discussed here.
- It is also easier to make a mistake with indentation than with the placement of explicit scopes with `{ }`.

Strong typing

In a strongly typed language:

```
int i = 1;
i = "ABC"; // Error, i should be an integer.
```

With dynamic typing:

```
i = 1
i = "ABC" // OK
```

Each local variable, function parameter, and class member having a specific type explicitly declared and known at compile time is a good thing. It is also required for languages like C++ to compile to an efficient, native code. There is no discussion about that.

But higher level, more dynamic scripting languages that traditionally allow dynamic typing also go in this direction. TypeScript adds explicit types over JavaScript. Python introduced type hints in version 3.5. This shows that dynamic typing was a dead end. The more we can check at compile time versus experiencing a runtime error, the better is the programming language, because preventing or finding and fixing such bugs is easier, and because of better runtime performance. Static typing is a great example of this rule.

Automatic type inference

One thing that scripting languages do right is automatic type inference. If the type can be inferred from an expressing that evaluates it, we don't need to specify it explicitly. In C++ we also have it in form of `auto` keyword (the new meaning of this ancient keyword).

Depending on the situation, this can be a bad or a good thing. It is good when it saves a lot of typing, like:

```
auto it = my_map.cbegin()
```

instead of:

```
std::map<std::string, std::vector<MyClass>>::const_iterator it =
my_map.cbegin()
```

It is bad when the type is not obvious from the context, so omitting it makes the piece of code less readable and self-explanatory. Still, it is a good feature to have available.

Multiple types

Another thing that scripting languages do right is allowing some variable to be of one of multiple types, like in TypeScript:

```
let name: string|null = "John";
name = null;
```

This is convenient and useful in many situations. In C++ we have it in form of `std::optional` and `std::variant`, although the syntax is verbose and clumsy. Like many other things, it should be built into the language syntax (like in TypeScript or Python) rather than implemented using template tricks in the standard library.

Preferably, a pattern matching syntax should also be available for "unpacking" such multichoice types, like in Rust:

```
fn ConsumeName(name: Option<String>) {
    match name {
        Some(value) => println!("Hello, {}!", value),
        None => println!("Hello, Guest!"),
    }
}
```

Const-ness

It is good that in C++ we can declare variables as `const` to promise we only assign their value once and never change it later. It is also good that we can declare `const` functions that promise to not mutate the members of its class, so we can call them on `const` objects. Higher-level scripting languages lack this feature.

```
// We are sure c will never change its value.
const int c = a + b;
// s is passed by reference, but we are sure the function won't change
it.
function Print(const std::string& s);
```

It might be a good idea to even make it the default, and require some special keywords to define a variable as mutable, like the `mut` keyword in Rust. With the right support for the language syntax, we can use const variables most of the time. We need mutable variables only occasionally, like for the iterator of a loop.

```
// i is mutable as the loop iterator.
for(size_t i = 0; i < user_count; ++i)
{
    // u can be const thanks to declaring it locally here.
    const UserRecord& u = users[i];
    // Use u...
}
```

However, it is not good that C++ may require writing a separate version of const and non-const methods. I like how C++ FQA calls it *"This is about the const keyword, which makes you write your program twice"*. I think it should be solved somehow. For example, the compiler could infer that the method doesn't mutate the members of its class and automatically allow calling it for const objects. C++23 solves this by introducing "deducing this".

Manual memory management

One of the best features of C++ and other native programming languages is the manual, explicit control over memory allocation and deallocation. This is important for the control and predictability of the memory usage and runtime performance.

Garbage collectors are bad in high-performance low-level systems code, because they allow an unpredictable amount of memory to be still used despite not needed, they activate in unpredictable moments causing performance hitches in the program for unpredictable amount

of time. They require additional logic and impose having some framework/runtime/interpreter executing our program - a thing that it alien to the native code.

Using reference counting for all objects by default is also bad, because they come with a runtime cost, while it is not needed most of the time. Reference-counting smart pointers like `std::shared_ptr` and `std::weak_ptr` in C++ are fine to be available for these rare cases where we may find them useful, but in the majority of cases, objects have clear ownership and lifetime - object A is an exclusive owner of object B and destroys it at most when it is destroyed.

Explicit ownership and lifetime

However, manual memory allocation comes in a multitude of forms in C and C++, dating back to the ancient `malloc/free`, through the `new/delete` operator, the older `std::auto_ptr` (which was a terrible solution), to the latest `std::unique_ptr` class and `std::make_unique` function, which let us never call `new` or `delete` explicitly. Out of these options, I like `std::unique_ptr` and the concept of RAII.

- It makes the code safer, as we don't need to explicitly free the memory / delete the object, so there is no danger of forgetting about it.
- It makes the code shorter, as the object is automatically freed at the end of the scope (for local variables) or when the parent object is destroyed (for member variables), so we don't need to type in the destruction matching the creation of every dynamically allocated object.
- It explicitly expresses the ownership and lifetime of the object within the language.

I think that instead of having some specific smart pointer class defined in the standard library, such "owning" pointer should be the default in the language. This is a thing that Rust does right. The syntax could look similarly to references to dynamically allocated objects using `*` or `&` operator, but instead of garbage collector freeing the object in some indefinite future, the end of scope should be obvious from the context.

```
{
    MyClass* obj1 = new MyClass();
}
// obj1 was destroyed at the end of the scope.
```

Move semantics and r-value reference

I think the idea of move semantics and r-value reference is one of the best things added to C++. It allows moving ownership or big objects without a need to copy them or even supporting the copy operation (like copy constructor). It allows returning objects from functions by value instead of using output parameters passed by a pointer or a reference, like we used to do before.

However, many aspects of this new mechanics are not so good:

1.
First, the complexity of the whole thing. If you read more about the details of it, you will learn how C++11 adds on top of the old *lvalue* (that can stand on the left hand side of the assignment) and *rvalue* (that can only stand on the right hand side of the assignment) new types of values: *xvalue*, *prvalue*, *glvalue*. It is hard to avoid the feeling that it wouldn't need to be so complex if only the compiler could optimize things better automatically while having more high-level information about what we intend to do.

Eric Lengyel also has some thoughts about it, with a proposal for `transient` qualifier to be potentially a better and simpler solution.

2.
Return Value Optimization (RVO) and Named Return Value Optimization (NRVO) - optimizations performed by the compiler, but mentioned in the language standard, can help reducing the number of object copies, but this is still limited and could be taken further.

We should never need to think about whether passing objects by pointer or reference as an “output parameter” instead of function return type, or passing object by `const&` instead of by value is needed to improve the performance. Compiler should figure out such optimizations automatically and avoid making copies, as long as it doesn’t change the logic, because the copy operation of the class is known to have no side effects. As side effects I understand opening disk files or showing things on the screen, not dynamic allocation of some buffer in memory. For this, the compiler would need to understand that a string is a string and an array is an array, not some user-defined class with a mess of random pointers inside.

This:

```
string ConcatStr(string a, string b) {
    return a + b;
}
```

Should be optimized by the compiler to be as efficient as this manually optimized and more verbose form:

```
void ConcatStr(string& out, const string& a, const string& b) {
    out = a + b;
}
```

In modern C++, returning string by value will probably be optimized, but passing strings as parameters by value will not.

3.

Finally, the fact that “moving” out of a variable leaves it in some empty, but valid and usable state is bad, because:

- It creates an opportunity for bugs when we try to use it afterwards, which is not something we would typically intend to do.
- It requires resetting the pointers and counters of the source object to 0, which is unnecessary work.

I think Rust does it better by disallowing using the variable after the object was moved out of it. For example, in the new language the syntax might look like this:

```
MyClass* obj1 = new MyClass();
MyClass* obj2 = move obj1;
// obj2 became the owner of the object, obj1 is no longer usable.
```

string_view and span

`std::string_view` and `std::span`, intended to point to externally-owned piece of memory with known length (a character string or a continuous array) is another addition in the modern C++ that I consider a very good idea. `string_view` allows getting rid of null-terminated strings, while `span` can replace passing raw pointer + size when an array of elements is needed. However, with all the existing codebases and libraries, I doubt C++ can enjoy their wide adoption any time soon.

I think this concept could be taken further. This is just a vague idea, but I can imagine a language where raw pointers are used only for backward compatibility with legacy libraries and APIs or for some very low-level and hacky optimizations. There should be no such thing as “pointer arithmetic” apart from those isolated cases explicitly enclosed with some `unsafe{ }` section.

All the other code should only use some form of “smarter” pointers and views. It is generally good to evaluate as much as possible at compile time (the “zero overhead abstraction” in C++), but I think that in case of pointers to an object, array, memory buffer, etc., they could be accompanied by some extra information (also in runtime if needed) to ensure:

- Ownership and lifetime control - like with the `std::unique_ptr`.
- The length of the array - like with `std::span`. This would allow bounds checking.
- The way to destroy the object, like with deleters optionally attached to `unique_ptr` and other smart pointers. This would allow passing objects and their ownership around, also between libraries, even if they were created out of various memory pools and allocators, and still be able destroy them correctly.

Resource access in graphics APIs

If you know graphics programming using any API like OpenGL, Direct3D, or Vulkan, you know that we read and write memory in form of specific resources: buffers and textures. To read or write some data, we specify the buffer to access + offset, not just a raw pointer. Thanks to this:

- Bounds checks are guaranteed - out of bounds reads return 0, out of bounds writes are ignored. This might not be the optimal behavior for the CPU code, but at least we can see the bounds are possible to check and the app doesn't immediately crash. (Yes, I know there are exceptions - buffers accessed through root descriptors in DX12 are not checked.)
- Compiler can better reason about our shader code and optimize it, which is important considering the complex nature of the GPUs, which have many types and levels of caches that are not coherent and require automatically flush/invalidate.
- Finally, the code is higher level and more explicit about what we do.

There is a direction in the development of graphics APIs to move towards more free-form pointers, like the VK_KHR_buffer_device_address extension in Vulkan. Some people anticipate it, others are afraid of it. It will make it easier to develop some kinds of algorithms and data structures (like linked lists, trees), but it may also become a source of more bugs, crashes, and make debugging more difficult.

Even shader compiler doesn't have all the information, especially when it crosses the realm between the CPU and the GPU code. That's why DXC shader compiler provides `-res_may_alias` parameter indicating that some resources declared as separate may really point to the same place in memory (alias). In C++ it is the opposite - the compiler assumes any pointers can alias (which may prevent some optimizations) unless we use `__restrict` keyword. It would be good if the compiler could infer that two pointers don't alias and optimize accordingly in as many cases as possible.

These examples demonstrate that having higher-level language is not contradictory to optimizations that result in high-performance code, quite the opposite. I believe that more high level knowledge about where the memory comes from (like the buffers and textures in shader languages) rather than using just raw pointers would allow the compiler to make the code both safer and more efficient (optimized better), including even far reaching ideas of auto vectorization, parallelization, and optimization separate from logic.

Eric Lengyel has some interesting thoughts about this. His idea is something opposite from mine - he proposes to make the compiler more restrictive (so to optimize less), but add a new keyword `disjoint` that would declare a pointer has some unique memory that doesn't alias with anything else. I like the idea that the newly allocated object has this attribute by default. I just think it should be tracked automatically by the compiler and be the default.

Passing by value or pointer/reference

The fact that objects can be created on the stack or dynamically allocated on the heap is generally a good thing and interacts nicely with the manual memory management. The choice of passing an object by value (creating a copy) or by reference as a function parameter is even better.

Some types are better passed by value and copied every time: built-in types like `int` and small custom structs like `struct Vector { float x, y, z; };` Others should not support cloning at all and only be passed by reference, like a class representing the entire application backend and owning a database connection. But where is the line? Should a structure representing a client with his name, address, and few other member variables support cloning, or always be passed by reference? How about arrays (vectors)? How about strings? If you pass a string to a function and modify it inside, should it reference the original string, or a copy?

```
function Foo(s: string)
{
    s = "New string";
}
```

Some languages like Python avoid this question by defining some types (like strings, tuples) as immutable. Some languages let us choose whether a struct will be passed by value or by reference while declaring it, like the distinction between a `struct` and a `class` in C# or the `Copy` trait in Rust. I don't like these. I think we can do better by supporting all 3 modes explicitly in the place where we use an object (not where we define it), with some convenient syntax.

1. Passing by value with making a copy.
2. Passing by a mutable reference to the original object.
3. Moving ownership.

This is all supported by C++, which is why this topic is mentioned in the "good parts" section. Only the syntax of `std::move` is clumsy, which stems from the aversion to introducing new syntax due to backward compatibility. It a new language, it could look like this:

```
function F1(s: string)
{
    s += " my addition...";
    // I modified a copy and I can use it like a local variable.
}
function F2(s: &string)
{
    s += " my addition...";
```

```
    // I refer to the original string, modifying it.
}

var s = "My string";
F1(s); // s was not modified, a copy was made.
F2(&s); // s was modified.
F1(move s); // Ownership of s was moved to the function.
// We cannot use s beyond this point.
```

This example shows that explicit ownership is something that Rust does right. However, one more feature that would make the new language convenient and efficient at the same time is avoiding copies when they can be optimized out. Think about the situations in C++ when we pass an object by `const&` instead of by value, or we define a reference/pointer parameter instead function return value to avoid creating a copy. I consider these a failure of the language. They shouldn't be every needed. This is a low level stuff that we shouldn't need to think about - it should be optimized automatically.

I'm not talking about any reference counting or copy-on-write. I think these are bad ideas to think about here. I'm talking more of how functional languages do it, where passing a returning new "logical" copy of a whole list is natural and gets optimized by the compiler when the copy is not really needed. Return Value Optimization (RVO) and Named Return Value Optimization (NRVO) performed by modern C++ compilers are steps in a good direction, but they are far from perfect. Consider this example:

```
std::vector<int> Foo(std::vector<int> vec, int queries_left)
{
    if(queries_left == 0)
        return vec;
    int a;
    std::cin >> a;
    vec.push_back(a);
    return Foo(vec, queries_left - 1);
}

int main()
{
    std::vector<int> v = Foo({}, 5);
    // ...
}
```

As you can see in Compiler Explorer, it is not optimized well, even with GCC flags `-std=c++20 -O3 -flto`. I would call the compiler any good if this code got optimized to the equivalent of the following code, with:

- tail recursion turned into a loop,
- iteration count inferred to be 5,
- and, most importantly, no copy and no dynamic allocation of the array needed beyond the initial initialization.

```
int main()
{
    std::vector<int> v;
    v.reserve(5); // The only dynamic allocation!
    for(size_t i = 0; i < 5; ++i)
    {
        int a;
        std::cin >> a;
        v.push_back(a);
    }
    for(auto i : v)
    {
        std::cout << i << std::endl;
    }
}
```

This is especially important for strings, which we should be able to pass to functions by value without worrying about making a copy and a new dynamic memory allocation until we try to modify them. But this requires the compiler to know that a string is a string and a vector is a vector, not just seeing a bunch of random pointers, like the C++ compiler does…

Iterators

The iterator design pattern is generally a good idea. It offers a convenient abstraction for traversing, searching, accessing, even inserting and removing elements from various kinds of containers. It is especially good in the containers where direct indexing with `container[i]` is not available or efficient, like linked lists or trees. The idea of providing a unified interface of various types of containers so we can apply STL algorithms to them (like `std::lower_bound`, `std::accumulate`) also looks good on paper.

However, iterators as implemented in C++ STL have multiple problems:

1.
Slow performance in debug builds, mostly because compilers in Debug configuration don't inline functions. In many applications this is a deal breaker. We even cannot use iterators, or cannot use debug builds, because they would work too slow.

2.
Unsafe when invalidated. Consider following code that traverses a vector and inserts value 100 after every element that is even:

```
std::vector<int> v = ...
for(auto it = v.begin(); it != v.end(); ++it) {
      if(*it % 2 == 0)
          v.insert(++it, 100);
}
```

The code works perfectly when the vector has enough capacity, but when it needs to be reallocated to grow in size, it will crash, because a vector iterator contains a pointer to the allocated memory, which gets invalid in this case. There are runtime checks for this in the Debug build (which surely causes extra memory and performance overhead), but there is no way to see this problem at compile time. Containers like `std::list` or `std::set` don't have this problem, only `std::vector` does. So the iterator here is a leaky abstraction, and a dangerous one.

3.
Another reason why using iterators with vectors is not a good idea is the inconvenient syntax. Any other programming language offers methods to insert or remove elements at specific place by just passing an index:

- C#: `List` methods `Insert`, `RemoveAt`
- Java: `ArrayList` methods `add`, `remove`
- JavaScript: `array` method `splice`
- Python: `list` methods `insert`, `pop`
- Even Rust, which also supports iterators, offers `Vec` methods `insert`, `remove` that take index as parameter.

Only in C++ we need to use this clumsy and verbose syntax like:

```
my_vector.insert(my_vector.begin() + index, value)
```

Operator overloading

I think that the possibility to overload operators for user-defined classes is a good thing in general. Sure, it can be overused, like with someone overloading `+=` to add a widget to a window in a GUI library, or when the designers of the standard C++ library overloaded `<<` to send data to an output stream. But for classes and structures that represent something like a matrix, it is preferable to be able to write `a*b` rather than having to call a function `mul(a, b)`.

Syntax of overloaded operators

When it comes to the syntax, I think that Python does this better than C++ by asking programmers to call these methods in the way like `__add__`, `__lt__`, etc. rather than `operator` plus the actual operator in C++. It surely makes the parsing easier, but it also looks better. Compare those two possibilities:

Real C++:

```
void operator()() { ... }
```

Syntax modified based on Python:

```
void __call__() { ... }
```

The bad parts

Things that C++ does poorly:

Backward compatibility

The main reason that makes C++ so bad and unfixable is probably its backward compatibility - will older versions of C++ before they added new useful facilities like `std::unique_ptr` or `std::span`, but also all the way to the 40-year-old history of C.

Of course, backward compatibility has its pros. It definitely helps the adoption of the language. For a new programming language to be used for whatever purpose other than small hobby projects or new projects starting from scratch right now, it needs to interop with the existing codebase. However, there are other possible ways a new programming language could take to ensure this, other than having a backward compatible syntax. Some vague ideas:

- The new language could be transpiled to C or C++. It sounds crazy, but TypeScript, which is transpiled to JavaScript, showed that is is feasible in the places where we cannot simply switch to a new language. It could be the best way as a temporary solution to create a compiler quickly and to interop easily with existing codebases.

- A translator could be made to automatically or semi-automatically translate C++ to the new language.
- The new language could be compiled to .obj code object compatible with C++ compilers.
- Some efficient binding code could be automatically generated to interact between the languages.
- Some new, intermediate representation could be defined (on the level like LLVM IR) where both languages could meet before being compiled to the native code.

Aversion to introducing new syntax

I think it is generally a good idea for the new language to use familiar syntax similar to C and C++ (a path that Java, JavaScript, HLSL, GLSL, and countless other languages took). See also Syntax.

But whenever a new feature needs to be added to the language, the need to retain backward compatibility makes the designers of C++ adverse to adding new keywords or other syntax elements that could break existing codebases. Even something as simple as defining new keyword is often replaced by either:

- inventing more and more elaborate clusters of symbols, (`->`, `::` are obvious examples),
- repurposing existing keywords and symbols for new uses (like starting an expression with `[]` to define a lambda),
- or defining symbols that look like functions from the standard library (which they often are), like `std::move`.

They are probably afraid that some developers would need to do search & replace in their codebase to rename the identifiers that now became keywords, which will cost them millions of dollars 😉 This approach is reasonable, but it is not good for the language development.

`auto` is an example of such old keyword that received a new meaning, but its syntax is usage is actually pretty good. There are many examples which are not so good:

- Just adding templates with comparison operators `<>` repurposed as brackets created a problem for parsing, which must disambiguate whether we are closing nested brackets like `std::list<std::vector<int>>` or we use the arithmetic shift right operator like `num >> bits`.
- The syntax of lambdas doesn't look nice, which can be exemplified by `[]{}();` - a correct code that declares an empty lambda and calls it in place.
- Splicer operator `[: xxx :]` proposed for reflection in C++26 looks even weirder.
- Last but not least, naming new things that are fundamental to the language like they were just some random functions from the standard library, best example being `std::move`.

I think a better approach in a new language would be to:

- Introduce new syntax and new keywords whenever it makes sense. Short single lower-case English words should be allowed to be reserved as keywords regardless of the context and don't require any prefix like `std::` If we all agree to have `if(condition)`, why not have `move(obj)` or `clone(obj)`, or even `move obj`?
- Use more of the symbols that we all have on the keyboard but are not used in C++ today, like backtick `` ` ``, at `@`, dollar `$`, hash `#` (assuming we don't have the preprocessor in the new language). C++/CLI language did a nice thing defining syntax like `String^` for its new type of reference. C++ chose to go with `string&&` for its new r-value reference, which is longer and looks similar to the old references.

Preprocessor

Preprocessing the source code on the token level before the actual compilation is a tempting idea. It is so flexible it can be used in many different ways. But in practice, we know it also causes many problems.

- It is a separate language with its own syntax, very limited, clumsy, with no type safety and other checks intrinsic to C++, so it is easy to introduce bugs.
- Macros pollute global namespace, need prefixes like `MYLIB_` before every name. Otherwise we have unpleasant side-effects like WinAPI headers redefining an identifier as generic as `GetObject` to become `GetObjectW`.
- Source code stuffed with macros may become difficult to work with for IDEs that try to provide autocomplete.
- When a mistake is made, compiler error messages may be difficult to interpret and far from what is really the culprit of the failure.

If we think from a broader perspective about what the processors tends to be used for, we can see there is a spectrum of things we may want to do with the code. On one end, we have things that we can express in modern C++ itself (like defining constants and inline functions instead of macros). On the other end, there are cases where even the preprocessor is not sufficient and we need to rely on some custom preprocessing or generation of the code. Many major frameworks do it, including Qt and Unreal Engine. The preprocessor sits somewhere in between, where it merges the disadvantages of both worlds.

I am not entirely sure what the ideal solution should be. I think it would be preferable to be able to preprocess the source code using the same language, not an entirely new one and a limited one, like the C/C++ preprocessor does.

Reflection

This, however, would require some form of reflection - the code being able to reflect on itself, at least in compile time. For example, being able to iterate on available structures and their data

members of various types, or enums, and generate some code based on them would allow generating the code to:

- serialize them to some binary format to store in a file or send over the network,
- expose them to some GUI for user manipulation in form of a “property grid” window.

Many programming languages offer reflection, especially the high-level/scripting ones like Python, C#. C++ is going to get them in version C++26, and Herb Sutter believes this topic will dominate the next decade of C++. I think that a successor of C++ should also support reflection, with the goal of being able to replace most cases where custom preprocessing/parsing/generation of the source code is needed today, while avoiding the disadvantages of the C/C++ preprocessor.

Null-terminated strings

I think that null-terminated strings are a very bad idea. They may be the worst idea ever in the history of computer science. Their disadvantages, compared to storing the length explicitly, include:

- Calculating the length is an O(n) operation - requires iteration over the whole string.
- Character `'\0'` is excluded from the allowed values of the characters.
- Forgetting about the null terminator causes an unbounded number of bytes to be read or written beyond the buffer, which may cause a crash or constitute a security vulnerability.

I cannot imagine how saving on those 4 or 8 bytes for storing explicit length could justify such disadvantages. Storing strings as null-terminated is not necessarily a universal standard. For example, Pascal language had length-prefixed string since 1980s. It is just the realm of C that has this problem. I think that null-terminated strings should be used only when compatibility with legacy libraries and APIs is needed.

Compiler optimizations

However controversial it may sound, I think C++ compilers are really bad at optimizing our code. Many of you may disagree. I admit compilers are great in optimizing single arithmetic instructions and doing bit tricks. What I mean is optimizations on a higher level.

I would call a compiler good if it optimized following code:

```
std::string GetName() { ... }
int GetLuckyNumber() { … }
std::string F1()
{
    std::string s;
    s += "Hello ";
```

```
    s += GetName();
    s += "!";
    return s;
}
int main()
{
    std::string s = F1();
    s += std::format(" Your lucky number is: {}", GetLuckyNumber());
    std::cout << s << std::endl;
}
```

To the following, or something similar, with doing no intermediate copies or reallocations of the string we are building:

```
int main()
{
    std::cout << std::format("Hello {}! Your lucky number is: {}",
        GetName(), GetLuckyNumber()) << std::endl;
}
```

As you can see in Compiler Explorer by the number of calls to `<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::append`, this is not what happens, even with `-O3 -flto` compiler flags.

Of course, compiler developers are not stupid. They are very smart people. There are entire PhDs written about compilers. If you ask one of those people why they don't optimize such things, they will tell you that the compiler cannot infer and prove certain things, so they must hold back the optimization for the program to be correct in all cases. In my opinion, such response uncovers a flaw in the design of the language, and its wrong level of abstraction.

Of course, if we want better optimizations but also faster compilation time, those two goals are contradictory. There is a balance to be found between them, or a spectrum of various compilation settings that we may want to choose as the right tradeoff, from faster builds (when we iterate on debugging, testing features) to more optimized code (when we want to profile the performance or produce the final build). But currently C++ is bad at both. Debug builds are extremely slow and don't optimize even the simplest things like inlining `std::vector<T>::operator[]`, while Release builds are not good in higher level optimizations, as presented above.

Pushing things to standard library

One of the reasons why the compiler has so little high-level knowledge about our code is the approach of C++ language designers to push as much as possible to be implemented in C++ as part of the standard library instead of making more things built into the language syntax.

I believe that strings, vectors, lists, hash maps, and other containers, along with string formatting, streams of text and bytes, and many other things should be part of the language. Of course, this would require everyone using the language to agree on using them instead of implementing their own versions. If the compiler knows that something is a string or a vector, it could optimize better. Currently in C++, from the compiler perspective, everything becomes just a mess of random pointers, so no wonder it cannot do much optimizations on such code.

Standard library is not a standard

On this topic, it deserves a separate chapter to complain about the standard library that comes with the C++ language. Modern versions provide a lot of ready facilities - strings, vectors and other containers, and much more. They are not bad. The biggest problem with them is that many developers don't use them. Instead, every major framework like Qt or Unreal Engine provides its own classes for those things - from a way of returning errors (whether using exceptions or not), through container classes, strings, also a custom CPU memory (heap) allocator, down to the most basic functions like assert, `min`, and `max`.

This is bad, especially because it prevents interoperability between libraries on a level higher than old plain C. This basically reverts all decades of new C++ developments and leaves us with 40-year-old approach to API design.

- We cannot develop a library taking `std::vector` or `std::string` on its interface if we know some users don't like and don't use such classes, but instead they have their own. We need to fall back to raw pointers and null-terminated strings.
- We cannot report errors in our library by throwing `std::exception` if we know some users completely disable exception support in the compiler options for their project.
- We cannot pass ownership of an object to inside the library to be destroyed there when not needed, neither we can receive a new object from a library without a matching library function that would delete it. Otherwise, we risk a crash if the memory allocator used across those two worlds is not the same. For example, FormatMessage function from WinAPI asks us to release the string using function `LocalFree`, while functions from the DXC shader compiler library (also by Microsoft) like IDxcCompiler2::CompileWithDebug require releasing the returned string with function `CoTaskMemFree`.

This adversity to using standard facilities of the language and the tendency to implement our own everything may not be called a flaw in the language itself, but rather the way it is used. At the same time, it is one of the biggest problems with C++. No other language has such problem.

I haven't seen anyone implementing their own string class in Java or Python or C# to use it everywhere throughout their project. Why developers who use C++ do it?

- Because the standard implementation is not good enough in many applications. For example, see this part of the talk "Heaps Don't Lie Guidelines for Memory Allocation in C++" by Mathieu Ropert from Code::Dive 2024 conference. According to him, there is an allocator better than the default - mimalloc, and it's made by Microsoft, still offered on GitHub instead of just making it the default in MSVC.
- Because they can - because the language is powerful enough to make their custom implementations looking and performing as well or better than the standard one.

When designing a new language, if we cannot agree on using one implementation of a hash map, one policy of growing dynamic array, one heap allocator, then the internal implementation should be possible to swap for a custom one for the entire project or a piece of it, but at least it should still happen under the same interface of those facilities, so the code is interoperable between various developers, libraries, and frameworks on the high level.

Error handling

Error handling is a complex topic. I don't have a ready answer about how to do it right or a clear syntax to propose, but I have some thoughts about it.

Ways of reporting errors

Returning error (failure) of an operation is another thing that is not standardized in C++. Every API, every library has its own way, which is bad.

1. Probably the worst option is to set some the error code in a **global variable** that is so easy to check in an incorrect place or just ignore. Examples are `errno` in standard C library and `GetLastError()` in WinAPI.

2. Many APIs define their own set of **error codes to be returned** from every function, like `HRESULT` type used in DirectX and the entire COM, or `VkResult` in Vulkan.

A variant of this approach that I consider quite appealing is returning a variant type that carries either the result of the function or an error code, like the `Result<OkType, ErrType>` in Rust. Example in Rust:

```
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 { Err("Division by zero".to_string()) }
    else { Ok(a / b) }
}
...
match divide(a, b) {
```

```
    Ok(result) => Use result...
    Err(error) => Handle error message...
}
```

3. In modern C++, the recommended way of error handling is **throwing exceptions**, similarly to how most high level languages support them (Java, C#, Python). In C++ a value of any type can be thrown, but there is a hierarchy of classes intended for exceptions, starting from `std::exception`. This would be good if we all agreed to use them, but as with many other aspects of C++, there is no common agreement, for several reasons:

a) Safe use of exceptions requires consistent use of RAII for the management of memory and other resources. Otherwise, an exception thrown in some place in the code, interrupting our control flow and unwinding the stack, would cause resource leak if we free the resource manually somewhere later with `delete ptr`, or `obj->Release()`. Not everyone agrees to use RAII. Many developers stay with the old manual resource allocation and deletion.

b) Exceptions are slow, not only when they are actually thrown and caught, but even when they are just enabled. Contrary what some people claim, developers from the gamedev community keep proving over and over again that simply disabling all exception support in the compiler gives the code a performance boost large enough that it cannot be neglected.

c) Finally, the fact that we don't all agree to use exceptions or any other specific method of returning errors makes them impossible to use in a C++ library, which makes every library developing its own way of error reporting, and the vicious circle continues.

The full error experience

If we want to be diligent about reporting errors, how should it look like? Consider following call stack:

```
int StrToInt(const std::string& s)
void ProcessConfigLine(uint32_t num, const std::string& line)
void LoadConfigFile(const std::string& file_path)
void InitApp()
int main()
```

When the innermost function fails to convert a string to a number, it can throw an exception. What message should it carry? What message should be displayed to the user in a MessageBox or saved to a log file for the system administrator?

I think that in order to carry the full information needed to understand the issue, we need **a stack of strings**, printed in the order opposite to the order of stack unwinding as the exception

was thrown. Such multi-line error message can be read as if every line was preceded by the word “because”. In this example:

Failed to initialize the application.
[because] Failed to load configuration file from "config/UserSettings.ini".
[because] Failed to process configuration line 42.
[because] Failed to convert "aaaa123" to number.

Notice that printing only the outermost error message (“Failed to initialize the application”) or only the innermost message (“Failed to convert "aaaa123" to number”) wouldn’t give enough information to quickly locate and fix the issue.

Simply putting the name of the function on the stack (“LoadConfigFile”, “ProcessConfigLine”) wouldn’t suffice either. We need to put this line number and this file path formatted into the string with error message.

This approach to error handling is already available in C++ and other high level languages by throwing, catching, and re-throwing exceptions. Many languages support chaining exceptions in form of “inner exception”, including Python, Java, C#.

Levels of error handling

Allocating, formatting, and passing such stack of strings is a heavyweight solution not suitable for every application or for place in the code. We can think of a whole spectrum of various levels of error handling:

1. Disable any error checking. Performance is critical. We don’t expect anything to fail. If anything fails, we are fine if the whole program crashes.
2. Just return error code. Performance is important, we know something can fail here, we need to handle it and we know how to do it.
3. Check all the errors and report them diligently so we can show them to the user or save them to a log (like in the example above).

Which level to choose depends on the type of application. For example, if a game fails on almost any of its operations, from opening a file with graphical assets (textures, 3D models), through allocating memory for them, to rendering graphics on the GPU, then it can basically crash - there is not much else to do. On the other hand, a GUI application trying to open a file by user request should handle any failure gracefully and provide as much information about the failure to the user as possible. For example:

Failed to open file "\\MyNeighbor\SharedFolder\Document1.docx".
Network connection failed.

However, which level of error checking to choose is not necessarily known in a specific place of the code. Should function `StrToInt` ignore any error returning undefined value, return some

error code on failure, or throw an exception with a verbose error message? It depends on the piece of code that uses this function. If we use it in a performance-critical loop to go over 10000 strings trying to convert each one to a number, it should probably be as efficient as possible, just returning an error code. However, if we use it to load few configuration options and we absolutely need to succeed for our application to continue loading, we may prefer the full exception path, like presented in the previous section.

Proposed solution

Thus, I think a preferred solution would be to provide a syntax for extensive error handling as part of the language, including error codes and string messages, but also provide some control over the level of error checking that the compiler should perform for a piece of code in the outermost scope (call site). The syntax might look somewhat like this:

```
function StrToInt(s: string): int32 {
    var result: int32 = 0;
    for(ch: char in s) {
        if(ch >= '0' && ch <= '9')
            result = result * 10 + (int32)(ch - '0');
        else
            throw ERR_INVALID_VALUE,
                format("Failed to convert \"{}\" to number.", s);
    }
    return result;
}
function ProcessConfigLine(num: uint32, line: string) {
    try {
        // Processing line, calling StrToInt...
    }
    catch {
        throw ERR_INVALID_VALUE,
            format("Failed to process configuration line {}.", num);
    }
}
...
function main() {
    ErrorHandling(Full) {
        InitApp();
    }
}
```

Depending on the declared mode of `ErrorHandling`, the compiler would compile all the functions called recursively with:

- error handling involving formatting error message strings,
- just returning the `ERR_INVALID_VALUE` error code,
- or with all error checking disabled completely.

Assertions

Assertions are special kind of error checks that assume some expression should always be true in some place of the code. When the condition is not met, it usually means programmer's bug, inconsistent state of the program, not just incorrect input data that we should handle gracefully. Thus, assets may trigger immediate termination of the app with some debug message, break into debugger with `__debugbreak` intrinsic (which is really doing `int 3` interrupt), write a message to some log, etc. In optimized Release builds asserts are typically removed to improve the performance.

Asserts are great idea in general, but in C++ there are following problems with them:

1. No standard

There is `assert()` macro defined in `<assert.h>` header in the standard library, but many developers, many apps, and frameworks define their own to use throughout their codebases. Just like with the standard library that is not a standard, this is a problem unique to C++ that makes writing libraries and interoperability between various pieces of code difficult. I think in a new, better language, we should have one standard `assert` that everyone would use and just be able to modify its internal implementation - globally for the project or locally for some module.

2. Syntax

I think Python got it right with "assert" being a keyword, and an optional text message possible to attach:

```
assert my_num > 0, "The number should be greater than zero here."
```

The new compile-time assert added in C++11 is not much worse with syntax:

```
static_assert(my_num > 0, "The number should be greater than zero
here.")
```

Only the old plain assert stays the same since the ancient times of C with ugly hacks like this required to attach a string message to its content (abusing the fact that a string is really a pointer that has non-null numeric value, so logical AND with it doesn't change the result):

```
assert(my_num > 0 && "The number should be greater than zero here.");
```

3. All or nothing

Enabling asserts is always a tradeoff - they offer extra safety checks during development, but the also cause extra performance overhead. But not all asserts are equal. Some are triggered only once per app startup, other are called thousands of times per second. I've seen some projects defining a separate `heavy_assert()` macro for checks in performance critical code like `vector<T>::operator[]` and I think this is a good idea.

Assert enablement should be configurable (enabled/disabled or more specific configuration of their behavior) not only globally for the entire app, but also per module, and per "level" with at least normal `assert` and `heavy_assert`, to be enabled for extra slow but extra safe code only when we really need it, possibly in some ExtraSafeDebug compiler configuration.

SIMD usage

Another thing not standardized by C++ is vector math using SIMD instructions like SSE/AVX on x86 and Neon on Arm. There are many libraries and APIs out there providing access to these, starting from types like `__mm128` and intrinsic functions `_mm*` available in the Visual Studio compiler, through separate libraries like DirectX Math, GLM, to every major framework dedicated to graphics or games like Unreal Engine defining their own everything.

If we agree that fully automatic vectorization is not possible, then, I think, explicit access to vector math should be part of the language and we should all agree to use it. Basic operations like per-component addition should be available as an operator `a+b` instead of the more verbose and less readable function call like `_mm_add_ps(a, b)`. Of course, some more advanced operations, like `_mm_blend_ps` are hard to express with operators, do they need to stay as functions.

Vector math is something that GPU shader languages do right. They provide:

- vector types like `float4` in HLSL / `vec4` in GLSL,
- per-component computations using basic operators, like `a+b`,
- compound types like matrix `float4x4` in HLSL / `mat4` in GLSL representing 4x4 matrix (or 4 vectors) with additional `row_major` or `column_major` attribute,
- shuffle and write mask syntax.

The last feature looks like this, and is impossible to achieve in C++ by just operator overloading.

```
// Select just components (y, z) from a 3D vector.
float2 my_vec_2d = my_vec_3d.yz;
```

```
// Assign just components (x, y) of a 3D vector.
my_vec_3d.xy = float2(1.0, 2.0);
```

I think a good CPU programming language to replace C++ in applications that involve 3D math, like graphics, games, robotics, physical simulations, etc. should be not worse than GPU shader languages in this regard.

For machine learning and other compute-intensive code doing lots of matrix/tensor math, it would be also good if the language supported advanced slicing like in Python, where we can access an array with `a[1:10:2]`, meaning elements from index 1 inclusive to index 10 exclusive, taking every second element. Such thing should be part of the language syntax, compiled to an optimized code, and efficient even in Debug builds (which C++ cannot provide if we implement it as a library with overloaded operators and templates).

If we cannot agree on a common implementation of the vector math library, then, at least, we should all use a common API built into the language, so that we can exchange vector/matrix/tensor data across modules and libraries, while the implementation could be swapped for the entire project or a specific module.

No common ABI

One of the biggest flaws of C++ is the lack of a common Application Binary Interface (ABI) that would ensure compatibility between .lib and .dll libraries and the .exe file that uses them. To be compatible at the C++ level, they need to be compiled using the same compiler in the same version. Otherwise, low-level implementation of how things are passed in memory (e.g. the layout of `std::string`, `std::vector`, etc.) may be incompatible.

Common solutions are:

- Distribute libraries in the source form whenever possible (when they are open source), but this involves all the issues that exist in C++ related to managing libraries and dependencies.
- Provide the interface in plain old C with structs, global functions, pointers, and null-terminated strings, which is ABI-compatible.

The fact that we cannot agree on using the standard library, one way or error handling, and the specific implementation of many other things also doesn't help and pushes us further into preferring pure C interface or every library defining its own way of doing everything.

I know C is the "lingua franca" of the native code. It also has an advantage of easy binding to other languages. But at the same time it neglects decades of advancements in C++ and other high level programming languages and requires sticking with the low-level, 40-year-old, inconvenient, and unsafe C.

One workaround is having library interface in C and providing a convenient object-oriented C++ wrapper. This is what Vulkan does in Vulkan-Hpp project. if you look at vulkan.hpp file, you can see 21k LOC of code doing nothing but resolving the issue of the narrow C-style bottleneck between a high-level C++ code that uses the API and the high-level C++ code that implements the API in the graphics driver.

Of course, not having a standardized C++ ABI and allowing any version of any compiler on any platform to do everything its own way has an advantage of allowing new and platform-specific optimizations.

COM

Component Object Model (COM) designed by Microsoft is probably the closest to be called a standard ABI compatible between various apps and libraries, standardizing classes (actually interfaces), error handling, and many other things.

However, it stays in the realm of Windows, with libraries in this style developed mostly by Microsoft, i.e. DirectX and various parts of the WinAPI. It also has its own problems, including the difficulty in developing new components, and the runtime overhead involved in the mandatory dynamic allocation and reference counting of every object. Thus, it is quite far from what a high performance native language like C++ really needs.

Intermediate representation

Could we do better? I am not sure, but I can imagine some new, common intermediate representation introduced to a native programming language. I'm not talking about the intermediate code like in Java or C# that would require some runtime and a JIT compiler installed on the client machine. I'm talking about a representation like LLVM IR that would be consumed by the final stage of the compiler and linker before producing the native, compiled binary.

Such intermediate representation would allow the distribution of (half-)compiled libraries without sharing the source code, yet to be compatible with various compilers and platforms. If high enough level, it would also allow passing all the types of objects like strings, vectors, and other containers, memory allocation and ownership between functions and classes of different modules. if we cannot agree on using one specific implementation on those things (like the heap memory allocator), then the low-level implementation of those could be provided to the compiler at this stage - when converting this intermediate code to the final binary.

Libraries and dependencies

Every modern programming language provides some standard where software libraries can be put into some packages, with clear definition of dependencies between them, and simple installation. Rust has Cargo. Python has pip. JavaScript has npm.

C++ has nothing. There are tries to fix it, like vcpkg from Microsoft, but overall, libraries are typically distributed in a source form with the integration with user's codebase completely custom and manual. Cmake became de fact standard, but this is just another programming language that everyone uses but everyone hates, and all it does is a series of custom steps that are different for each library. Setting up a new C++ library to be used by our project is typically a lot of work, fighting compilation errors, tweaking compiler settings, predefined macros for some configuration, etc. The fact that standard library is not a standard doesn't help here.

This is a failure of C++. It makes C++ developers avoiding using libraries, "reinventing the weel" instead by reimplementing everything on their own. It also gave birth to to the concept of "stb-style" single-header libraries that can be just added to our codebase without a need to compile a separate project with static or dynamic library that would involve all this hassle.

But this is just a workaround to a problem we shouldn't have in the first place. Sure, a library may need some configuration option (like whether we want to use a version of the library optimized for the GPU with CUDA or not), but it doesn't change the fact that in Python, for example, you can install a library as extensive as PyTorch with just one command, after which you can immediately start using it.

Disadvantages

The simplicity of pulling dependencies has its own disadvantages. When it is easy, some developers may use too many third-party libraries carelessly. I remember using some simple math library in Rust that downloaded and built a library with the entire Windows API and its documentation (which took a lot of time) only to use a function like `GetTickTime` to be able to read the current time for seeding its random number generator. Yet, I think this is a good problem to have. when pulling dependencies is easy, we can always promote a hygiene of doing it as sparingly as possible. When it is hard like in C++, there is no good way to make it easy.

Many ways of doing anything

I don't think it's good to impose a single narrow path of doing everything. Rust compiler throwing a warning if you don't use snake_case naming convention is too much. But C++ goes too far in the opposite direction. Mostly due to decades of backward compatibility all the way to C, there are so many ways of doing anything. Consider all the ways you can try to define and initialize a variable. There is an entire book about it!

```
int i1;
int i2; i2 = 123;
int i3 = 123;
int i4();
int i5(123);
int i6 = int(123);
```

```
int i7{};
int i8 = {};
int i9 = int{};
int iA{123};
int iB = {123};
int iC = int{123};
```

Do you know the difference between them? Which variable stays uninitialized, which is initialized with a value 0 or 123? What if I used a custom type instead of the basic `int`? How many copies of the object would be created? What if that type was a class having some custom constructors? Which constructor would get called? Finally, which of those is actually a function declaration and not a variable?

Here is an even more dangerous example. The first vector contains one element `{5}`, the second vector contains 5 elements default-initialized with zeros `{0, 0, 0, 0, 0}`.

```
std::vector<int> a{5};
std::vector<int> b(5);
```

It would help if the second option required named argument like `b(count=5)`, but C++ doesn't support this simple and useful feature of named arguments, unlike many other languages: C#, Python, PHP.

As the result of so many options for doing anything and so many legacy and not-so-good features, every large project or large company has their own coding guidelines, which define a large number of C++ features that should not be used. The fact that we need to deprecate them this way is another failure of C++.

Compilation time and header files

The fact that we have declarations separate from definitions, declarations need to be known before uses, and the resulting need for using header files and `#include` directive, may be the biggest flaw of C and C++. It makes every .cpp file including, parsing, and compiling megabytes of code, included from headers of the libraries we use. This is likely the main reason why C++ compilation is so slow.

Features of modern compilers and IDEs like precompiled headers, unity builds, or distributed builds like IncrediBuild help with compilation times significantly, but they are just workarounds to overcome the intrinsic flaw of the language.

Of course, the need for declarations and headers can be justified by the way the compiler works, but this is a poor excuse. Other programming languages prove that this is not really

needed. Pretty much no other programming language has such problem. We typically just write the full source of a specific module, declare which functions or classes should become public/exported (visible outside), and import other modules that we want to use.

Let's consider following example. Our class `ApplicationBackend` needs to own an object of another class `DatabaseConnection`, so in a file "ApplicationBackend.hpp" we define it like this:

```
#include "DatabaseConnection.hpp"

class ApplicationBackend
{
public:
    void ExecuteQuery(Query q);
private:
    DatabaseConnection db_conenction_;
};
```

Any .cpp file that wants to use an object of the `ApplicationBackend` class needs to `#include` and parse a header with its definition, which declares not only the public interface of the class, but also its internal implementation (private members). Whenever we change something in the class, even in the `private` part, every .cpp file that uses our class needs to be recompiled. Moreover, the definition of `DatabaseConnection` class must also be known at this point, so this header must include another header with that definition. It cascades further, so that a small change in a private implementation of one class may trigger recompilation of pretty much all the .cpp files in the project. This is a big failure of C and C++. Techniques like "pimpl" try to solve this issue, but they are nothing more than workarounds for a problem that we shouldn't have in the first place.

Some may say that it is reasonable and necessary to know the definition of both classes and all their members recursively, including private members, so we know their sizes in memory, in case we want to create them on the stack or or pass them by value. But if we think about it on a higher level, we don't really need to parse megabytes of textual source code for this. If the language was designed better and compilers were implemented better, this simple information could be cached in some database and only updated when the source file has changed.

C++20 modules

The appearance of modules in C++20 may look like a step in a good direction, but considering all the legacy C and C++ code using old headers and the difficulty of using the new modules, I can't imagine their widespread adoption in the near future. Just look at some learning materials about them, like THIS… "Global Module Fragment" vs "Module Pureview" vs "Private Module Fragment"? Visibility vs Reachability vs Necessary Reachability? Couldn't this be simpler?! I

guess not… Which is yet another proof that C++ is unfixable due to backward compatibility. Besides this, some developers shown than converting their codebase to modules makes it compiling even slower due to the way modules are processed by the compiler.

Slow debug builds

The idea of “zero overhead abstraction” that promises to evaluate things in compile time and execute them optimized as efficiently as possible in runtime fails miserably when we switch to the Debug build configuration. Of course we agree that debug builds can be slower due to compiler optimizations disabled and additional checks like `assert` preserved. However, if we iterate over a vector of elements like this:

```
int sum = 0;
for(size_t i = 0; i < numbers.size(); ++i)
    sum += numbers[i];
```

The code can become 1184 times slower in Debug than in Release, as you can see in this article. This is due to the compiler not inlining function calls and generating even `vector<T>::size()` and `vector<T>::operator[]` as a function called with every iteration of the loop. A version using iterators works even slower:

```
int sum = 0;
for(auto it = numbers.begin(); it != numbers.end(); ++it)
    sum += *it;
```

While a version using raw pointer is only 7.6 times slower in Debug than in Release - a much more tolerable ratio.

```
int sum = 0;
int* dataPtr = numbers.data();
size_t count = numbers.size();
for(size_t i = 0; i < count; ++i)
    sum += dataPtr[i];
```

Having to use old raw (verbose and unsafe) pointers to get a decent performance in Debug is anything but “zero overhead abstraction”. Performance of debug builds does matter. A code working 10x times slower versus 1000x times slower is a difference that can make a game unplayable, and thus - make the debug build completely unusable. This is indeed what happens with many games, where we are not able to work with debug builds, need to compile in Release and only disable optimizations in a specific .cpp file using `#pragma optimize("", off)`.

This is yet another workaround for problem that we shouldn't have. The compiler should provide decent performance also in debug builds by inlining basic things and having vectors and similar types built into the language rather than implemented i the standard library. None of us are interested in stepping into the implementation of `vector<T>::size()` or `vector<T>::operator[]` while debugging our code.

Unspecified sizes of data types

How many bits are there inside `int`, or `short`? We can tell it for a specific platform and compiler, but the standard doesn't define it. This is because C and C++ are intended to run on all kinds of platforms, some very exotic. `int` type was intended to represent the natural size of the machine word. It worked when PCs transitioned from 16-bit to 32-bit processors, but when 64-bit processors appeared, we stayed with 32-bit `int` not to break backward compatibility with the old code. This is a failure.

Even bigger failure are the names of integer data types used in WinAPI. The 8-bit type is naturally a `BYTE`, but then, `WORD` was defined as 16-bit type and stayed this way even on platforms where the real machine word is 32b or 64b. 32-bit integer is then called `DWORD` ("double word") and 64-bit: `QWORD` ("quad word"). Why call 16-bit integer the "word" despite the smallest addressable unit of data is 8b and the real machine word is 64b? There is no good reason other than decades backward compatibility.

The fact that every major framework defines its own names for the fundamental data types is a failure in itself. Besides WinAPI, we have custom types in Unreal Engine (`int8`, `int16`, `int32` etc.), in Qt (`qint8`, `qint16`, `quint32` etc.). No such thing happens in other programming languages.

There is also no agreement whether sizes expressed in bytes or the number of element in an array should be stored as a signed (like in C standard library) or unsigned integer (like in C++ standard library). I vote for unsigned, because whenever negative numbers are not needed:

- It is not good to waste half of the available range by using signed numbers.
- With unsigned numbers it is enough to check if `index < count`, no need to additionally check if `index >= 0`.

A counter argument is that with unsigned numbers it becomes harder to write `for` loops that count down to 0 inclusive, but there are elegant solutions for this possible and I believe they should be available in the language syntax, like in Python:

```
for i in reversed(range(count)):
```

This should all really be standardized and types with explicit sizes defined so that everyone uses them. C and C++ try to fix this by providing `<stdint.h>` header with types like `uint8_t`, `uint16_t`, `uint32_t` etc. along with `size_t`, but with all the existing code for those languages, it is too late.

String formatting

Converting numbers and other data types to strings and building complex strings with those data inserted is a fundamental functionality that almost every program needs. If not for showing text to the user, then at least for some debug logging. Modern C++ provides multiple ways of doing it. None of them is good. Consider an example of printing a number in 8-digit hexadecimal form like “0x000058A0”:

1. C printf

```
uint32_t i = 0x000058A0;
printf("Value=0x%08X\n", i);
```

Functions like `printf`, `fprintf`, `sprintf` coming from C are convenient, concise, but they don’t provide type safety. Modern compilers may show some warning, but in general, if you pass too few parameters or mix up their types, you can print random data, crash the program, or create a security vulnerability.

2. C++ streams

```
std::cout << "Value=0x"
    << std::uppercase << std::setfill('0') << std::setw(8) << std::hex
    << i << std::endl;
```

The streams from the standard C++ library are type-safe, but they are bad in a number of different ways:

- The verbosity of the syntax, as you can see above.
- They are known to perform badly.
- They contain a state, so any next number sent to `cout` will still be printed in hexadecimal, with 8 digits, etc. until we restore the original settings of the stream.

3. C++20 std::format

```
std::cout << std::format("Value=0x{:08X}\n", i);
```

The new `std::format` added to C++20 is a step in a good direction. It is type-safe and the syntax is nice and concise. However, it is also known to perform poorly, slower than the previous 2 solutions.

Conclusions

For comparison, here is the same code implemented in Python:

```
print(f"Value=0x{i:08X}")
```

I think a good programming language should contain some string formatting facility. I don't have an opinion about the syntax - whether it should be `format("%i", var)`, or `format("{}", var)`, or just `f"{var}"`. I only know that it should be convenient and concise, type-safe, and efficient. This would probably require (like many other things described in this document) to be part of the language syntax and understood by the compiler instead of being implemented in the language itself as part of a standard library.

Templates

Overall, templates are a good idea. Being able to parametrize a function or a class with a custom type is useful especially for developing containers like vector, list, tree, etc. Moreover, resolving all this at compile time allow for compile-time type safety and good runtime performance, thanks to no need for boxing.

However, the way templates look and how they are used in C++ makes them so bad that we would probably do better without them, assuming we would have good container data types available in the core language that we would all agree to use. There are many reasons why templates in C++ are so bad:

Language within the language

Templates are a separate language within the language that is functional (so a different programming paradigm), based on pattern matching, and non-debuggable. For example, if we want to define a function comparing two numbers, and change its behavior based on some run-time enum, we can write:

```
bool NumbersEqual(float a, float b, CompareMode mode)
{
    if(mode == CompareMode::Equal)
        return a == b;
    else if(mode == CompareMode::Around)
        return fabsf(b - a) < 1e-6f;
    else
```

```
        assert(0 && "Unsupported CompareMode.");
}
```

If we want to do something similar based on the type of numeric parameters (`int` or `float`), we need to make it a template with specializations, which is longer and more convoluted:

```
template<typename T>
bool NumbersEqual(T a, T b)
{
    static_assert(0, "Unsupported data type for NumbersEqual.");
}
template<>
bool NumbersEqual(int a, int b)
{
    return a == b;
}
template<>
bool NumbersEqual(float a, float b)
{
    return fabsf(b - a) < 1e-6f;
}
```

I think it should rather look like below, with the same language used for templates as for the runtime code, with types as top level entities in the language (like in functional languages), and template parameters looking like normal parameters, just with the requirement to be specified (or automatically inferred) at compile time. After all, compiler should infer and optimize statically as much as possible, no matter if we talk about template type `T` or enum parameter `mode`.

```
function NumbersEqual(a: T, b: T, T: type): bool
{
    if(T == int)
        return a == b;
    else if(T == float)
        return abs(b - a) < 1e-6;
    else
        assert false, "Unsupported data type for NumbersEqual.";
}

var b1 = NumbersEqual(1, 1);
var b2 = NumbersEqual(1, 1.0, float);
```

New features of C++ like variadic templates, fold expressions, concepts, `constexpr` and `consteval` are steps in a good direction, but with all the existing C++ code written using templates, it is already too late to significantly improve the situation.

I must admit that having to define specializations separately has one advantage: just like with function overloads, we can define a new version of a function or a class specialized for new types in a different .cpp file, outside of its original location. But do we really need this? It reminds me of the “extension methods” available in C#, which allow extending existing classes with new methods outside of their original location. A useful feature, but we don’t have it in C++ and we are fine without it.

Error messages

When we make a mistake in some code that uses templates, error messages printed by the compiler as often very long, convoluted, and difficult to understand. They tell precisely why the compiler couldn't finish its job, but they tend to be far from indicating what did we do wrong in the code and how to fix it.

Slow compilation

Using libraries that are full of templates is known to increase compilation times significantly, whether due to the amount of code to be parsed or due to all the combinations of template parameters that need to be instantiated for various types. This is another argument against using templates beyond simple applications like the container types.

Abusing templates

The way templates are sometimes used in C++ is so convoluted that it becomes hard to develop, maintain, and fix if a bug is found. Think of all the template metaprogramming as promoted by Andrei Alexandrescu. Think of the implementation of Boost or many other “clever” C++ libraries.

With all the respect to mr Alexandrescu (the ideas of policies or type traits are brilliant), we must admit that using too much templates is not good for a project. It may be good for a particular programmer who writes it (to show off, or for his job security), but in the long run someone else, possibly less experienced, will struggle with maintaining their code.

As an extreme example of how a simple function calculating a distance between two points can be over-generalized and convoluted using templates, look at the Boost Geometry Design Rationale document. For any graphics programmer it just looks ridiculous. Compare it with the documentation of distance() function in GLSL, which is an intrinsic in the language, simple to describe and understand, compiles fast, and works fast. It is limited to 2D/3D/4D vectors of floats or doubles with distance calculated using cartesian coordinates, but it covers 99% of the use cases. I think a good programming language should take the approach like GLSL not like Boost.

Scott Meyers at the very beginning of his book “Effective C++” quotes Petronius Arbiter Satyricon, XCIV: *“Wisdom and beauty form a very rare combination.”* The C++ language fits this quote very well, as it tries to be “too wise”. There are esoteric programming languages out there designed specifically to be unreadable and difficult to use (Malbolge being an extreme case) but at least no one writes serious programs in them…

Enums

Enums are great and this is good that C++ provides them. However, due to backward compatibility we have two kinds of them. The legacy one:

```
enum Result1 {
    Result1_Success = 0,
    Result1_OutOfMemory = 1,
    Result1_OtherError = 2
};
Result1 res = Result1_Success;
uint32_t numValue = res;
```

It has two disadvantages:

- Individual item names are not nested in any scope - they pollute our main scope, so we need to prefix them like we did above so we don’t introduce a global symbol with a generic name like `Success` that could collide with something else.
- Not a full type safety - they can be explicitly cast to integers.

Then, we have the new ones:

```
enum class Result2 {
    Success = 0,
    OutOfMemory = 1,
    OtherError = 2
};
Result2 res = Result2::Success;
uint32_t numValue = (uint32_t)res;
```

This is much better, as it puts the items inside a scope that looks like a namespace and requires prefix `Result::`, and because it constitutes a separate type that requires explicit cast.

However, if our enum is actually a set of bit flags intended to be combined using bitwise OR operator `|` and modified using other bit operators (like `flags |= f` to add a flag or `flags &= ~f` to remove a flag), then none of these would work:

```
Result res = Result_OutOfMemory | Result_OtherError;
```

This results in compilation error because enum items can be implicitly cast to integers to perform the bitwise OR, but then this integer cannot be assigned to `Result`.

```
Result2 res = Result2::OutOfMemory | Result2::OtherError;
```

This results in compilation error because we cannot perform operator `|` on type `Result2`.

What we need to do when coding in C++ is to either define constants and use pure `uint32_t` whenever we need bit flags, where we lose any type safety (which is what most libraries are doing):

```
const uint32_t RESULT_SUCCESS = 0;
const uint32_t RESULT_OUT_OF_MEMORY = 1;
const uint32_t RESULT_OTHER_ERROR = 2;
uint32_t res = RESULT_OUT_OF_MEMORY | RESULT_OTHER_ERROR;
```

…Or use hack that Microsoft prepared for us in “winnt.h” in form of macro DEFINE_ENUM_FLAG_OPERATORS that overloads a bunch of bitwise operators for our enum type.

In a better programming language, none of these should be needed. Ideally enums should work similarly to how they do in Python, so that:

- We can define our type to be enum or bit flags, and these should be distinct types.
- It should be type-safe, so that explicit cast are required from (and to?) an integer.
- Still, bit flags can be manipulated using bitwise operators.
- Full reflection is provided, so we can enumerate available items in the (compile-time) code, their values, and their names as strings.

Safety versus performance

I think that as a general principle **safety should be the default, performance should be opt-in**. Unfortunately, in C++ it is the opposite.

I’m not talking about checking overflow on every addition or multiplication - it would be too slow. I’m also not talking about using a cryptographically safe (meaning slower) hash function in the hash map container by default, like Rust is doing. I’m talking about things like:

- range checking when indexing an array (so we don't cause memory page fault or access some random data),
- initializing local variables (so we don't read random, garbage data, potentially leaking some secrets).

Most code is not in the critical path, while every piece of code can contain some potential bug that can cause a crash or a security vulnerability. If we use a profiler and determine that a piece of code is the performance bottleneck, we should be able to enclose the code with some `unsafe{ … }` section, add some `[unchecked]` attribute to the array or `[uninitialized]` to the variable to improve the performance, which explicitly states that we know what we are doing and we take the responsibility for the unsafety that we introduce in this particular place.

But more importantly, the compiler should be able to prove the safety and optimize out the run-time security checks whenever possible. This should be easy in the following case, where we know the variable is initialized with some defined value regardless of whether the condition is met or not, so we don't need to initialize it with a default value on the first declaration:

```
int i;
if(condition)
    i = 1;
else
    i = 2;
```

This should also be easy when using the new range-based for loop, when the compiler can infer that we iterate over all elements of the array, so out-of-bounds check is not needed:

```
for(auto i: my_vector)
    printf("%i\n", i);
```

However, I would call a compiler really optimizing well only if it could also prove that the following code is safe and doesn't need bounds check for the array access:

```
std::vector<int> vector1 = ...
int value_to_remove = ...
for(size_t i = 0; i < vector1.size(); )
{
    if(vector1[i] == value_to_remove)
        vector1.erase(vector1.begin() + i);
    else
        ++i;
}
```

NonUniformResourceIndex example

Going back to the topic of “safety by default, performance as opt-in”, NonUniformResourceIndex symbol in HLSL language is an example of a particularly bad design.

1. Dynamically indexing descriptors in shaders should be scalar by default - uniform across all threads in a wave.
2. When it is not, we must decorate it with `NonUniformResourceIndex()`, which can cause some performance penalty, because some GPUs need to put a “waterfall loop” around it to scalarize the access.
3. When our index is not scalar, but we forget about this decorator, we make a bug that is tricky to find - everything works fine on some GPUs (particularly on Nvidia cards), but gives incorrect results on others (particularly AMD cards). Many graphics developers make this mistake.

A better language design would be to do the opposite:

1. Assume every dynamic descriptor index to be non-uniform by default (safety first).
2. Shader compilers should try hard to prove whether the value used for descriptor indexing is scalar across the wave (e.g. coming from a constant/uniform buffer, mixed with some immediate constants etc., like `myCB.myConstIndex + 10`) and then optimize it.
3. In those rare cases when the compiler couldn’t optimize it, but we know the index is really scalar, and we determined this is our performance bottleneck, we should be able to opt-in for the optimization by using some `UniformResourceIndex()` keyword, thus declaring that we know what we are doing and we agree to introduce a bug if we don’t keep our promise to ensure the index is really scalar.

Complexity and difficulty

While this problem may be neglected by some developers who already know C++ well, this is not without importance that the language is so complex and difficult to learn for newcomers. Not only it involves so large amount of features and quirks, ever increasing with each version, but it also makes it so easy to make mistakes, introduce bugs, “shoot yourself in the foot”. It is a common opinion that C++ is a difficult language. This doesn’t help for the availability of good programmers who decided to make C++ the programming language of their choice and their career path.

I think a good programming language should be easy to use, difficult to introduce bugs, and provide just enough features to be expressive, concise, and convenient to use, but not too many not to be difficult to learn or to read and understand someone else’s code.

Design by committee

No programming language is perfect and none is universal enough to work well with all kinds of software. There are different languages used to write operating systems, compilers, web

browsers, games, safety-critical software, and different for web or mobile apps. However, C++ is used in may of those applications and designed by a committee where arguments of many kinds of developers meet.

I think that a design by a committee is bad. By observing how C++ develops over the years compared to e.g. C# (made by Microsoft), similarly to how OpenGL and Vulkan are developed by Khronos Group compared to Direct3D (also made by Microsoft), I can see that the technologies designed collectively:

- over time become overly complicated,
- are reluctant to make bigger changes,
- slower to introduce new features,
- overloaded with various bloat trying to please everyone, and become good fit for noone.

For example, lambda expressions were added to Python in 1994, to C# in 2007 with C# 3.0, to C++ in 2011 with C++11.

Gamedev involvement

I was recently told that game developers are not well represented in the C++ committee. This is bad, considering that game/engine development is one of the applications where C++ is commonly used. Maybe this explains why C++ is steering in the direction that makes even simple things like std::lerp unusable from our perspective due to poor performance dictated by too strict requirements, instead of doing a simple `a+t*(b-a)`.

I can understand how small gamedev companies cannot afford dedicating precious time of their programmers to participate in the committee, because they are all crunching on converting Unreal Engine blueprints to C++ to make the game run decently before the deadline, as the publisher said the game has to ship before the next holiday season. But big companies that hire hundreds or thousands of engineers and that move the gaming technology forward, like Valve, Epic, and console platform vendors (Microsoft, Sony) could get involved. They are involved in Khronos to help pushing graphics further. Did they lose any hope in making C++ better? It would make sense, but then, they should think of creating a new programming language instead.

Far reaching ideas

This chapter contains more far reaching ideas, some crazy and wild, some quite vague and difficult to achieve, but nonetheless something I consider good directions to take in the long term.

Optimization separate from logic

Do you know Halide programming language? It may not be very popular, but it introduces one idea that I find brilliant: it separates the logic from the optimization. Intended for digital image

processing, it allows describing the computations to be made, and then separately describing the “scheduling” to be applied on a particular platform to execute those computations efficiently - like vectorization, parallelization, or dividing the image into tiles.

I think this overall idea could be taken further and applied in a general-purpose native programming language, to be used whenever we do some computations on binary data. Of course, it would require working on some higher-level concepts than raw pointers, e.g. objects like `std::span`.

If we all agree that automatic optimization (vectorization into SIMD instructions, parallelization to multiple threads, etc.) is not achievable, then I believe separating the logic from the description of the optimizations is the best thing we can do. If we are ever going to make a big leap in development productivity (not considering the perspective of AI replacing programmers, but in the classical sense like in The Mythical Man-Month book), then, I think, in the world of native programming languages this would be it.

I think that far too much time and stress of very skilled and well-paid developers is consumed by rewriting their compute algorithms (whether for the CPU or for the GPU) to make them optimized for better performance, and then finding and fixing new bugs in this increasingly complex code. It would be much better if the logic could be written once and tested to work correctly, followed by optimizations in form of “*performance hints*” that cannot break the logic, only increase or decrease the performance.

Having optimizations separate from logic would allow a whole variety of methods to optimize the code:

- Basic, naive implementation to be used as a reference or a starting point when enabling a new algorithm or running it on a new platform.
- Default implementation optimized for general case, possibly based on some heuristics.
- Implementation optimized manually using explicit performance hints mentioned above.
- Implementation optimized automatically using profile-guided optimization.
- Implementation optimized automatically using other methods, e.g. machine learning.

Existing examples

It may seem fantastical and impossible, but if we think about it, we can already see some signs of this approach in today’s programming languages, like:

- `inline` in C++, which is a hint for the compiler that we prefer to have this function inlined, while the logic stays the same, and the compiler is free to make its own decision.
- `[unroll]` and the opposite `[loop]` attribute in HLSL, which gives a hint to the shader compiler that we want this loop to be unrolled or not unrolled, which, again, can improve the performance or make it worse, but it won’t break the correctness of once written and tested logic.

- `__mm_prefetch()` intrinsic function and the accompanying cache controlling flags that can prefetch given address into the cache memory before the code really needs it.

SQL is another example of this. Of course, it is a different type of language - a declarative one where we only describe *what* query we want to ask to the database, not *how* to execute it, we don't write the loop that goes over the records. Still, it is not optimized automatically. We can *hint* the database about what columns we are going to search for by creating indexes. They are the hints defined separately from the query. They also cannot break our logic. They can only increase or decrease the performance and memory/disk usage.

Integrated development environment

Traditional approach to programming is to write the code in some text editor (like Vim, Emacs, Notepad), then switch to the terminal and issue a command to compile it. A disadvantage of this approach is not only the lack of an integrated editing, building, and debugging experience, but also the fact that the computer stays idle while developer is coding (apart from the short moments of typing or scrolling), while the developer sits idle while the computer is compiling the code.

Modern IDEs use the time spent in the editor for various tasks done in the background like indexing the code for faster full-text search or autocomplete, and for doing some static analysis. IDEs for Python like PyCharm are good examples, where we can see inline in the code an overlay with useful information like which `import` is unused or how many places call our function.

I can imagine a future where an IDE is doing much more work in the background and showing much more information before we launch the build. The database built for autocomplete would be the same as the compiler uses. By the time we finish typing, the project would be almost completely compiled. The editor would show not only the results of some static analysis and syntax errors underlined inline in the code, but also some tooltips or a separate panel on the right with:

- more in-depth analysis of the code, like the estimated time and memory complexity of the function we call (whether it contains some loops that look like O(n) or worse), whether they are pure functions or have any side effects, whether they involve some file I/O or other lengthy operations,
- disassembly (think of the Compiler Explorer)
- results of the last performance profiling session,

…scrolling hand in hand with our code, so that we would always look at them while developing a high-performance code.

Version control system

Version control systems like Git, Mercurial, Perforce, SVN, CVS… While not directly related to C++ or any other specific programming language, this is yet another topic that I think needs improvement, so I wanted to write down some thoughts and ideas about this.

What we use right now is usually:

Git, which seems to be the winner in terms of the popularity among programmers for storing the source code, whether it's GitHub website or some other server used to host the central repository. However, in my opinion it has many disadvantages:

- It is too complex and difficult to use. It was created in the context of Linux kernel development, so it has this inherent Linux approach of "difficult to use, easy to break" and "we didn't design UX for humans, users should adjust to the underlying technology". Xkcd comic 1597 shows this well. Some of the features are not really needed, e.g. the distributed nature with many *remotes*, while we typically have just one central repository anyway.
- It is not good at handling large repositories with many large, binary files. LFS is not good enough. You still need to have 1 copy of your files as a local copy, 2nd copy in .git subdirectory, and 3rd copy temporarily while pulling an update, so you need disk space of at least 3x the size of the local copy. This is not good. Repositories of large modern games can be as big as 100 GB or more.

Perforce is the solution popular among game developers. It is much better at handling very large repositories full of binary files and a large number of files. However, it has its own flaws:

- Creating branches is a heavy process that we rarely do, which isn't good, because branches are convenient to develop large feature that involve long development time, many commits, or many contributing programmers. Git does it better, because in Git branches are encouraged, lightweight, and natural.
- Changelist is a weird concept. Shelved changelist can only store one version of our files. As we develop a change before it gets submitted, we cannot make small commits and see their history. We only have one version shelved. Sure, we can create new shelved changelists to store multiple versions of our code, but it is as "advanced" as creating directories with dates - just a workaround for a lacking feature. There should be a possibility for local commits that would retain the history.

Hierarchical history

When using Git, we can commit small incremental changes, whether only locally or pushing them to the server. However, when merging our finished implementation to the main branch, we need to decide whether we squash it to a single commit or not. If we squash, the commit history stays clean and simple, but we lose track of the individual incremental changes that we did,

possibly with some interesting code added only to be removed or modified later. If we retain the history of our commits, the commit history of the entire project can become long and convoluted.

I think that in a better version control system we shouldn't need to make such a choice. The solution is to support some hierarchical grouping of commits. Having at least one level of grouping would allow us to see a single commit with the new feature, but if we are interested in details, we could expand it to see individual incremental commits made by the developer. Maybe this could be taken further to an entire hierarchy of groups to enclose larger portions of the history (commits, tags, even entire branches) and give them names, e.g. "Development of version 1.2".

In a new generation IDE like described in the previous chapter, this concept could be taken even further to incorporate basic code editing Undo-Redo history as part of it. Current Undo-Redo mechanics has some limitations and inconveniences:

- If we close a document (unload it from memory), we lose the Undo history.
- If we start pressing Ctrl+Z, we can move back in history of our edits and then move forward with the Redo command (like Ctrl+Shift+Z), but if we accidentally press something else like typing a new letter, we break the sequence and we lose our Redo history.

This would be solved if basic edits and the Undo-Redo history was treated like the most fain-grained level of the commit history, possibly non-linear with unnamed branches when we start typing after Undo. This history could be stored only on the local machine and removed (squashed) after few days, but otherwise it could use the same logic as regular commits and branches.

If we think about the whole problem of managing files during game development, we can see 4 types of files:

1. Game builds

A binary game build, created on a build server, intended to be tested by QA, sent to external partners, finally uploaded to Steam, Epic, and other gaming platforms for end users to download and launch, is typically a ZIP or some other archive. Game assets are additionally often packed in some custom archive format (also called virtual file system). A build can be as big as tens of gigabytes or more. We may need to store multiple of such builds on a local development machine and even more versions on a server. If we notice that most large files (like textures) don't change between builds, the fact that we pack them again per each build is an big waste of space in storage and a waste of time in transfer.

Proposal:
The solution would be to stop packing them in some custom archive formats or .zip archives during development, only pack when really needed for the final build to be sent externally.

Ideally, every version of every file should be stored only once on the server (not counting backups) and only once on each developer's local machine if the user really needs it. Think of "hard links" in the file system. To implement it, we would probably need to go beyond raw disk files and folders and have some kind of a database integrated with the file system in the OS and/or with the IDE + version control system + game editor. Think of how Dropbox and OneDrive make a "magical folder" and virtualize file access today.

2. Binary assets

Another problem is that we typically download new version of all the large files (game assets) when updating source code to the latest version. The code and the assets much match, because some file format may have changed (like a new parameter added to a JSON file), so after updating source code only, mismatching files wouldn't work. Updating all the files can take long time.

Proposal:
A solution would be to have some kind of a distributed file system / mapped network drive plus local caching, so that we can update only the source code, build and launch the game, while large asset files would be downloaded and cached locally only on demand when opened for the first time. Not all 100 GB of the assets are needed for every gameplay session.

If downloading on demand would be undesirable due to long game loading times, maybe there should be a configuration in this software with file masks specified for files to always download when updating the repository, with all other files downloaded on demand. For example:

```
[AlwaysDownload]
//depot/source/...
//depot/assets/.../*.json
```

3. Editor data

There is an unsolved problem of storing files saved by all kinds of visual editors.

> Should they be binary files? Then, we cannot merge them. When two people edit the same file, we need to reject one of the changes.

> Maybe we should lock the file, so only one person can edit it, while others need to wait or they can access it as read-only? This approach has its own problems, like some developer locking a file (or all the files accidentally) and going for vacation, which stops the work for others and requires administrator intervention.

> Should they be text files, so we can merge them? Maybe in XML format? Imagine two developers editing the same scene, each one adding an object. In the text editor we would see following changes to merge:

```
DeveloperA: +<object id="7435" type="Wall">
DeveloperA: +    <position x="253" y="3454" z="0" />
DeveloperA: +</object>
DeveloperB: +<object id="7435" type="Monster">
DeveloperB: +    <position x="-454" y="3556" z="0" />
DeveloperB: +</object>
```

By simply letting merge tools handle the text-based merge of those two changes, we risk at least two bugs:

1. If the id of a new added object is incremented from the previous highest id, then both objects will receive the same id, while it should likely be unique.
2. Merge tools with recognize the line with the closing tag `</object>` as added by both developers, so it will occur in the merged file only once, which will break the XML syntax correctness.

Problem 1 can be solved using random GUIDs instead of incremented identifiers. Problem 2 can be solved using other text formats like YAML. But still, I think this is not the best option.

Proposal:
Meanwhile, businesses other than gamedev solved this problem already. They simply allow “multiplayer” editing of the document by many users simultaneously over the network. Microsoft PowerPoint can do this. Google Docs can do this. Figma can do this. Game editors should do this as well, so we can forget about any need to manually merge the files saved the the editor, just like we don’t ever look inside .docx or .pptx files (despite we can - these are simply ZIP files with content stored in XML).

4. Source code

Editing source code has long history of using text files and some programming language. Many tries of introducing some visual language with interconnected nodes show that editing source code is usually faster, more convenient, and much more concise. Node-based visual languages are good for tools dedicated more for artists (like in Houdini) or designers (like Blueprints in Unreal Engine) who don’t want to learn and use the syntax of a programming language. Still, creating any complex logic in them quickly gets visually complex and difficult to manage. UE4 Blueprints From Hell page demonstrates this well.

The flexibility of organizing the source code in files varies per language. In C and C++ we have large flexibility. We can organize files and folders however we want. We can split the implementation of methods of a single class across multiple files. We can, on the other hand, have just one .cpp file with 100k LOC that contains the source of the entire program or library. (SQLite Amalgamation is an extreme case with the entire library merged into a single 238K LOC .c file.)

In many modern languages, a more strict organization is imposed. E.g., in Python one .py file is one module, which needs to be explicitly imported to be used by another module. The freedom of typing individual characters, line endings, etc. is also increasingly limited as we obey to some written “coding standards” or reformat our files automatically using tools like Clang-Format. In modern IDEs we also often navigate across the code by searching for specific symbols or jumping from one symbol to another instead of opening a specific file and scrolling to a specific place.

The text-based and freeform nature of source code files has some drawbacks that are best demonstrated by the behavior of Git and oher version control systems. Consider these examples:

- When we move a function to a different file, it always shows up as new code added by us.
- When we move a function to a different place in the same file, the behavior of Git depends on how many lines of code we move.
    - Moving a short function from top to bottom of a lengthy file will appear as removing those lines from the top and adding those lines at the bottom.
    - If the function being moved is long enough compared to the code around it, it may instead appear as our function left unchanged, while Git showing that we supposedly added some lines unrelated to our change above the function and removed them from below.
- When we rename a file while also editing it, Git may show that this is the same file renamed or that it is a completely new file depending on how much code we changed inside (similarity index < 50%).

In our everyday work we are so used to looking at the source code as text that we treat is as normal, but if we think about it on a higher level, this is really bad. It can obscure the original author of the code when doing Blame command (in Git) or Time-lapse view (in Perforce).

Proposal:
I can imagine the future when we don’t store the source code in text files, but rather in some hierarchical database of projects, groups, modules, classes, and functions. They could still be shown one after the other, scrolling vertically like one long document. They would still allow selection and editing like a text editor - on the level of individual characters of entire functions or classes. Yet, all edits would be tracked directly by the editor, so that we know where the function comes from even when it was moved from another place or renamed. Of course, this would require an IDE integrated with the version control system always used for editing the code rather than a simple text editor.

About the author

Adam Sawicki: homepage/blog, GitHub, LinkedIn, X (Twitter), Bluesky, e-mail
(remove the __REMOVE__ part of the address before sending e-mail)

Key facts about me, to help you have your own opinion about whether my opinions are worth considering:

I started programming as a kid around the year 1995 and I do it professionally since 2006. Most of the time in my career I work with games, game engines, and real-time graphics for PC and "big" gaming consoles (like Xbox, PlayStation). Runtime performance is very important in these applications.

Because of this, the main programming language I use is C++. I also sometimes code in Python and C#. I have some experience in other languages like C, Java, Lua, PHP, SQL, JavaScript, HTML, CSS, although smaller. I was learning Rust for few months as a hobby, but I didn't write any big project with it. I also have experience programming GPUs in shader languages: HLSL, GLSL. I don't know Objective-C, Go, Swift, Ruby, Fortran, Basic, COBOL, and many other programming languages that are popular according to TIOBE Index.

I have moderate interest in programming languages. On one hand, I am somewhat interested in the language design, as you can see by this document. On the other hand however, I don't consider myself a passionate of learning new programming languages for the sake of it. I treat them as tools that should help us write the software we need to write, make writing safe and efficient software simple, make bugs difficult to create and easy to debug, and otherwise get out of the way in expressing our intents as clearly and directly as possible.

I am not involved in the C++ committee. I don't read or write proposals for the further development of the language. I sometimes learn about the new features added to C++, but I don't try to passionately stay on top of all the latest developments in this language.

I have love-hate relationship with compiler programming. I've been doing it professionally in the past. I still sometimes design and develop hobby languages and parsers, like my MinScriptLang. On the other hand, I think that compiler development is a higher-order hardcore, so I am not sure I would like to do it professionally ever again.