C++ Club UK Meeting 135

Gleb Dolgich

2021-09-09

Contents

Jetbrains: The state of developer ecosystem in 2021

Results, Reddit

The results of the survey are available, together with methodology. More than 47000 people participated in the survey from 23 geographical entities. To clean the data, they identified and threw away suspicious replies, including those that were filled too fast, multiple surveys with overlapping results and/or from the same IP address, surveys with conflicting answers (like a 18-20 year-old with 16 years of professional experience).

Some interesting results:

- The C++ standard people use is mostly C++11 or C++17. 18% already use C++20, which is encouraging news. But then 12% are still on C++98/C++03, which is pretty terrible; of these, 41% don't plan to migrate to a newer standard (*hello game developers!*). 11% respondents replied that they are not sure which standard they use, and I'm not sure how that happens.
- Of C++20 features, 48% plan to use modules in the next year, which may be a tough cookie, as only MSVC has production-level support for modules at the moment. 46% plan to use concepts, and 33% plan to use coroutines. 31% don't plan to use any C++20 in the next year.
- The most popular C++ IDE is Visual Studio Code (28%) followed by CLion (26%), followed by Visual Studio (24%). Vi is used by 6%, whereas Emacs only used by 2%.
- 30% don't write unit tests for C++, which is bad news. Google Test is the most popular unit test framework, followed by 'no framework', then Catch, CppUnit, Boost.Test, and doctest.
- Third-party libraries are most often included in the project (26%) or compiled separately (23%). 21% of developers rely on system package managers, and 21% download binaries from the internet. Package managers, like Conan and Vcpkg, are in the single-digit percentages. And 14% are lucky to not have any dependencies.
- In error reporting stats, 80% use exceptions.
- In build systems CMake is the king (55%), followed by Make and Visual Studio projects.
- In compilers, GCC wins with a whopping 78%, followed by Clang (30%) and MSVC (30%).

MSVC C++20 and the `/std:c++20` Switch

Microsoft announced that the new version of VS2019, 16.11, has a new compiler switch `/std:c++20`.

> The addition of this switch indicates that we've reached a point of sufficient stabilization of the MSVC C++20 feature set for it be used in production, with full support in VS servicing updates.

If you think it means that the new switch allows you to use all the standard C++20 features, you are in for a couple of small surprises.

The new switch signifies a stable ABI. The experimental switch is still `/std:c++latest`. Both imply `/permissive-` for strict standard compliance. You can turn this off, but then some C++20 features won't be available, like modules.

Let's see what are those surprises.

C++20 defect reports were raised during implementation of C++20 due to some late discoveries. Fixes for those issues will be available under `/std::c++latest` and then, eventually, under `/std::c++20`. This includes things like `std::format` improvements, superior string splitting and other fixes and clarifications.

There is uncertainty in how to implement Allowing Virtual Function Calls in Constant Expressions. The way it is implemented has ABI implications, and so Microsoft implemented two additional switches:

- `/experimental:constevalVfuncVtable`, and
- `/experimental:constevalVfuncNoVtable`.

Once they decide which way is the best, it will be made available under both `/std:c++latest` and `/std:c++20`.

Coroutines have been supported in MSVC since C++14. (*I'm sorry, what?*) Given the fact the C++20 is the first official standard to support them, it becomes a bit awkward. So, there is now a new switch `/await:strict` which removes coroutine features not available in C++20. Luckily, it is implied by `/std:c++20`.

And now we come to the most interesting surprise of them all. The new C++20 attribute `[[no_unique_address]]` breaks ABI by changing class layout. Combined with the fact that MSVC ignores unknown attributes, this results in ABI incompatibility between the same code built in C++17 and C++20 modes, and also when linking with libraries built using older toolsets in the supposedly binary-compatible range. So Microsoft decided to just *not implement* this attribute to keep backward binary compatibility until the next ABI-breaking release, whenever that happens (as you may remember, it won't happen in the next version of Visual Studio yet, as VS2022/msvc143 keeps binary compatibility). For those lucky

developers who don't care about older versions of MSVC, there is now a Microsoft-specific attribute `[[msvc::no_unique_address]]` which enables the optimization in *all* language modes, C++14 to C++20. They advise to use a macro that guards this attribute with a MSVC version check. More macros, yay!

There is a thread on Reddit with some interesting replies from STL himself.

Safer Usage Of C++

Google released a long document describing their internal C++ safe programming techniques, as applicable to Chromium.

> Chrome Security has been asked to consider what it would take to make C++ less dangerous. This document outlines various mechanisms we could use to make it significantly easier to use C++ safely. Some are radical, and adopting them (especially adopting many of them) may result in code that looks quite different from what C++ programmers expect. <...> Most of the proposed mechanisms are new usage patterns, libraries, and classes, but some call for the use of compiler-specific flags that change the language somewhat. (For example, Chromium already uses `-fno-exceptions`) <...> Some of these mechanisms are already being built in Chromium, with varying degrees of success. <...> Other mechanisms we propose represent significant new directions for C++ and Chromium. <...> The C++ language and culture tend to trade off safety in favor of efficiency, and therefore many of these proposed changes are complex, controversial, and not as robust as similar changes might be in another language.

They define a couple of terms:

- **Spatial safety** is the guarantee that the program will behave in a defined and safe way if it accesses memory outside valid bounds. Examples include array bounds, struct and union field access, and iterator access.
- **Temporal safety** is the guarantee that the program will behave in a defined and safe way if it accesses memory when that memory is not valid at the time of the access. Examples include use after free (UAF), which is responsible for 48% of security bugs; double-free, use before initialization, and use after move (UAM).

The document names Undefined Behavior (UB) as the source of many problems in C++.

The authors propose a number of measures to make C++ programming safer in Chromium.

- Remove/reduce raw pointers:
  - ban the direct use of raw pointers, `new` and `delete`
  - MiraclePtr aims to make a smart pointer type that eliminates UAF while not impacting the performance too much.
  - Oilpan is Blink render engine's garbage collector
- Annotate lifetimes using Clang's `[[clang::lifetimebound]]` attribute – this has too many limitations and increases visual noise in the code.
- Implement automatic memory management like reference counting (see Objective-C and Swift) or full GC
- Implement ownership analysis – enforce at run-time that there is a single 'owner' of any object, which can only be changed via `std::move`, like 'borrows' in Rust. This solution is a poor fit for C++.
- Use `-Wdangling-gsl` – Google has had good results with this switch
- Define all Standard Library behaviours (where possible) by adding a compile switch for 'hardened mode', like in Abseil.
- Define undefined iterator behaviours by using a checked iterator facility
- Define integer semantics:
  - Use a wrapper numerics library
  - Use a compiler option to make signed integer overflow defined (wrap around).
  - Use UB sanitizer
  - Problem: assuming overflow behavior is a significant change semantics, and if developers come to rely on the newly defined behaviour, a compiler switch change would make all that code defective.
- Set pointers to NULL after free. Peter Somerlad pointed out on Twitter that these would likely be optimized out.
- Define null pointer dereferences.

Clang has the switch:

`-fno-delete-null-pointer-checks`

which results in not optimizing out null pointer dereferences and crashing instead of continuing.

- Require coding patterns to reduce lifetime errors:
  - Use `variant` instead of `enums` – Google has `absl::variant`
  - Ban `std::unique_ptr::get`, use shared pointers - this is the opposite of the current guideline to prefer `std::unique_ptr` to `std::shared_ptr` (*I don't see it this way – if you need to share, use a shared pointer and not the raw pointer extracted from a unique pointer*)
- Initialize all memory - can be expensive at run time
- Remove primitive arrays

- Remove mutable shared state – one of the solutions is a borrow checker in the compiler, like one in Rust.
- Check type casts – type confusion accounts for 7% of Chromium high-severity security bugs
- Prevent use after move using a Clang plugin
- Memory tagging: Depending on the specific mechanism, pointers to and/or regions of memory are 'tagged', and if code tries to load or store memory without using the right tag, the program faults.
- Control flow integrity enforcement – Intel CET technology is enabled on Windows and Apple ARM-based systems.

Reddit says:

> Wow, it sounds as if Google is considering moving to C++11! <...> For the rest it reads like a total disaster, with a major C++ player apparently intending to entirely go their own way, forking a compiler and the entire language, and doing their own thing from here on out...

And another one:

> <...> it kinda looks like they're trying to mimic Rust's safety with compiler extensions and static analysis to C++

And then:

> From what I hear, the preferred solution would've been to rewrite Chromium in Rust, but it required too much engineering efforts, so this is the alternative.

I vividly remember the kind of alien code we used to write back at Symbian to prevent various memory problems and bugs – a large part of every function was memory management. It was an entire custom dialect of C++, and Google seems to be going in the same direction. Good luck with that.

Be Wise, Sanitize

This article by Marin Peko is an introduction to sanitizers: Address Sanitizer (ASan), Memory Santitizer (MSan), and Undefined Behavior Sanitizer (UBSan). Unfortunately, it omits Thread Sanitizer (TSan). Still, a useful summary of what the sanitizers do and how they work, with a shout-out to Valgrind, which is much slower than any of these.

By the way, regarding false positives:

- ASan doesn't produce any – errors it finds are actual bugs
- MSan can produce them if you don't rebuild all your libraries with MSan
- UBSan can sometimes produce them

Reddit thread has some useful comments, including this tip on how to prevent ASan from aborting the program on the first error:

- In addition to the usual switch `-fsanitize=address`, use:

  `-fsanitize-recover=address`

- At run time set the following environment variable:

  `ASAN_OPTIONS=halt_on_error=0`

If you are not using sanitizers in your C++ projects yet, you really should start.

Quote

Meir Lehman:

> An evolving system increases its complexity unless work is done to reduce it.