C++ Club UK Meeting 127

Gleb Dolgich

2021-04-22

Bazel Build

Bazel Build is Google's cross-platform build system for C++ and other languages.

> Bazel only rebuilds what is necessary. With advanced local and distributed caching, optimized dependency analysis and parallel execution, you get fast and incremental builds.

Overview of v4

Visual Studio 2022 Announced

Microsoft announced the next version of Visual Studio, 2022, on their blog. The first public preview wil be released this summer. The main highlights are:

- Visual Studio is now 64-bit! Reddit seems excited about this.
- You can still build 32-bit apps.
- Refreshed user interface.
- A new monospaced font: Cascadia Code which you can try today.
- Support for new C++20 features.
- Backward binary-compatible runtime! Again, some redditors think that binary compatibility is holding back progress. STL replies:

  > We used to break ABI every major release (2008-2010-2012-2013-2015 were all ABI breaks). It was awesome for development, I was able to fix so many bugs. Lots of customers had trouble keeping up, though, which led to them sticking to really old VS releases like VS 2010 for years and years, helping nobody. (I think that customers should be able to rebuild the world on demand, but that is not the case for many of them.) Keeping ABI compatibility in the 2015-2017-2019-2022 release series makes development harder, but has allowed customers to continuously upgrade, which is an improvement.
  >
  > I hope that when we finally have time to do vNext, we can establish customer expectations for long but not infinite periods of ABI stability, followed by periodic migrations, as we'll always be learn-

> ing better implementation techniques and will need ABI breaks to establish a solid foundation for the future.

Other features include:

- Integrated support for Linux, Windows Subsystem for Linux (WSL), and CMake.
- AI-enhanced IntelliSense.
- Support for Git and GitHub.

All C++20 core language features with examples

Oleksandr Koval wrote an article describing all C++20 features, with examples.

> I wanted to learn about new C++20 language features and to have a brief summary for all of them on a single page. So, I decided to read all proposals and create this "cheat sheet" that explains and demonstrates each feature.

It looks like an excellent overview, a brief summary of what's new in C++20, and the examples seem like a very good idea. I'm definitely adding this at the top of my reading list (or stack, even).

The Reddit thread thread has a few useful links, like a similar list of features in C++17 with 'Tony Tables', which are short before/after code snippets 'invented' by Tony Van Eerd (hence the name).

C++20 Range Adaptors and Range Factories

Barry Revzin writes:

> Ranges in C++20 introduce <...> a bunch of range adaptors (basically, algorithms that take one or more ranges and return a new "adapted" range) and range factories (algorithms that return a range but without a range as input). All of these algorithms have several important properties, and I've found that it's a bit difficult to actually determine what those properties are just by looking at the standard, so I'm hoping to make this post serve as that reference.

He proceeds with listing range types and their properties, which include *range reference type* (returned by `operator *`) and *traversal category* (input, forward, bidirectional, random access, and contiguous). He also explains the following range types:

- **Common** - when `begin(r)` and `end(r)` return the same type - starting with C++20 the sentinel type can be different;
- **Sized** - when the range has O(1) member `size()`;
- **Const-iterable** - can it be iterated with a `const` object of the range type (not the case for all C++20 ranges);

- **Borrowed** - guarantees that an iterator into a destroyed range does not dangle.

Barry Revzin then documents range factories: `empty_view`, `single`, `iota` and `istream_view`, and range adaptors: `filter`, `transform`, `take`, `take_while`, `drop`, `drop_while`, `join`, `split`, `common`, `reverse`, `elements`, `keys`, and `values`. Each description has a code snippet illustrating its usage.

Things I should know before starting a C++ developer job

A redditor writes:

> I'm a third year undergrad student and I just landed a position at a company as a cpp dev. I have been using cpp for a while for academic and personal projects, but I am wondering if there are certain things I should know about the industry standard in programming that I may be unprepared for? Any pointers you would like to give to someone starting out?

Some advice from the thread:

- Write self-documenting code. Too many C++ programmers are scared to type more than one-letter variables.
- Self-documenting code doesn't exist. Comment your code.
- Learn to properly touch type (*I can't properly touch-type, I admit, as do many other commenters.*)
- Write tests, but don't create a Google Mock straightjacket.
- Identify the people you should listen to.
- White space is free, this is not the 70s, use it.
- Keep the coding style of the file you're in.
- Don't C++ just for the sake of it. Scripting is your friend in many tooling areas. (*An interesting use of C++ as a verb, must adopt.*)
- Read *Effective C++* by Scott Meyers.
- Always be looking at ways to speed up the build.
- Use clang-format and clang-tidy.
- Use sanitizers.
- Develop relationships with both developers and business domain experts at work.
- Study up on templates, learn the Standard Library, move semantics, RVO, shared pointers.
- If you use `const` everywhere it makes code a lot easier to think about.
- Build small learning projects to reinforce your knowledge.
- There will never be a time in your career where you are "done learning". In this field, you are forever a student.

Writing Interfaces: Stay true in booleans

Radek Vít writes:

> When faced with passing multiple options as arguments to functions, C++ programmers will use `enum class` without much hesitation. Since C++11, it feels like we have moved away from magic numbers and magic string values, and in most cases, it's for the best. There seems to be one exception: when we only have two options to choose from. Here, I will try to convince you to stop overusing bools that don't convey true/false or yes/no.

He then comes up with the following advice:

> If you read out the function call with its arguments, bool arguments and return values must be able to be substituted with true/false or yes/no.

Good:

```
void set_done(bool done);
set_done(true);
```

Bad:

```
void show_window(bool showInForeground);
show_window(false); // ???
```

The author suggests using enums (or enum classes) instead of booleans in cases like this. From my own experience I can add that combining boolean parameters with default values is a recipe for disaster due to implicit conversion of erroneously placed arguments to booleans.

```
enum class WindowOpenMode {
    Foreground,
    Background
};
void show_window(WindowOpenMode mode);
```

The Reddit thread suggests more alternatives:

- bit masks
- builder pattern

```
Beverage
    MakeTea(TeaCup{TeaType::EarlGrey}.withMilk().leaveTeabagIn());
```

- using a struct for multiple flags (the call site looks similar to named parameters):

```
struct tea_input {
    TeaType type;
```

```
    bool milk;
    bool lemon;
    bool sugar;
    bool leave_bag;
};
Beverage MakeTea(const tea_input&);
auto beverage = MakeTea({
    .type = TeaType::EarlGrey,
    .milk = true,
    .lemon = false,
    .sugar = false,
    .leave_bag = true,
});
```

"Proper" named arguments could also help here. There is a proposal P0671R2 by Axel Naumann from CERN to introduce named arguments that are ignored by the compiler:

```
double Gauss(double x, double mean, double width, double
    height);
Gauss(0.1, mean: 0., width: 2., height: 1.);
```

People tell me this syntax resembles Algol 60. To me it looks like Swift.

Metalang99: Full-blown preprocessor metaprogramming for pure C

"What C needs is more macros", said nobody ever, except this guy.

> Metalang99 is a functional language aimed at full-blown C99 preprocessor metaprogramming. It features a wide range of concepts, including algebraic data types, control flow operators, collections, recursion, and auto-currying – to develop both small and complex metaprograms painlessly.

The most amusing word in this quote is, of course, *painlessly*.

- Docs
- GitHub

But wait, it's not all. There is also Datatype99 implemented using the above library. Quoting the author:

> Safe, intuitive sum types with exhaustive pattern matching & compile-time introspection facilities.

Datatype99 requires C99/C++11 and *a lot of macros*.

This is preprocessor macro wizardry Level 80. I hope I never see it used in production.

Modern C++ “result” type based on Swift / Rust

A redditor writes:

> Hey everyone, I created a C++ “result” monad type with functionalities and behaviours much like Swift or Rust’s equivalent `Result` type.

- Header-only, C++11, MIT license.
- Zero-overhead, supports `constexpr`, optimizes very well
- GitHub
- Tutorial

Initially the library was based on the `std::expected` proposal P0323 but it grew over time.

The main differences from `std::expected` are:

- Support for monadic operations
- Support for reference `result` types
- Marked `[[nodiscard]]` to make sure the result is not ignored
- Better naming: with `std::expected` you have to expect unexpected, and `result` is less confusing

See also

- absl::StatusOr
  - not generic over error type
  - doesn’t have monadic operations
- Boost.Outcome
  - More complex, has more customization points
  - Detailed differences from `std::expected`
  - Why Outcome doesn’t duplicate `std::expected` design

Dean Roddey writes:

> The vast majority of code doesn’t care what failed, it wants to just clean and pass the error upstream with minimal hassles. Exceptions plus RAII does that perfectly. Manual error handling is never as clean, no matter how much syntactical sugar you pour on it.
>
> In actual fact, very, very little code in a system should end up actually handling errors in any specific way, because generally only the highest level code understands the context and consequences of a failure. And exceptions also work very well for that.
>
> As I play around with Rust, I spend an inordinate amount of time dealing with error return related silliness in code that doesn’t care in the slightest what happened.

Judging by the reply thread, people disagreeing with this usually come from embedded background, where exceptions are often prohibited.

Boost version 1.76.0 released

Release page has all the details, and that's all I shall say about that. I think Boost is a great set of libraries, and you should use it, but be careful as it can be quite a heavyweight dependency.

Quote

Manisha tweeted:

> Changing random stuff until your program works is bad coding practice, but if you do it fast enough it's Machine Learning.