# C++ Club UK Meeting 130

Gleb Dolgich

2021-06-03

## Contents

## Pure Virtual C++ Conference by Microsoft

Microsoft has published videos of the conference talks. Of a particular interest is the talk by Gabriel Dos Reis, C++ Modules: Year 2021.

### C++ Modules: Year 2021, by Gabriel Dos Reis

After a short introduction to modules, Gabriel gives an update on the current status of modules in different compilers:

- MSVC: modules are production-ready.
- GCC11: still in experimental mode.
- Clang has a feature branch for module development, which is still in experimental mode - see https://github.com/iains/clang-cxx20-modules.

Gabriel expects GCC and Clang modules to reach production-level quality by the end of 2021 or early 2022.

Build tools that support modules: production quality - MSBuild, experimental support available in CMake, Build2, and Meson.

Gabriel mentions the following articles on modules in MSVC:

- A Tour of C++ Modules in Visual Studio
- Walkthrough: Build and import header units in Microsoft Visual C++
- Walkthrough: Import STL libraries as header units

He also lists the following talks on modules and how to use them:

- GDR: Programming with C++ Modules: Guide for the working programmer (Video)

- Daniela Engert: Visibility, Reachability, Linkage - The Three Spices of C++ Modules (Video)
- Boris Kolpackov: Practical C++ Modules (PDF)
- GDR: Programming in the Large With C++20 - Meeting C++ 2020 Keynote (Video) – I found this one to be very detail-heavy on module interface file (.ifc) structure and implementation, which could be useful for tool developers, but perhaps not so much for the ordinary programmer who wants to use modules.

As an aside, Gabriel used a concept to prevent implicit conversion in a function:

```
template <std::same_as<uint32_t> I>
void foo(I index) {...}
```

This means the parameter `index` must be exactly of type `uint32_t` and not some other type implicitly convertible to `uint32_t`.

Gabriel talks about the new opportunities for tooling, enabled by the new semantic representation of the code provided by module interface build artefacts. He mentions his disappointment at the lack of common representation of the module interface across compilers, and his hope for an eventual convergence onto a common format.

The binary representation (.ifc) that MSVC uses for modules is based on the open-source Internal Program Representation (IPR), developed by GDR and Bjarne Stroustrup 15 years ago.

One thing that came up in the Q&A session after the Meeting C++ 2020 keynote was the fact that .ifc files are compiler setting-specific, but I don't remember any mentions of compiler flags being stored in .ifc – I'm curious if compilers and tools will be able to check if an .ifc file was built with compatible compiler flags.

Modules present an opportunity to replace the commonly used interface with other languages, `extern "C"`, with something better. This is of a particular interest to me as I use SWIG to interface with Java. SWIG is a *'Simplified Wrapper Interface Generator'*, and if you have something called *'Simple'* it is never that (case in point: SOAP). I would really like to replace SWIG with something modern.

Another interesting thing said in the Q&A session was the future possibility of embedding .ifc in DLLs. This would provide tools with run-time introspection capability, much like .NET languages have today.

At the end of the talk Gabriel asks all the interested developers to go and try using modules with all the compilers and build tools, and provide feedback to WG21 and tool developers.

**Visibility, Reachability, Linkage - The Three Spices of C++ Modules, by Daniela Engert**

Daniela has been building software since the late 1970s. She says:

> Folks, I just think we are doing this wrong now. We follow the Book of John - I mean, John Lakos's book about layering software and designing large systems.

She explains the drawbacks of the current header-based compilation model (MACROS! ODR violations!) and says that modules allow us to shift the library 'protection' boundary from the current position between libbary interface and implementation, to the more natural position between library as a whole and its consuming translation unit.

> This is the way.

(A nice call out to The Mandalorian there.)

Daniela goes through C++ name lookup rules and explains how modules allow better control of name visibility across translation units by using the `export` qualifier. She also explains the various types of linkage and how modules affect the picture. A good detailed talk worth your attention.

## Modules in managed C++

After the Meeting C++ 2020 keynote someone asked Gaby if modules were supported in managed C++ (or C++/CLI). He said no work has been done in that area.

Module support in MSVC requires `/std:c++latest` switch, which requires MSVC standard-compliant mode (`/permissive-`). This switch enables two-phase lookup, which is not supported with managed C++.

## I tried using modules in a Hello World app in Visual Studio 2019

I created a new Win32 console project that included `iostream` and wrote `Hello World!` to `std::cout`:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

Then I went to **Project Settings > C/C++ > Language** and set the following options:

- **C++ Language Standard**: Preview - Features from the latest C++ Working Draft (/std:c++latest)

- **Enable Experimental C++ Standard Library Modules**: Yes (/experimental:module)

Note that this requires that you select the Standard Library Modules component during Visual Studio installation.

After that I replaced the `#include` with `import`:

```
import std.core;

int main()
{
    std::cout << "Hello World!\n";
}
```

When I did all that in the currently available VS2019 v16.9.2, the program built and ran successfully. I didn't have to manually tweak the system module paths like in earlier versions. However, IntelliSense didn't seem to know about modules - the `import` keyword was highlighted red, and `std::cout` appeared as an unknown symbol.

When I did the above in the latest VS2019 v16.10.0 Preview 4, everything worked, including IntelliSense, which didn't flag any errors.

## VS2019 16.10.0 Released

Just as I was writing this, Microsoft released the next version of VS2019, 16.10.0. The main news is that all C++20 features are now available under the `/std:c++latest` switch. It's the first compiler to be C++20 compliant.

A redditor writes:

> I love that EVERY SINGLE TIME they put in Release Notes, that they repaired IntelliSense for C++20 Modules. Then I write a single Hello World program and Intellisense fails.

In my experience with a Hello World program IntelliSense in the new version seems to work, as opposed to the previous release, but I haven't used modules properly yet.

### VS2019 16.11 Preview 1

The first preview of the next version 16.11 is now available. Visit the announcement page to see lots of pretty screenshots. Lots of improvements in Git support, but I'm probably not going to use them as I prefer Git command line. Most Git tools hide the crucial details from you, and then you end up with some weird history or a tragic merge you can't get out of.

I gave it a go, and IntelliSense works well with modules. I also tried importing `<iostream>` as a header unit instead of `std.core` into my Hello World program, and IntelliSense didn't flag any errors. However, the build failed because the compiler couldn't find the header unit.

## MSVC Standards conformance mode compatibility

Starting with VS2017 (MSVC compiler v14.1) it is possible to opt in to the standards conformance mode with the compiler switch `/permissive-`.

> This option disables permissive behaviors, and sets the `/Zc` compiler options for strict conformance. In the IDE, this option also makes the IntelliSense engine underline non-conforming code.

This is the default mode for new projects created with Visual Studio 2017 version 15.5 and later.

> Starting in Visual Studio 2019 version 16.8, the `/std:c++latest` option implicitly sets the `/permissive-` option. It's required for C++20 Modules support.

The conformant mode requires Windows SDK v10.0.16299.0 or later (Windows Fall Creators SDK). Earlier SDK versions do not compile in conformant mode.

The two-phase lookup is not supported in C++/CLI. OpenMP is not compatible with two-phase lookup until VS2019 v16.5.

To disable two-phase lookup and leave other conformant behaviour enabled, use the switch `/Zc:twoPhase-`.

## WG21 May 2021 mailing

The May 2021 mailing is out. Here are some papers that caught my attention.

### Monadic operations for `std::optional`

P0798R6

The author, Sy Brand of Microsoft, writes:

> `std::optional` is a very important vocabulary type in C++17 and up. Some uses of it can be very verbose and would benefit from operations which allow functional composition. I propose adding `map`, `and_then`, and `or_else` member functions to `std::optional` to support this monadic style of programming.

Imagine a set of image processing functions that return an image. To call them in sequence, you would use a series of nested function calls:

```
image get_cute_cat(const image& img)
{
    return add_rainbow(
        make_smaller(
            make_eyes_sparkle(
                add_bow_tie(crop_to_cat(img)))));
}
```

Notice how you have to read the calls from the innermost function call outwards, which isn't very natural.

If we want these functions to return an optional image or an error code, we can use `std::optional<image>` for that. But now we have to check every call's return value so that we don't call the subsequent functions if a call returned an error. This proposal makes it possible to chain such function calls and handle failures transparently for the entire call chain:

```
std::optional<image> get_cute_cat(const image& img)
{
    return crop_to_cat(img)
        .and_then(add_bow_tie)
        .and_then(make_eyes_sparkle)
        .map(make_smaller)
        .map(add_rainbow);
}
```

This is how optional data type works in other languages – `Optional` (Swift, Java), `Maybe` (Haskell, Agda, Idris), `Option` (Rust, Scala). According to Sy Brand, C++ is the only language that has an optional type without monadic API.

### Mark all library static cast wrappers as `[[nodiscard]]`

P2351R0

This paper by Hana Dusìkova is a great example of a small targeted proposal that makes sense and fits on a single page.

> This paper proposes adding the attribute `[[nodiscard]]` to all library based cast function templates (which only wrap language casts) as they are meaningless without accessing the resulting value.

These templates are: `to_integer`, `forward`, `move`, `move_if_noexcept`, `as_const`, `to_underlying`, `identity`, and `bit_cast`. Hana notes that Microsoft STL already marks these as `[[nodiscard]]`.

### Pattern Matching with Exception Handling

P2381R0

The *n+1* monster strikes again! This time it's about using the proposed pattern matching feature (see P1371R3) with exceptions. The author, Jarrad J. Waterloo, proposes to allow the `inspect` statement to catch exceptions without having to use `try/catch`. This way, if a function returns `A` and throws `B` and `C`, the cases of the `inspect` statement for the function call would be able to match `A`, `B`, and `C` in a uniform way.

Reddit points out many flaws in the proposal. What happens if the inspected function returns `A` or `variant<A,B>` and also throws `A`? Compared with exception handling, which you can and often should avoid doing for a single function call, the proposal adds quite a lot of error checking machinery at call site.

Some corner cases could also lead to hard-to-find bugs. As the redditor gracicot notes, imagine a function returning `variant<A,B,C>` being called using the new `inspect` statement that has the 3 cases. If the function signature changes to return `variant<A,B>` the `inspect` statement still compiles, but now the case `C` is silently a `catch`. If you had an enclosing `catch(C)` at a higher level, it won't catch it anymore.

The paper seems to miss quite a few important use cases and it's not clear if the proposed feature actually improves anything, while introducing many potential breaks and subtle hard-to-find bugs.

## C++ has a Wikipedia page dedicated to criticisms of it

Wikipedia

The first comment on the Reddit thread summarizes it nicely with a Bjarne Stroustrup quote:

> There are only two kinds of languages: the ones people complain about and the ones nobody uses.

The article has sections on slow compile times, `<iostream>`, iterators, uniform initialization syntax, exceptions, encoding of string literals (of all things), and code bloat (for 'some old implementations of C++'). It also links to some articles by authors that seem to have an irrational hate for C++. I mean, it's just a programming language, you don't *have* to use it.

## Semantics of unsigned integers

A redditor asks, what should be the semantics of unsigned integers. As it often happens, to get the right answer someone has to post a wrong answer, and the redditor Full-Spectral was happy to oblige. They wrote:

> I'm one of those that doesn't agree with the always use signed types thing. If you don't understand the magnitude of the values involved, using signed types isn't going to magically make everything better. I believe in modelling the things you are operating on, and if that can never be negative, then I don't see how using signed values is better unless you always check the result.

Tony Van Eerd replied:

> People expect numbers to act like numbers. <...> Neither signed nor unsigned really work like numbers when you are near the boundaries, but unsigned puts a boundary at 0, which may very well be the most common number in all of programming. <...> So avoid unsigned numbers for numbers. Use signed. The committee apologises profusely for making `size_t` be unsigned.

Another redditor wrote:

> These are some valid use cases of modulo behaviour of unsigned arithmetic:
>
> - hashes
> - random numbers
> - implementing multiprecision types
> - crypto
> - emulation of hardware

And bit operations, of course.

Another redditor reminded what The C++ Core Guidelines say about this in the section ES.106: *Don't try to avoid negative values by using unsigned*:

> The C++ Core Guidelines also have the "unsigned for bitwise, signed for arithmetic" rule.

And of course someone mentioned Rust:

> Rust does a better job of divorcing signedness from overflow behaviour and what's undefined behaviour, along with avoiding the implicit integer casts and promotions that make these subtleties problematic.

## Twitter

Sara Drasner @sarah_edo describes the ideal development workflow:

> Make it work
> Make it right
> Make it fast
> Make it work again because you broke it making it fast