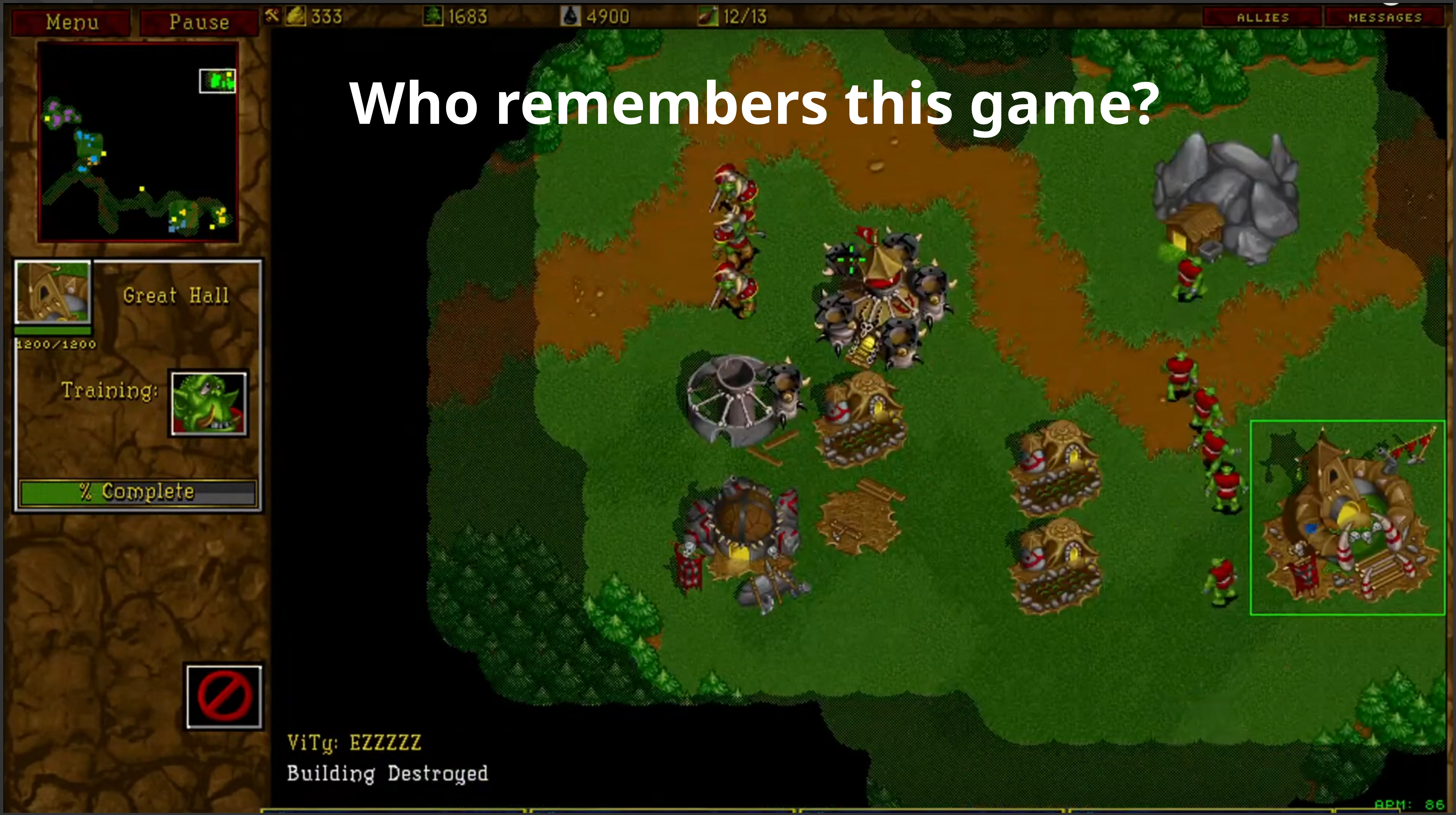
Modularized C++

Alexander Christensen

COMPUTER GRAPHICS ENGINEER, C++

Who remembers this game?



ViTy: EZZZZZ
Building Destroyed

Briefly on my background

- Former “Graphics Engineer” at Alexandra Instituttet
- Studied CS to learn about how games work
- Became interested in modules ~4 years ago, for my own game engine
- Presently learning tech-art stuff in Unreal Engine
- Excited and honored to be here today to talk a subject of great interest

The evolution of C++

- Bjarne Stroustrup began work in "C with classes" 1979.
- Inspired by Simula
- Changed to "C++" 1982, with virtual functions, operator overload, references, …
- And **MORE!**

"C makes it easy to shoot yourselves in the foot; C++ makes it harder, but when you do it blows your whole leg off."
-- Bjarne Stroustrup

Where are we now

- C++ now has so many features that people actively complain about it.
- People who tire of it just move to Rust world, et al.
- Every time someone gets a fancy new idea in the Committee, we are all thinking, "Oh no, not another feature…"

- But why is C++ so complicated? At the *"head"* of the problem...

At the head(-er) of the problem

- We are forced to think about in which files we place stuff, where these files are located, where/how these files are #included. And we invent a lot of hacks around this to save compilation time.
- We still end up compiling the same headers 100s of times in larger projects.
- Precompiled headers can be used, but they are tricky to set up, and need recompiling each time we add a new header. And they easily become >100Mb in size!
- We need stuff like include guards, "static inline", "declspec" for shared libraries, "header-only" for templates, and many other inconveniences..

At the head(-er) of the problem (2)

- Macro name clash, e.g., if two headers #define the same macro, commonly on Windows for min() and max().
- Macro names leak, because *sometimes*™ we forget to #undefine them..
- Circular header inclusion, which can really be a pain to work around.
- No definition of static class member variables in header files < C++17.
- Templates have **reduced** usefulness because we cannot freely place them in source files.
- Macros have no type safety => "Intellisense" tools are difficult to maintain, and they often break.

At the head(-er) of the problem (3)

- Build tools are complicated and unreliable: Difference between internal/external inclusion of a header -> needless complexity.
- Packaging libraries is non-standard, and library maintainers often need to write a dedicated guide, just to explain how to install the library! (where's my button?)
- Community divergence: cmake, premake, xmake, bazel, nuget, ...

- Who enjoys working with C++ build configuration? (please raise a hand :))

Painful experience

- My PoolAllocator<T> needed to be in a header-file.
- Cannot use internal engine log since header can be included from anywhere.
- Could maybe be solved with more macro hacks, but that’s not pretty and could introduce more problems.
- Instances like this got me interested in researching an alternative...

```cpp
/* workaround for core_log_fatal, since this file is header-only,
   and core_logger is not available outside DLL-space! */
YGG_API void _core_log_fatal_wrapper(const char* msg);

// ============================== //
// === CONCEPT IMPLEMENTATION === //
// ============================== //

// An implementation of PoolAllocator which uses C++20 concepts instead
// of SFINAE to perform compile-time substitution.

template<typename T, U32 Count>
concept pointerFits =
        (sizeof(T) >= sizeof(void*)) &&
        std::is_default_constructible_v<T> &&
        Count > 0;
```

```cpp
// Implementation using free pointers
template<class T, U32 Count>
requires pointerFits<T, Count>
class YGG_API PoolAllocator<T,Count>
        : public Allocator
{
public:
        PoolAllocator(const char* description = "PoolAllocator[pointerFits]")
                : Allocator(AllocatorType::pool_allocator, description)
```

```cpp
T* Alloc()
{
        mLock.Lock();

        if (mpFreePointer == nullptr) {
                _core_log_fatal_wrapper(
                        "PoolAllocator[Ptr]::Alloc(): mpFreePointer is nullptr!");
```

Blank slate is impossible

- We live in a world of software, with dependencies > 50 years old. Before C everyone was writing Fortran. C was adopted because Fortran code was easily translated into C, and the C compiler was small and simple. Then came C++ with full support for C, which allowed a *gradual* transition.

- Rust is popular, but it breaks compatibility, and will likely introduce "violent" changes again.

- Too much existing C++ code is still in use. Rewriting it is infeasible.

Modules to the rescue

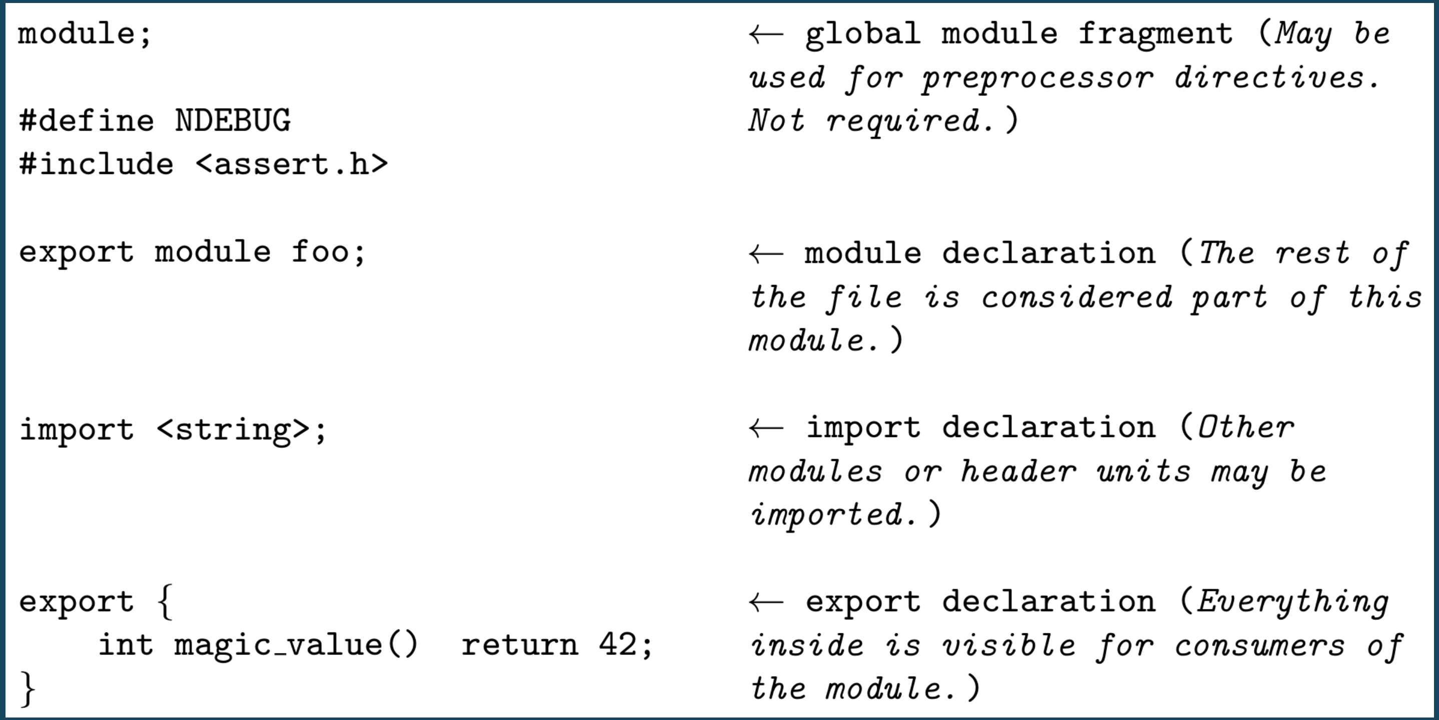
- Proposed by Daveed Vandevoorde 2012; Doug Gregor 2012; 2014 by Gabriel Dos Reis, Mark Hall, Gor Nishanov; and others. After several divergent proposals, was finalized and adopted into C++20. C++23 aims to modularize the standard library.

- Idea is to completely scratch header files from projects, and only have source files. A source file will always expose a named module.

- A source file is parsed *once*, and then serialized into an abstract syntax tree. This means that templates can be arbitrarily instantiated when using them, instead of parsing them again from a header file.

So what is a module?

- A module is an entity, orthogonal to a namespace, exposed by a translation unit. We create it by compiling a source file, which yields two outputs: An object file, and an abstract syntax tree (aka. binary module interface or BMI).



`module;`
 	
`#define NDEBUG`
`#include <assert.h>`
	← global module fragment (*May be used for preprocessor directives. Not required.*)

`export module foo;`	← module declaration (*The rest of the file is considered part of this module.*)
`import <string>;`	← import declaration (*Other modules or header units may be imported.*)

`export {`
`    int magic_value()  return 42;`
`}`
	← export declaration (*Everything inside is visible for consumers of the module.*)

The promise of modules

- That we may rid our projects of header files.
- No include directories, inline functions, header-only libraries.
- No more magic macro customizations for conditional includes and dynamic dependency injection.
- The preprocessor has no understanding of C++ code, so let's use the compiler for that instead!
- Everything (well, mostly everything) will be easier, simpler, and better.
- My estimate: ~50% language complexity reduction.
- Vastly improved build times.
- No loss of expressiveness.
- Arguably, the most important addition to the language since templates.

Let's try it out!

1. Compile a single file
2. Use a module partition / submodule
3. Templated logging library
4. Compile-time disabled code
5. Source location

Example: Templated logging library

Let's create a small logging library. Use std::string for logging messages.

- 6 logging levels: trace, info, debug, warn, error, fatal.
- Each level has its own output color

- Parameters are handled by exported variadic template functions
- Compile all the template logic inside module interface unit.

- Creating a new, unseen log call (ie. new template instantiation) can still use the same compiled module unit! (not possible with header files)

Templated logging library - challenges

- Manual log message formatting since std::format doesn't compile in a module, and {fmt} doesn't compile in GCC with modules enabled. :(
- Create my own custom string_view class, because std::string_view does not work in a module either. (Can compile; but fails at runtime)
- std::string objects that cross the module interface boundaries needs to be manually copied. Maybe this is a vtable error, but definitely inside the compiler.

Templated logging library - challenges

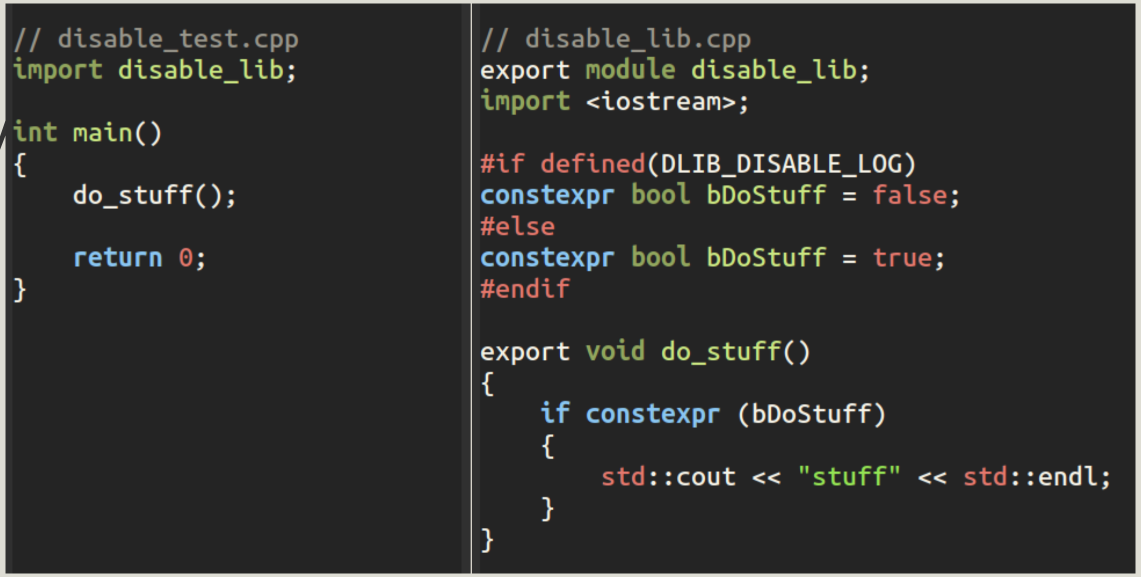
- Variadic template functions don't expand correctly inside a module unit when recursively pruning parameters. (compiler error)

- It works, but feels very fragile. (Hopefully this changes with next GCC installment ?)

- Output a newline "std << '\n", because multiple subsequent calls to "std::endl" inside the module crashes the program (segmentation fault).

Example: Compile-time disabled code

- One of the usages of logging libraries is for development-only and either completely or limited output in production environments.
- Can we obtain this feature, compile-time, with modules, without macros?



```cpp
// disable_test.cpp
import disable_lib;

int main()
{
    do_stuff();

    return 0;
}
```

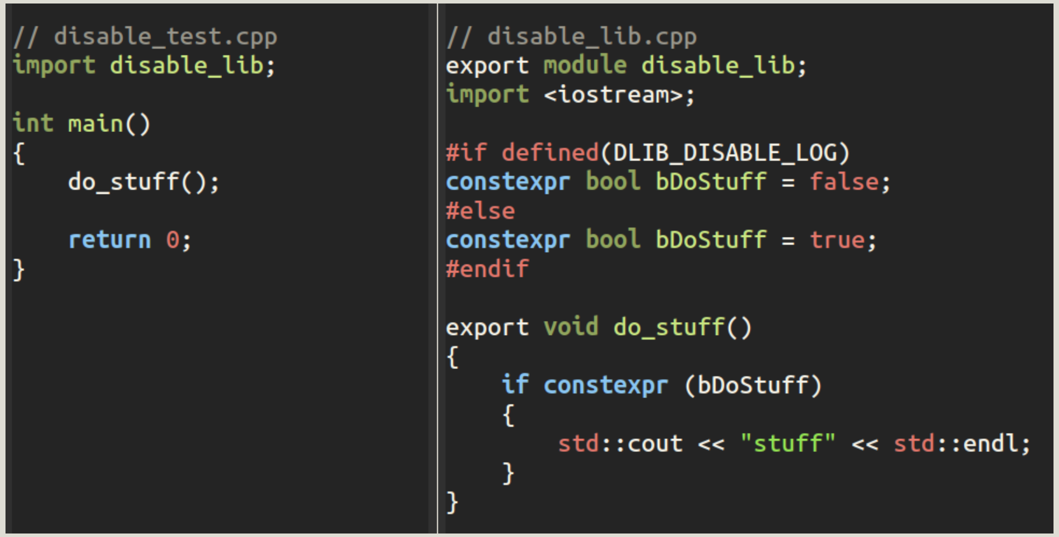
```cpp
// disable_lib.cpp
export module disable_lib;
import <iostream>;

#if defined(DLIB_DISABLE_LOG)
constexpr bool bDoStuff = false;
#else
constexpr bool bDoStuff = true;
#endif

export void do_stuff()
{
    if constexpr (bDoStuff)
    {
        std::cout << "stuff" << std::endl;
    }
}
```

Yes and no.

Compile module interface with "`-DDLIB_DISABLE_LOG`"*,*
Rely on LTO to remove function call.

Inspecting the output binary(ies)

```
$ objdump -d disable_test | wc -l
164
$ objdump -d disable_test_release | wc
-l
135

$ readelf -h disable_test
[…]
Number of section headers: 37
Section header string table index: 36

$ readelf -h disable_test_release
[…]
Number of section headers: 35
Section header string table index: 34
```



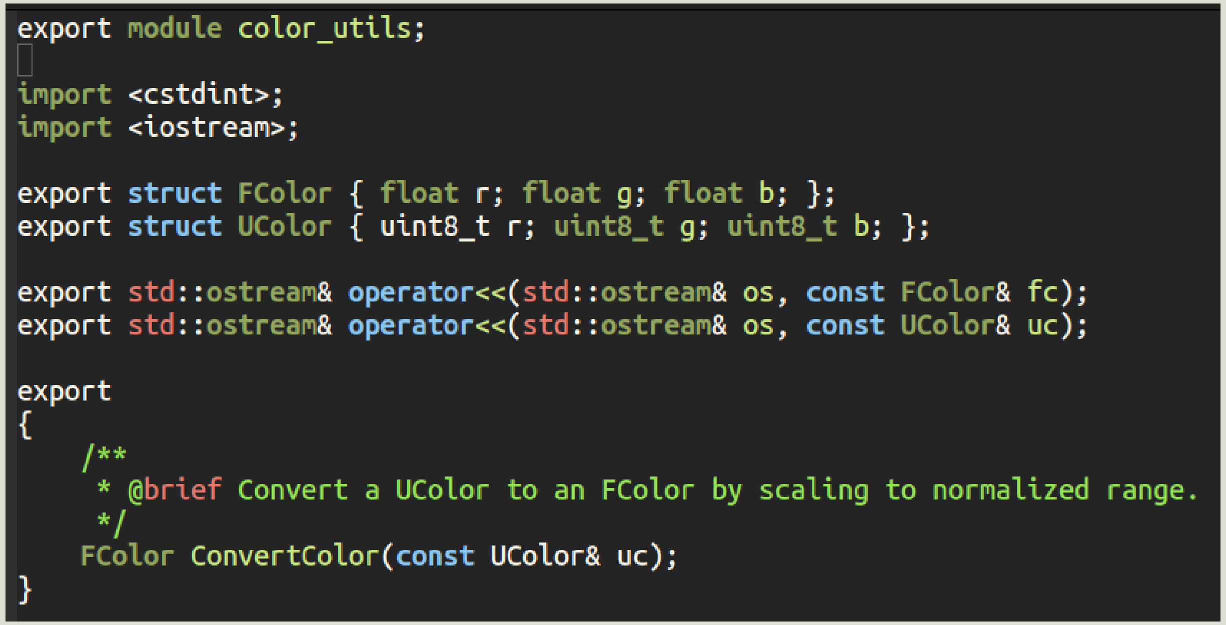
```cpp
// disable_test.cpp
import disable_lib;

int main()
{
    do_stuff();

    return 0;
}
```

```cpp
// disable_lib.cpp
export module disable_lib;
import <iostream>;

#if defined(DLIB_DISABLE_LOG)
constexpr bool bDoStuff = false;
#else
constexpr bool bDoStuff = true;
#endif

export void do_stuff()
{
    if constexpr (bDoStuff)
    {
        std::cout << "stuff" << std::endl;
    }
}
```

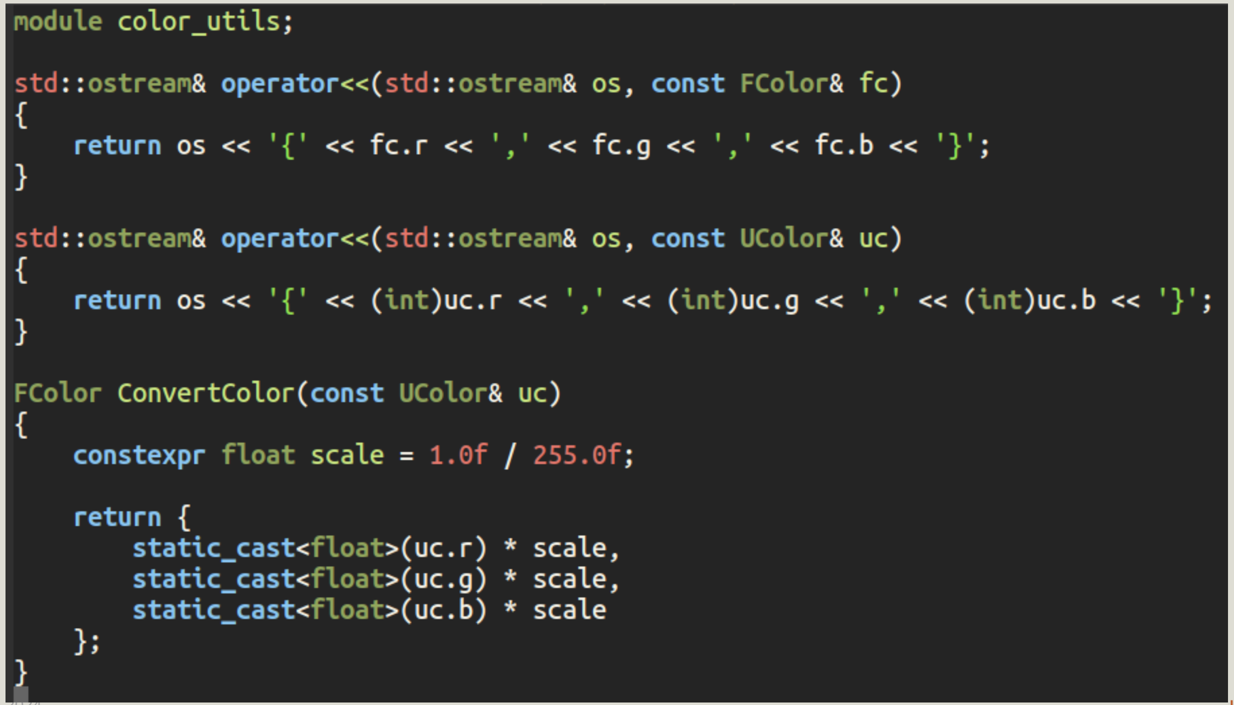
- *No global string data*
- *No reference to basic_ostream*
- *Smaller output binary*

Documentation?

- Previously in header file, accompanying declarations.
- We can use a Module Implementation Unit for definition.
- And the Module Interface Unit for declaration & documentation.



```cpp
export module color_utils;

import <cstdint>;
import <iostream>;

export struct FColor { float r; float g; float b; };
export struct UColor { uint8_t r; uint8_t g; uint8_t b; };

export std::ostream& operator<<(std::ostream& os, const FColor& fc);
export std::ostream& operator<<(std::ostream& os, const UColor& uc);

export
{
    /**
     * @brief Convert a UColor to an FColor by scaling to normalized range.
     */
    FColor ConvertColor(const UColor& uc);
}
```



```cpp
module color_utils;

std::ostream& operator<<(std::ostream& os, const FColor& fc)
{
    return os << '{' << fc.r << ',' << fc.g << ',' << fc.b << '}';
}

std::ostream& operator<<(std::ostream& os, const UColor& uc)
{
    return os << '{' << (int)uc.r << ',' << (int)uc.g << ',' << (int)uc.b << '}';
}

FColor ConvertColor(const UColor& uc)
{
    constexpr float scale = 1.0f / 255.0f;

    return {
        static_cast<float>(uc.r) * scale,
        static_cast<float>(uc.g) * scale,
        static_cast<float>(uc.b) * scale
    };
}
```

Some (other) experiments I have done

- Module dependency parser (https://github.com/alexpanter/cpp_module_parser)
- Create small test programs (https://github.com/alexpanter/modules_testing)
- Trying to modularize my own custom game engine (very hacky around premake build system, can't export stuff from submodules, GCC was extremely unstable, didn’t really work)
- Very few experiments with Clang, none with MSVC

What are the challenges

- Not yet reached consensus on binary format of module partitions, their file extensions, or file structure of module projects. (Microsoft certainly has tried!)
- Tools can only do so much as there is community/committee consensus, so they are very limited and not mature.
- For formal guidelines on project structure, when/when not to use module partitions, how to document (no header files!).
- GCC with modules feels like an alpha-stage indie MMORPG game!

Feedback from an {fmt} developer

- *"I've tried to create a BMI from the {fmt} library just yesterday: msvc worked 2 years ago, clang can do it now, and gcc is failing spectacularly." (April 23rd, 2023)*

Q/A with a modules expert

I was building an experimental parser (fall 2021) for reading a number of module files, which would generate an order in which they should be compiled, because research into modules convinced me this was necessary.

- *Despite common belief, there is no difference to precompiled headers. Both suffer from both the intricacies of the preprocessor and dynamically created content. This is the most condensed summary of our findings in SG15 in the committee.*

These are the problems as I see it:

1. Builds cannot (easily) be parallelized because translation units must be compiled in a particular order
2. Every build run needs a pre-build run where ALL source files are scanned, dependencies extracted, and a DAG generated

- *It all depends on the scenario you're in: static dependencies or dynamic ones (most people underestimate the impact of conditional compilation and computed includes/imports) and the amount of artifact caching you're willing to spend. Think of header files as manually cached interface artefacts! It's one of the many roles that they assume.*
- *To your first point: That's exactly the same as with precompiled headers. If dependencies haven't changed, all the existing intermediate artifacts like PCHs or BMIs can be reused, and you can parallelize to your heart's content.*
- *To your second point: that's the same as it ever was with headers.*

Q/A with a modules expert (continued)

- *Most people aren't aware of the brokenness of their existing build systems that don't take all possible scenarios into account. In the past with mostly static dependencies and cached interfaces (i.e. headers), everything seemed so easy and simple and build tools got away with it. The few exceptions of dynamically created dependencies were handled by manually inserting pseudo nodes into their makefiles. This is (and was) neither scalable nor sustainable. The advent of modules was just putting the spotlight onto these nasty, dark corners of the ecosystems, exposing their weaknesses, deficiencies, and lack of functionality. This is all true *without modules*!*

The main concern is the compilation order, which seems to me like despite the other major advantages of modules, makes C++ worse.
And I don't hear anyone talking about this. Making build tools is already a nightmare, and now it's going to get a lot worse.

- *Nothing has changed by modules. The problem is: people want to keep their broken tools.*

Are these issues mitigated by creating module implementation units?

- *Those are the equivalents of regular source files: they don't expose interfaces - they implement them.*

Do you know of any efforts to address these issues?

- *SG15 is looking into that. There are people trying to address this in their build systems. There are newer build systems designed for *all* scenarios.*[1]

1: [CMake has experimental module support: https://www.kitware.com/import-cmake-c20-modules/]

What does the Committee say

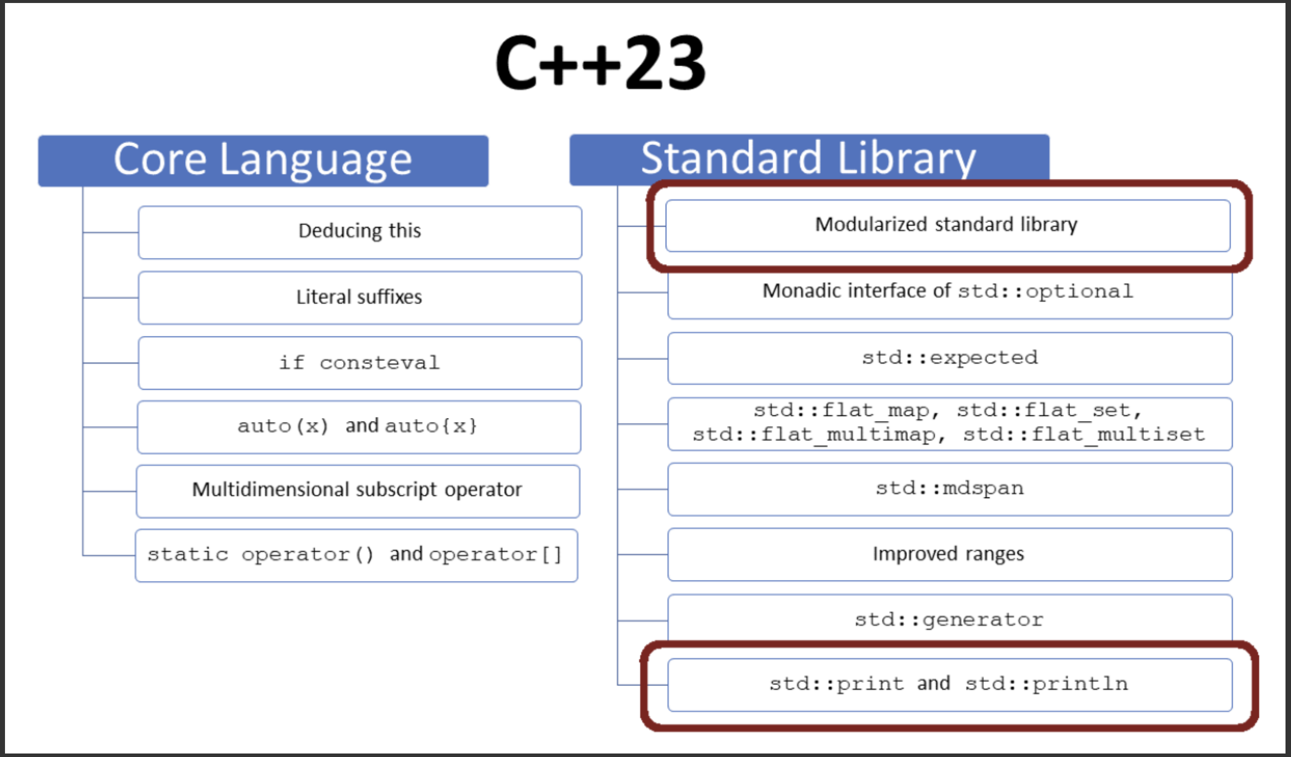
- *"[…] the canonical way to share definitions in C++ is now modules, not textual inclusion of text blobs."*

- *"Yes, as I see GCC still have broken implementation (I checked it today and on a simple example I get ICE) but sooner or later it will be fixed. This is very big change that spread for multiple layers of compiler, is very easy to mess up something on every level."*

- *"This is true that for "production ready" code it is too early to use modules but again it does not change the fact that in the next 1 or 2 years this will change, and after 10 years we could even in new projects not use `#include` at all (or at least only for very specific usages like C headers)."*

C++23 roadmap

- https://www.modernescpp.com/index.php/c23-a-modularized-standard-library-stdprint-and-stdprintln/


C++23
Core Language
Deducing this
Literal suffixes
if consteval
auto(x) and auto{x}
Multidimensional subscript operator
static operator() and operator[]
Standard Library
Modularized standard library
Monadic interface of std::optional
std::expected
std::flat_map, std::flat_set, std::flat_multimap, std::flat_multiset
std::mdspan
Improved ranges
std::generator
std::print and std::println

Recommended talks

- CppCon 2021: *"A (Short) Tour of C++ Modules,* Daniela Engert
- CppNow 2023: “The Challenges of Implementing C++ Header Units”, Daniel Russo
- CppCon 2023: “The Packaging and Binary Redistribution Story”, Luis Caro Campos

Conclusion:

- Very promising
- Certainly the future
- Toolchain not ready
- Waiting for (at least) C++23

Thank you

for your time

ALEXANDRA INSTITUTE