The Hitchhiker's Guide to *FASTER BUILDS*

by Viktor Kirilov

Me, myself and I

- my name is Viktor Kirilov - from Bulgaria
- 7+ years of professional C++ in the games / VFX industries
- worked only on open source for 3 years (2016-2019)
- creator of doctest - the **fastest** C++ testing framework

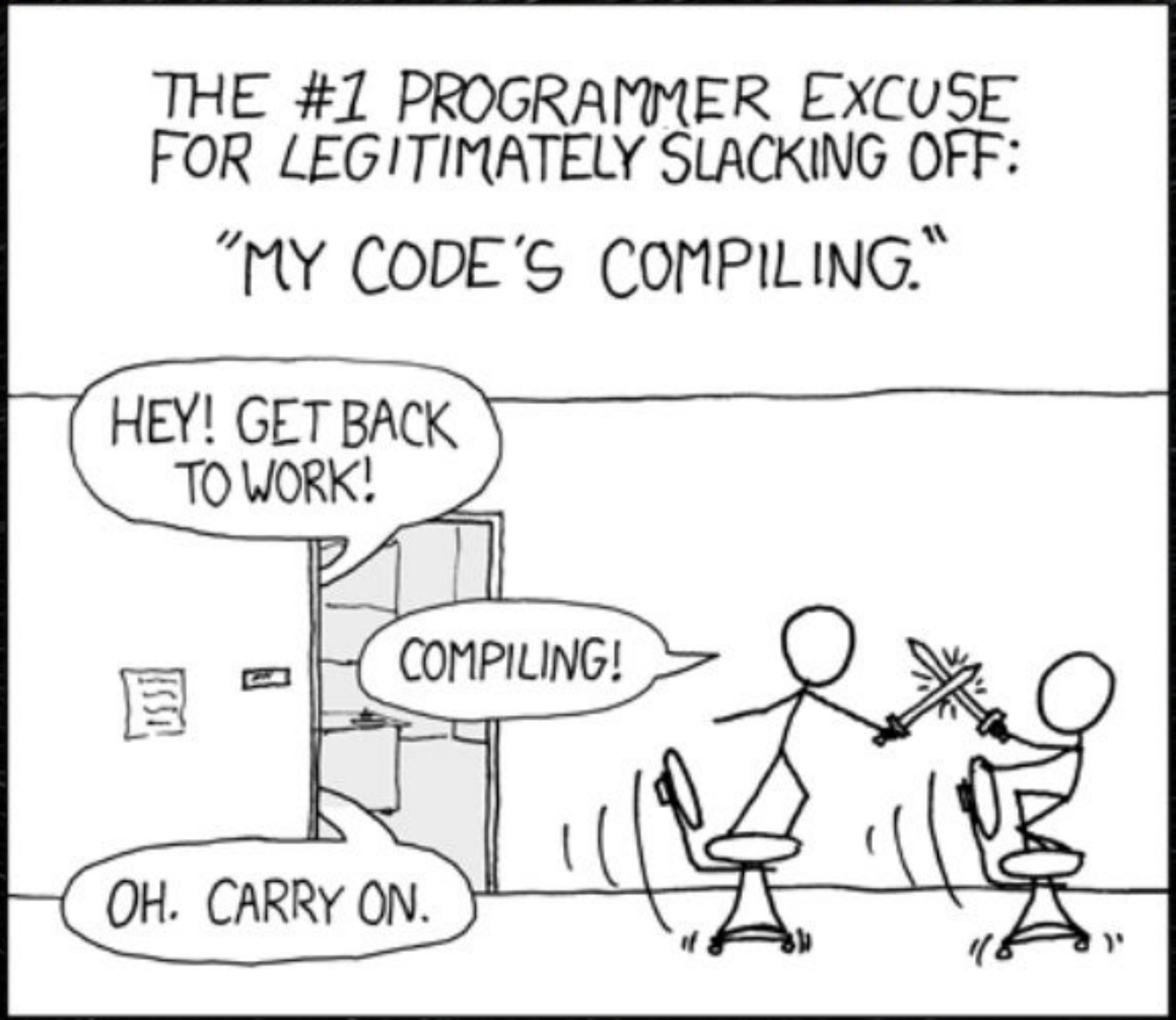
Passionate about

-
-
-
-
- **!!! not wasting time !!!**

C++ is known for...

- ▲ performance
- ▲ expressiveness
- ▲ being multi-paradigm
- ▼ complexity
- ▼ metaprogramming could be much more powerful and simple
- ▼ no standard build system and package management
- ▼ L0oo0oOoNG COMPILE TIMES

Major C++ issue - compile times



This presentation

Slides: https://slides.com/onqtam/faster_builds

Introduction: ***WHAT*** and ***WHY***

templates

binary bloat

forward declarations

modules

toolchains

memes

annotations

HOW

caching

parallel builds

precompiled headers

tooling

build systems

compilers

symbol visibility

distributed builds

hardware

PIMPL

diagnostics

unity builds

includes

linkers

static vs dynamic linking

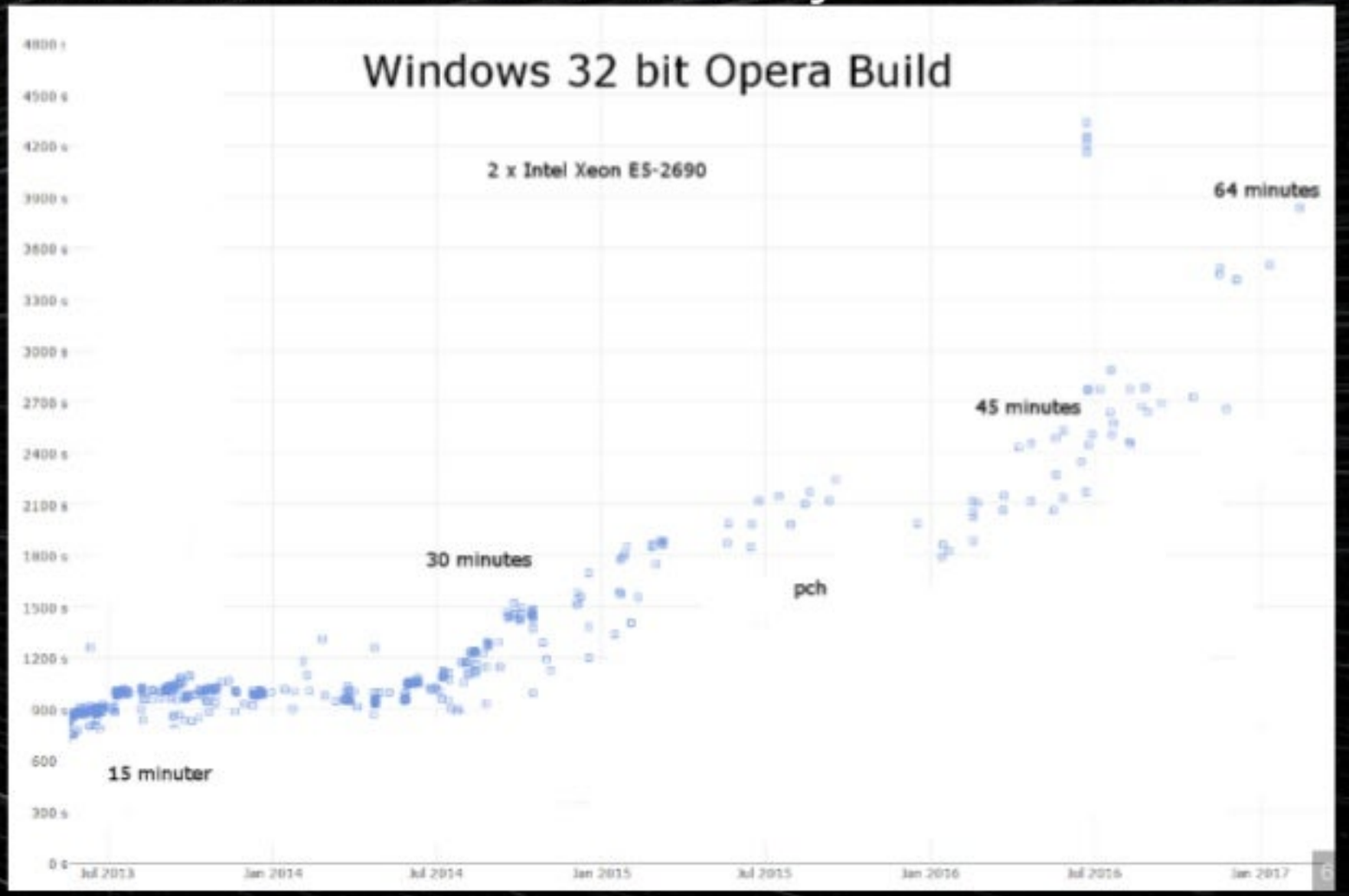
Reality

- some projects take (tens of) hours to compile - WebKit: a big offender
- both minimal and full rebuilds suffer more and more in time
- all projects start small - build times increase inevitably

Opera 32 bit on Windows

in 3 years 6 months:

- 15 minutes to 64+
- + incorporated PCH !!!



A unique problem to C++

Actually the folks at Rust sympathize with us :D

Why reduce build times

- idle time is costly - lets do the math:
  - assuming a 80k $ annual salary
  - 1 hour daily for waiting (out of 8 hours: 12.5%)
  - 10k $ per developer annually
  - value to company > at least x3 the salary
  - 100 developers ==> 3 million $ annually
- the unquantifiable effects of long build times
  - discourage refactoring and experimentation
  - lead to mental context switches (expensive)
- any time spent reducing build times is worthwhile !!!

The steps in building C++ code

- preprocessor: source file => translation unit
  - includes
  - macros, conditional compilation (ifdef)
- compiler: translation unit => object file
  - parsing, tokens, AST
  - optimizations
  - code generation
- linker: object files & libraries => executable
- Absolutely fantastic explanation:
  - https://github.com/green7ea/cpp-compilation

Why C++ compilation takes so long

- backwards compatibility with C: a killer feature ==> adoption!!!
  - inherits the archaic textual include approach
    - originally for sharing just declarations & structs between TUs
- language is complex (ADL, overloads, templates, etc.)
  - multiple phases of translation - each depending on the previous
  - not context independent
  - new standards complicate things further
- complex zero-cost abstractions end up in headers
- even the preprocessor can become a bottleneck
- sub-optimal tools and build system setups
- bad project structure - component and header dependencies
  - touch a header - rebuild everything

C vs C++ and different standards

C codebase (impl)

https://github.com/vurtun/nuklear

compiled as	time	increase
C	5 sec	baseline
C++98	5.6 sec	+12%
C++11	5.6 sec	+12%
C++17	5.8 sec	+14%

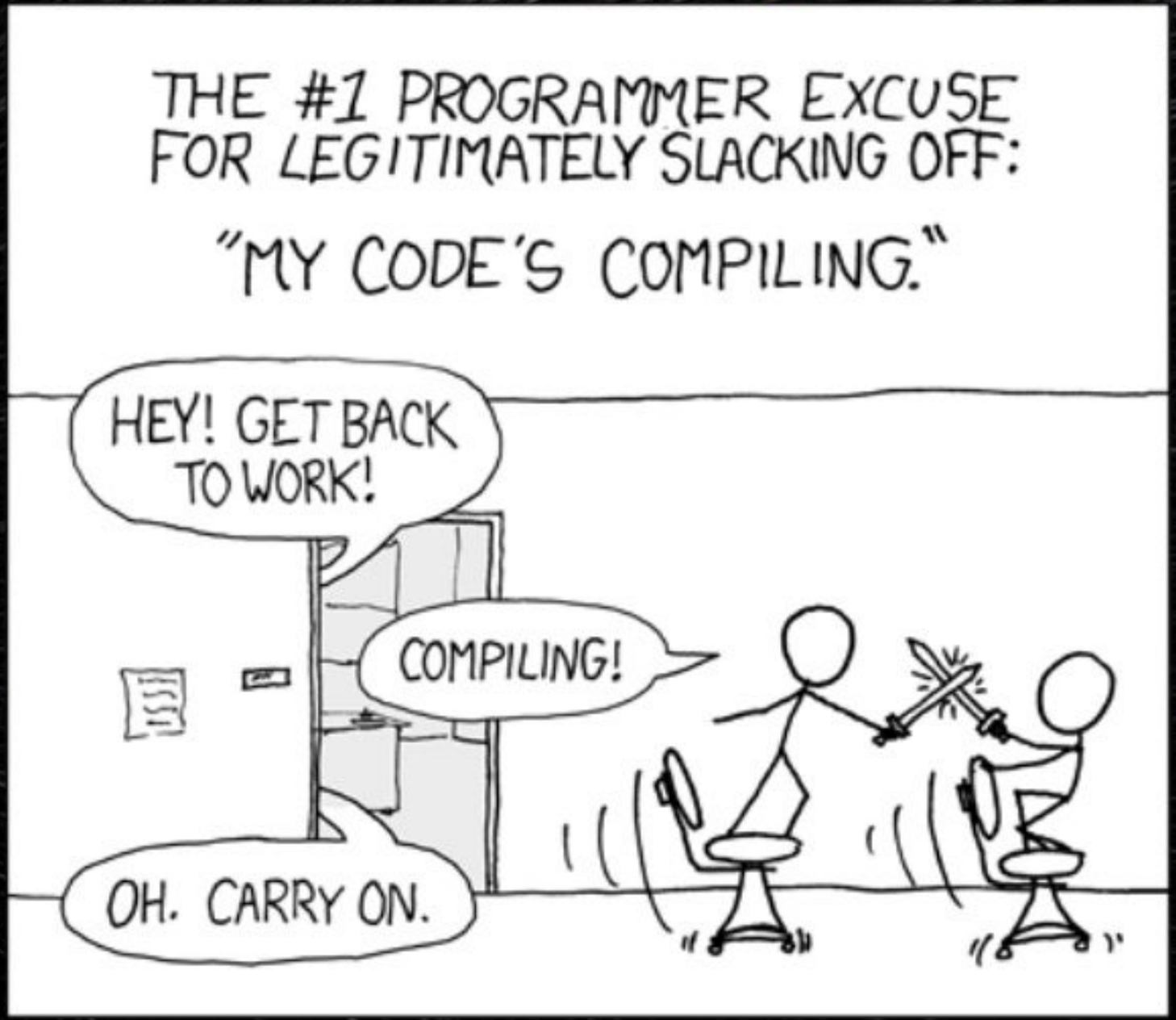
C++ codebase (impl + tests) (ver 1.2.8)

https://github.com/onqtam/doctest

compiled as	time	increase
C++98	8.4 sec	baseline
C++11	9.8 sec	+17%
C++17	10.8 sec	+29%

- Environment: GCC 7.2 on Windows, only compilation: "-c", in Debug: "-O0"
- optimizer doesn't care about language/standard: -O2 ==> same diff in seconds
- newer standards might be even heavier depending on the code
  - more constexpr, metaclasses, concepts - further pressure on compilers
  - constexpr - more complex compiler, but faster compile time algorithms

That meme again...



Why includes suck

- textual - not symbolic
  - parsed results cannot be cached - macros...
- implementation dependencies are leaked/dragged
  - privately used types by value need the definition
- the same code gets compiled many times
  - the same headers end up used in many sources
  - M headers with N source files ==> M x N build cost
- templates
  - re-instantiated for the same types
  - require more code to be in headers
- inline functions in headers (this includes templates)
  - more work for the compiler/linker

Standard includes - after the preprocessor

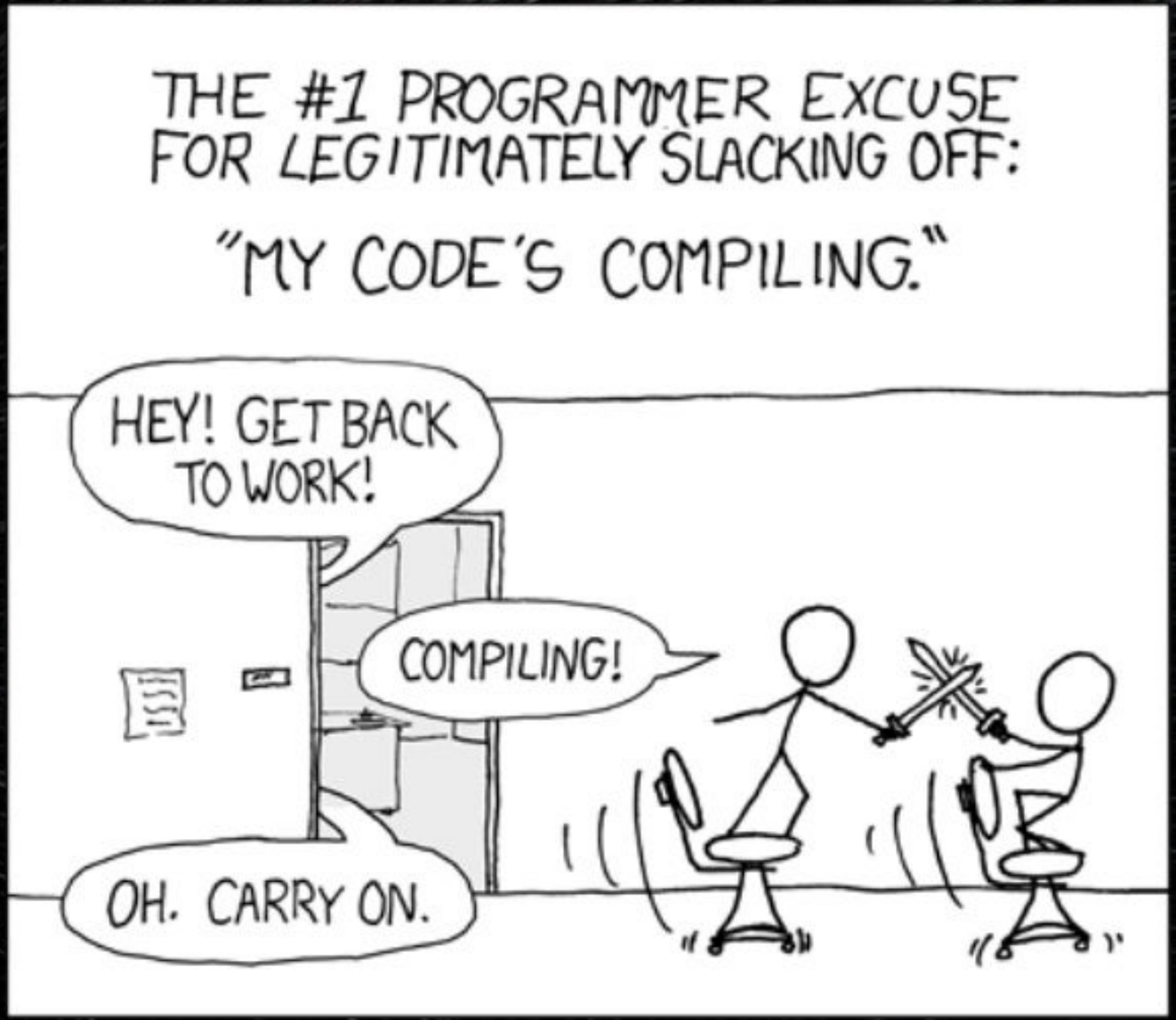
Header	GCC 7 - size	lines of code	MSVC 2017 - size	lines of code
cstdlib	43 kb	1k	158 kb	11k
cstdio	60 kb	1k	251 kb	12k
iosfwd	80 kb	1.7k	482 kb	23k
chrono	190 kb	6.6k	841 kb	31k
variant	282 kb	10k	1.1 mb	43k
vector	320 kb	13k	950 kb	45k
algorithm	446 kb	16k	880 kb	41k
string	500 kb	17k	1.1 mb	52k
optional	660 kb	22k	967 kb	37k
tuple	700 kb	23k	857 kb	33k
map	700 kb	24k	980 kb	46k
iostream	750 kb	26k	1.1 mb	52k
memory	852 kb	29k	1.0 mb	40k
random	1.1 mb	37k	1.4 mb	67k
functional	1.2 mb	42k	1.4 mb	58k
regex	1.7 mb	64k	1.5 mb	71k
ALL OF THEM	**2.6mb**	**95k**	**2.3mb**	**98k**

Boost includes - after the preprocessor

Header	GCC 7 - size	lines of code	MSVC 2017 - size	lines of code
hana	857 kb	24k	1.5 mb	69k
optional	1.6 mb	50k	2.2 mb	90k
variant	2 mb	65k	2.5 mb	124k
function	2 mb	68k	2.6 mb	118k
format	2.3 mb	75k	3.2 mb	158k
signals2	3.7 mb	120k	4.7 mb	250k
thread	5.8 mb	188k	4.8 mb	304k
asio	5.9 mb	194k	7.6 mb	513k
wave	6.5 mb	213k	6.7 mb	454k
spirit	6.6 mb	207k	7.8 mb	563k
geometry	9.6 mb	295k	9.8 mb	448k
ALL OF THEM	**18 mb**	**560k**	**16 mb**	**975k**

- environment: C++17, GCC 7, VS 2017, Boost version: 1.66
- 100 .cpp files including geometry ==> 1 GB of source code!
- not meant to discredit Boost - this is a C++ issue

Insert title of slide



Unreachable (dead) code

The C++ philosophy: "don't pay for what you don't use"

- GCC/Clang warning: -Wunused, -Wunused-function
  - -ffunction/data-sections -Wl,--gc-sections,--print-gc-sections
- cppcheck --enable=unusedFunction
- Coverity - DEADCODE/UNREACHABLE
- Understand by scitools
- CppDepend
- PVS Studio
- OCLint
- Parasoft C/C++ Test
- Polyspace
- oovcde - static and dynamic dead code detection
- others... also runtime coverage (gcov (& lcov for GUI), others)

might also uncover bugs & improve runtime!

Other low hanging fruit

- compile in Debug instead of Release if feasible
  - -O0, -O1, -O2, -O3, -Os - different results!
- unused sources/libraries
  - look for such after dead code removal
  - often a problem with makefiles and old cruft
  - MSVC warning LNK4221: This object file does not define any previously undefined public symbols
- compile each source file just once
  - scan the build logs for such instances
  - commonly used sources => static libraries

Finding unnecessary includes

- strive for loose coupling
  - specific includes is better than #include "everything.h"
  - more granular rebuilds when touching headers
- https://include-what-you-use.org/ - the most popular (Clang-based)
  - ```
    find_program(iwyu_path NAMES include-what-you-use iwyu)
    set_property(TARGET hello PROPERTY CXX_INCLUDE_WHAT_YOU_USE ${iwyu_path})
    ```
- Doxygen + Graphviz
- https://github.com/myint/cppclean
- Header Hero (or the fork) - article
- https://github.com/tomtom-international/cpp-dependencies
- ReSharper C++ Includes Analyzer << also has a test runner for doctest!
- Includator - plugin for Eclipse CDT (or the Cevelop IDE)
- https://gitlab.com/esr/deheader
- ProFactor IncludeManager - https://www.profactor.co.uk/
- MSVC: /showIncludes GCC: -H
    ```

Declarations vs definitions - functions

```
// interface.h

int foo(); // fwd decl - definition is in a source file somewhere

inline int bar() { // needs to be inline if the body is in a header
    return 42;
}

struct my_type {
    int method { return 666; } // implicitly inline - can get inlined!
};

template <typename T> // a template - "usually" has to be in a header
T identity(T in) { return in; } // implicitly inline
```

- move function definitions out of headers
  - builds will be faster
  - use link time optimizations for inlining (or unity builds!)
- some optimizations for templates exist - in later slides

Declarations vs definitions - types

```
// interface.h
#pragma once

struct foo; // fwd decl - don't #include "foo.h"!

#include "bar.h" // drag the definition of bar

struct my_type {
    foo f_1(foo obj); // fwd decl used by value in signature - OK

    foo* foo_ptr; // fwd decl used as pointer - OK
    bar member; // requires the definition of bar
};
```

```
// interface.cpp
#include "interface.h"

#include "foo.h" // drag the definition of foo

foo my_type::f_1(foo obj) { return obj; } // needs the definition of f
```

Declarations vs definitions - types

definition of types is necessary only when they are:

- used by value
  - including in class definitions (size for memory layout)
  - not in function declarations !!!

forward declarations are enough if:

- types are used by reference/pointer
  - pointer size is known by the compiler (4/8 bytes)
  - compiler doesn't need to know the size of types
- types are used by value in function declarations
  - either as a return type or as an argument
  - this is less known !!!
  - function definition will still need the full type definition
  - call site also needs the full type definition

PIMPL - pointer to implementation

- AKA compilation firewall
- holds a pointer to the implementation type

```
// widget.h
struct widget {
    widget();
    ~widget(); // cannot be defaulted here
    void foo();
private:
    struct impl; // just a fwd decl
    std::unique_ptr<impl> m_impl;
};
```

```
// widget.cpp
struct widget::impl {
   void foo() {}
};

widget::widget() : m_impl(std::make_unique<widget::impl>()) {}
widget::~widget() = default; // here it is possible
void widget::foo() { m_impl->foo(); }
```

PIMPL - pros & cons

PROS:

- breaks the interface/implementation dependency
  - less dragged headers ==> better compile times!
  - implementation data members no longer affect the size
- helps with preserving ABI when changing the implementation
  - lots of APIs use it (example: Autodesk Maya)

CONS:

- requires allocation, also space overhead for pointer
  - can use allocators
- extra indirection of calls - link time optimizations can help
- more code

on const propagation, moves, copies and other details:

http://en.cppreference.com/w/cpp/language/pimpl

Abstract interfaces & factories

```cpp
// interface.h

struct IBase {
    virtual int do_stuff() = 0;

    virtual ~IBase() = default;
};

// factory functions
std::unique_ptr<IBase> make_der_1(int);
std::unique_ptr<IBase> make_der_2();
```

Implementations of derived classes are obtained through factory functions

```cpp
// interface.cpp
#include "interface.h"

// implementation 1
struct Derived_1 : public IBase {
    int data;

    Derived_1(int in) : data(in) {}
    int do_stuff() override { return data; }
};

// implementation 2
struct Derived_2 : public IBase {
    int do_stuff() override { return 42; }
};

// factory functions
std::unique_ptr<IBase> make_der_1(int in) {
    return std::make_unique<Derived_1>(in);
}
std::unique_ptr<IBase> make_der_2() {
    return std::make_unique<Derived_2>();
}
```

Abstract interfaces & factories

PROS:

- breaks the interface/implementation dependency
- helps with preserving ABI (careful with the virtual interface)

CONS:

- requires allocation, also space overhead for pointer
- extra indirection of calls (virtual)
- more code
- users work with (smart) pointers instead of values
- link time optimizations - harder than for PIMPL

Precompiled headers

```
// precompiled.h
#pragma once

// system, runtime, STL
#include <cstdio>
#include <vector>
#include <map>

// third-party
#include <dynamix/dynamix.hpp>
#include <doctest.h>

// rarely changing project-specific
#include "utils/singleton.h"
#include "utils/transform.h"
```

- put headers that are used in many of the sources
- put headers that don't change often
- do regular audits to reflect the latest project needs
- often disregarded as a hack by many - HUGE mistake IMHO
  - easy setup and benefits can be huge

Precompiled headers - usage

- supported by mainstream compilers
  - MSVC: "/Yc" & "/Yu"
  - GCC/Clang: "-x c++-header"
  - only one PCH per translation unit :(
  - force-include for convenience
    - /FI (MSVC) or -include (GCC/Clang)
- build system support
  - Meson (native)
  - Buck (native)
  - FASTBuild (native)
  - CMake - with modules
    - cotire
    - cmake-precompiled-header

Unity builds

- AKA: amalgamated, jumbo, SCU
- include all original source files in one (or a few) unity source file(s)

```
// unity file
#include "core.cpp"
#include "widget.cpp"
#include "gui.cpp"
```

The main benefit is compile times:

- headers get parsed only once if included from multiple sources
- less compiler invocations
- less instantiation of the same templates
- HIGHLY underrated
- NOT because of reduced disk I/O - filesystems cache aggressively
- less linker work
- used at Ubisoft (14+ years), Unreal, WebKit

Unity builds - pros

- up to 10+ times faster (depends on modularity)
  - best gains: short sources + lots of includes
- faster (and not slower) LTO (WPO/LTCG)
- ODR (One Definition Rule) violations get caught
  - this one is huge - still no reliable tools

```cpp
// a.cpp
struct Foo {
    // implicit inline
    int method() { return 42; }
};
```

```cpp
// b.cpp
struct Foo {
    // implicit inline
    int method() { return 666; }
};
```

- enforces code hygiene
  - include guards in headers
  - no statics and anon namespaces in headers
  - less copy/paste of types & inline functions

Unity builds - cons

- might slow down some workflows
  - minimal rebuilds
  - might interfere with parallel compilation
  - these ^^ can be solved with tuning/configuring
- some C++ stops compiling as a unity
  - clashes of symbols in anonymous namespaces
  - overload ambiguities
  - using namespaces in sources
  - preprocessor
- miscompilation - rare (but possible)
  - successful compilation with a wrong result
  - preprocessor, overloads (also non-explicit 1 arg constructors)
  - good tests can help!

Unity builds - how to maintain

- cotire (as a CMake module, there are others)
- FASTBuild (build system)
- Visual Studio experimental support for Unity builds

these:

- can produce multiple unity source files
- can exclude files (if problematic, or for incremental builds)

others:

- Meson (build system)
- waf (build system)
- RudeBuild (Visual Studio extension)
- custom solution

Inlining annotations

explicitly disable inlining:

- __declspec(noinline) / __attribute__((noinline))
  - and careful with __forceinline / __attribute__((always_inline))
- leads to compile time savings
- might even help with runtime (instruction cache bloat)
  - implicit culprit: destructors
- especially for heavily repeated constructs
  - asserts in testing frameworks
- example: doctest - "The fastest feature-rich C++11 single-header testing framework"
  - CppCon 2017: Viktor Kirilov "Mix Tests and Production Code With Doctest - Implementation and usage"

Templates - argument-independent code

```
template <typename T>
class my_vector {
    int m_size;
    // ...
public:
    int size() const { return m_size; }
    // ...
};

// my_vector<T>::size() is instantiated
// for types such as int, char, etc...
// even though it doesn't depend on T
```

```
class my_vector_base { // type independent
protected:             // code is moved out
    int m_size;
public:
    int size() const { return m_size; }
};

template <typename T> // type dependent
class my_vector : public my_vector_base {
    // ...
public:
    // ...
};
```

- linkers sometimes can fold identical functions - but more work
  - only if proven you aren't taking their addresses (required by the standard)
- this way: less work for the compiler/linker
- now it is syntactically easier to move function bodies out of the header
  - use link time optimizations for inlining (or unity builds!)

Templates - C++11 extern template

```cpp
// interface.h
#include <vector>

template <typename T>
T identity(T in) { return in; }
```

```cpp
// somewhere_else.cpp
#include "interface.h"

// instantiation for float
template float identity<float>(float);
// instantiation for std::vector<int>
template class std::vector<int>;

// instantiation for identity<int>()
auto g_int_2 = identity(1);
```

```cpp
// somewhere.cpp
#include "interface.h"

// do not instantiate these templates here
extern template int identity<int>(int);
extern template float identity<float>(float);

extern template class std::vector<int>;

// will not instantiate
auto g_int = identity(5);
auto g_float = identity(15.f);
std::vector<int> a;

// will instantiate for bool
auto g_unsigned = identity(true);
```

- no instantiation/codegen where extern-ed but still has to be parsed...
  - precompiled headers or unity builds !!!!
- some STL implementations do it for std::basic_string<char, ...>
- https://arne-mertz.de/2019/02/extern-template-reduce-compile-times/

Templates - move functions out of headers

```cpp
// interface.h

// declaration
template <typename T>
T identity(T in);

template <typename T>
struct answer {
    // declaration
    T calculate();
};
```

```cpp
// user.cpp
#include "interface.h"

auto g_int = identity(666);
auto g_float = identity(0.541f);

auto g_answer = answer<int>().calculate();
```

```cpp
// interface.cpp
#include "interface.h"

// definition
template <typename T>
T identity(T in) { return in; }

// definition
template <typename T>
T answer<T>::calculate() {
    return T(42);
}

// explicit instantiations for int/float
template int identity<int>(int);
template float identity<float>(float);

// explicit instantiation for int
template struct answer<int>;
```

the class/function has to be explicitly instantiated somewhere for all types it is used with... or linker errors!

Templates - cost of operations

The "Rule of Chiel":

1. looking up a memoized type
2. adding a parameter to an alias call
3. adding a parameter to a type
4. calling an alias
5. instantiating a type
6. instantiating a function template
7. SFINAE

A fascinating talk:

code::dive 2017 – Odin Holmes – The fastest template metaprogramming in the West

Templates - other notes

- use variadic templates sparingly
- recursive templates are bad business
  - especially when not memoized
- recursive template arguments are even worse
- now that we have "constexpr if" we can use a lot less SFINAE
  - if constexpr might be preferrable even to tag dispatching

Templates - tools

- metashell - interactive template metaprogramming shell
  - Clang-based
  - GUI: https://github.com/RangelReale/msgui
- https://github.com/mikael-s-persson/templight
  - Clang-based
  - profile time/memory consumption of template instantiations
  - interactive debugging sessions
  - https://github.com/mikael-s-persson/templight-tools
- Cevelop IDE (Eclipse CDT based) - Template Visualization

Templates - other good talks

- CppCon 2017: Eddie Elizondo “Optimizing compilation times with Templates”
  - reducing compile times of Thrift-generated code: type tags to circumvent SFINAE and use overload resolution
  - diagnostics
    - -ftime-report - breakdown of time spent in compiler phases
    - templight
- C++Now 2018: Odin Holmes “Boost.TMP: Your DSL for Metaprogramming” - Alternative title: "Boost your TMP-Fu"
  - metaprogramming through composition
  - builds onto his previous talk
  - brigand, kvasir, metal, mp11 - http://metaben.ch/

Diagnostics - compilation

some functions trigger pathological cases (or bugs) in compilers/optimizers

Clang/GCC:

- -ftime-report - breakdown of different compiler phases
- -ftime-trace (Clang) - Chrome tracing flamegraph - by Aras Pranckevičius

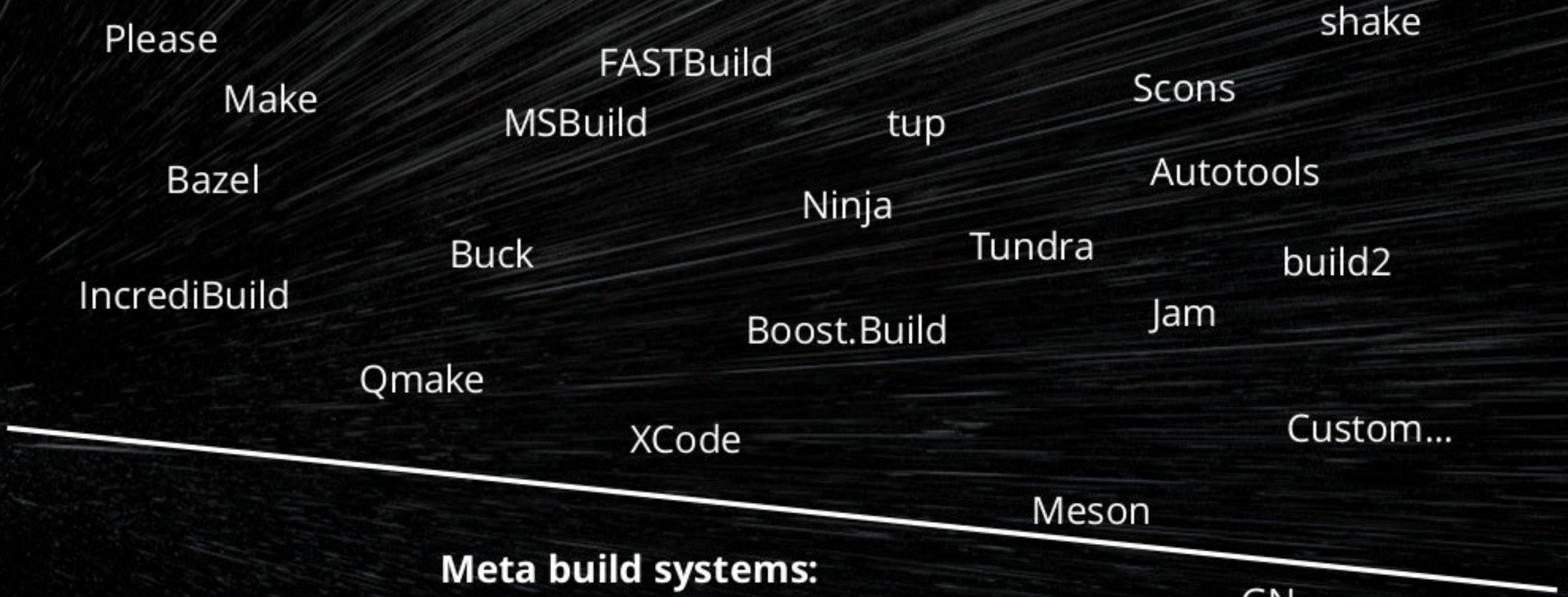
MSVC:

- /Bt - frontend & backend (optimizer/codegen)
- /time - for the linker
- /d2cgsummary - Anomalistic Compile Times for functions
  - first advertised by Aras Pranckevičius - "Best-unknown-MSVC-flag"
- /d1reportTime - includes, classes, functions - times - blog post
- /d1templateStats
- surprising 1 line fix => 20% faster builds thanks to these flags
- Updating VS 2017 15.6 to 15.8 => 2x slower for template instantiations

Diagnostics - binary bloat

- Bloaty McBloatface: a size profiler for ELF/Mach-O binaries
  - https://github.com/google/bloaty
- Sizer: Win32/64 executable size report utility
  - https://github.com/aras-p/sizer
- SymbolSort: measuring C++ code bloat in Windows binaries
  - https://github.com/adrianstone55/SymbolSort
- twiggy: call graph analyzer (wasm only, no ELF/Mach-O/PE/COFF yet)
  - https://github.com/rustwasm/twiggy
- nm - Unix tool to inspect the symbol table of binaries
  - nm --print-size --size-sort --demangle --radix=d YOUR_BINARY
- dumpbin: Microsoft COFF Binary File Dumper
- common culprit: templates (because length of names matters!!!)
  - demangling: undname.exe / c++filt / https://demangler.com

Build systems



Meta build systems:

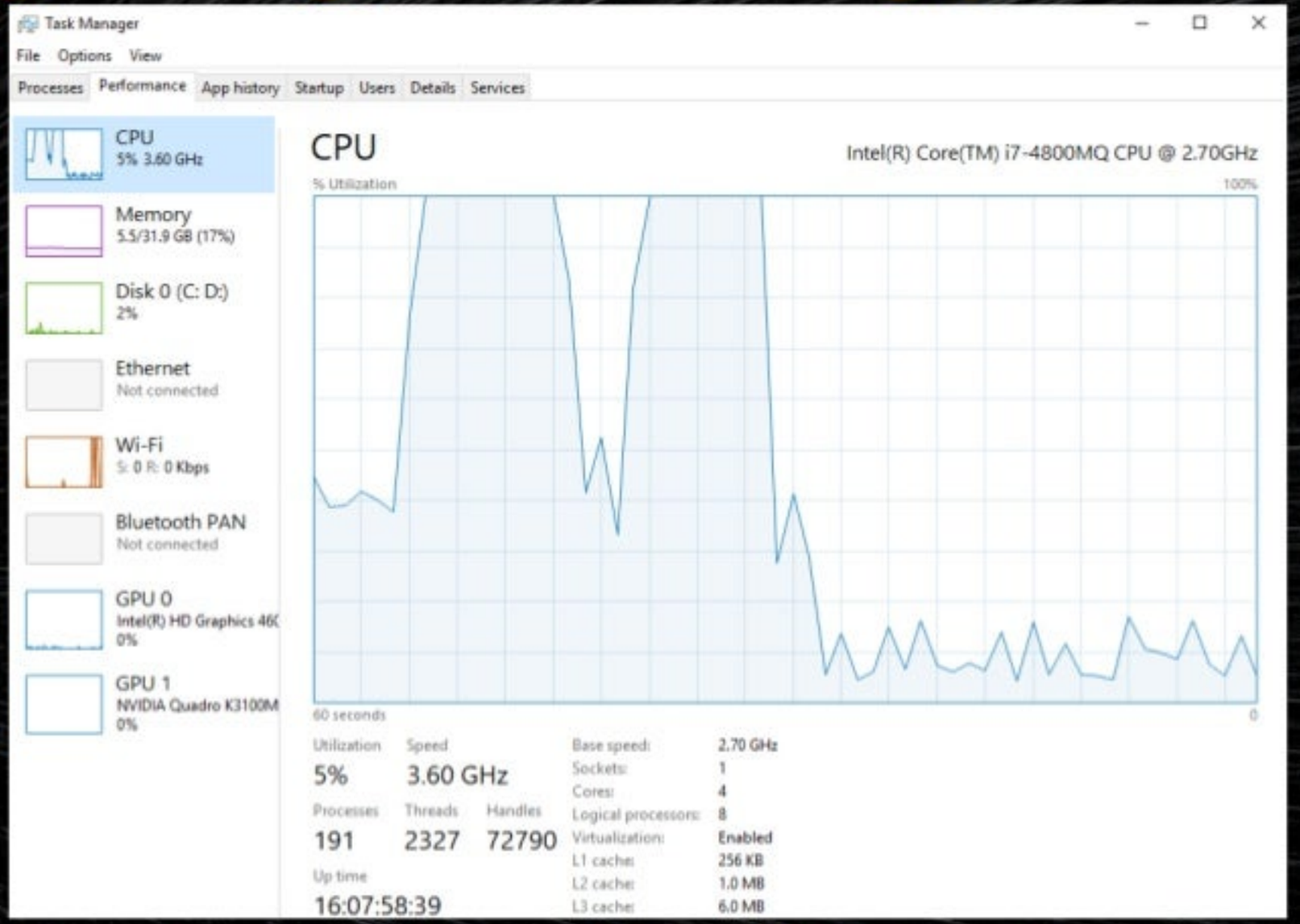
Build systems

- "g++ src1.cpp src2.cpp ..." is not a build system
- dependency tracking - for minimal and parallel rebuilds
- help managing scale
  - includes, flags, link libs
- help with refactoring/restructuring
- help with tool integration
- not all are created equal
- HUGE topic (trade-offs, approaches) - but lets focus just on speed
- most old build systems have high overhead (MSBuild, make)
  - suboptimal dependency and change detection/scanning

Parallel builds

- Ninja, FASTBuild, IncrediBuild: best in this regard
- make: -jN (and through CMake: "cmake --build . -- -jN")
- MSBuild (build system behind Visual Studio):
  - /MP[processMax] - in CMake: "target_compile_options(target /MP)"
  - inability to parallelize compilation of dependent dlls
    - IncrediBuild fixes this when integrated with Visual Studio
  - thread oversubscription (no overarching scheduler):
    - 8 projects in parallel (8 cpp each + /MP) ==> 64 threads!
  - Ninja instead of MSBuild as the VS backend!
  - since recently supports running Custom Build Tools in parallel
  - Make VC++ Compiles Fast Through Parallel Compilation - Bruce Dawson
- careful with unity builds and "long poles"
- modern LTO/LTCG technology is parallelizable too

Diagnosing/monitoring parallelism



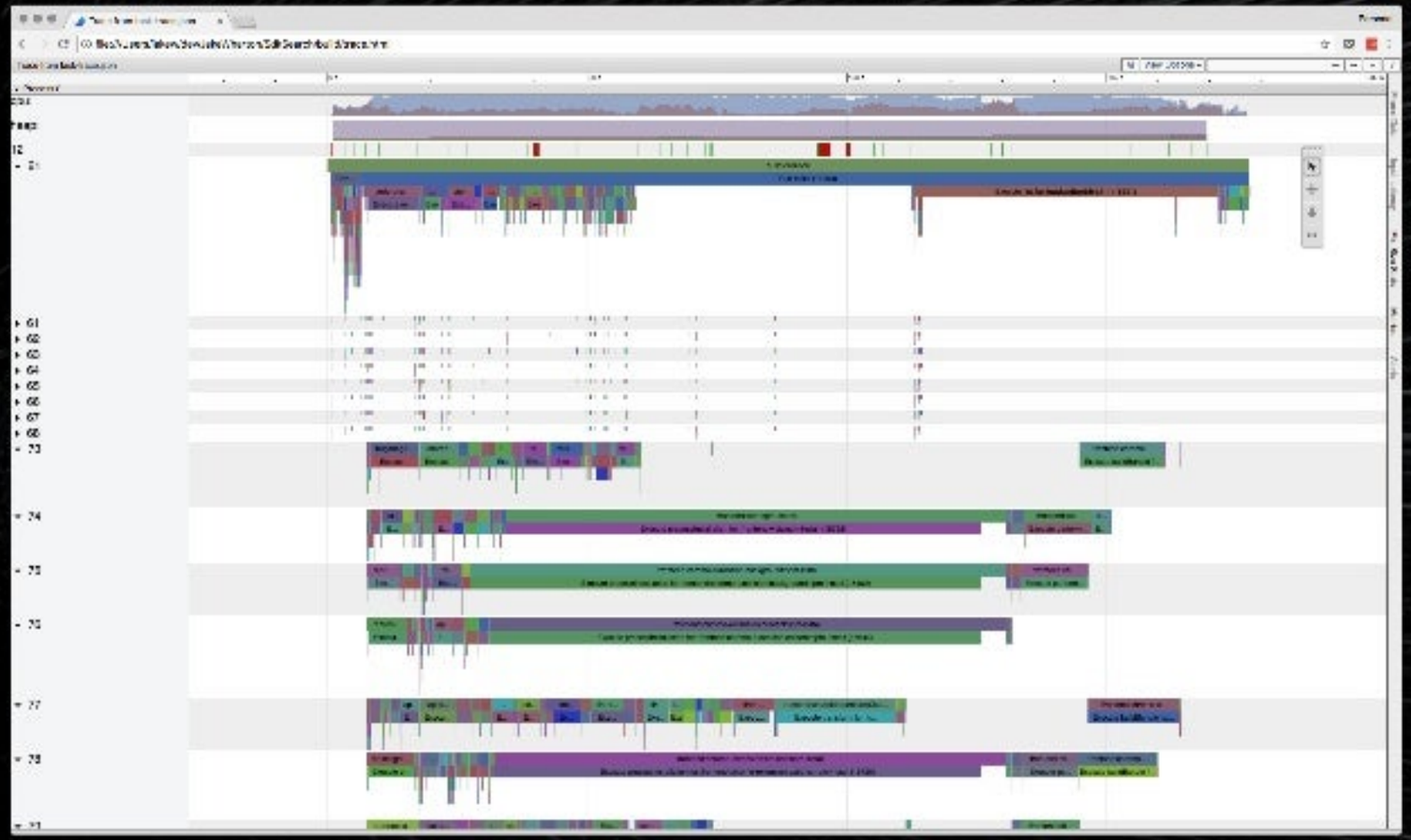
Make VC++ Compiles Fast Through Parallel Compilation - Bruce Dawson

Diagnosing/monitoring parallelism

IncrediBuild build monitor

Diagnosing/monitoring parallelism

Chrome's about:tracing (flame graph) - chrome://tracing/



- ninjatracing
- buck build system
- not related to parallelism, but for -ftime-trace in clang:
  - blog post
  - llvm mailing list
- other blog post

Sort-of winners (in terms of speed)

- FASTBuild
  - unity builds
  - precompiled headers
  - caching
  - distributed builds
  - best parallelization
  - generates project files
  - multiplatform
  - multi-compiler
  - tiny
  - might get a CMake backend
  - others...
- Ninja - less features but crazy fast - intented to be generated by a meta-build system
  - GN (GYP replacement)
  - can generate ONLY ninja
- IndrediBuild
- tup

haven't looked into these yet

- buck << by Facebook
- bazel << by Google
- build2 << modules! :D
- meson << hot contender
- ...

Static vs dynamic linking

prefer dynamic (.dll, .so, .dylib) over static (.lib, .a)

- big discussion overall - but here we care about build times
- much much faster - handles only the exported interface

Dynamic linking - symbol visibility

- Unix: all symbols exported by default
  - opposite to Windows - which got the default right!
- UNIX: compile with -fvisibility=hidden
  - improves link time
  - annotate the required parts of the API for exporting
  - circumvents the GOT (Global Offset Table) for calls => profit
  - load times also improved (extreme templates case: 45 times)
  - reduces the size of your DSO by 5-20%
  - for details: https://gcc.gnu.org/wiki/Visibility
- clang-cl on windows: /Zc:dllexportInlines- => 30% faster chrome build
- alternative: -fvisibility-inlines-hidden
  - all inlined member functions are hidden
  - no source code modifications needed
  - careful with local statics in inline functions
    - can also explicitly annotate with default visibility

Multiplatform symbol export annotations

```
#if defined _WIN32 || defined __CYGWIN__
#define SYMBOL_EXPORT __declspec(dllexport)
#define SYMBOL_IMPORT __declspec(dllimport)
#define SYMBOL_HIDDEN
#else
#define SYMBOL_EXPORT __attribute__((visibility("default")))
#define SYMBOL_IMPORT
#define SYMBOL_HIDDEN __attribute__((visibility("hidden")))
#endif

#ifdef DLL_EXPORTS // on this depends if we are exporting or importing
#define MY_API SYMBOL_EXPORT
#else
#define MY_API SYMBOL_IMPORT
#endif

MY_API void foo();

class MY_API MyClass {
   int c;
   SYMBOL_HIDDEN void privateMethod();  // only for use within this DSO
public:
   MyClass(int _c) : c(_c) { }
   static void foo(int a);
};
```

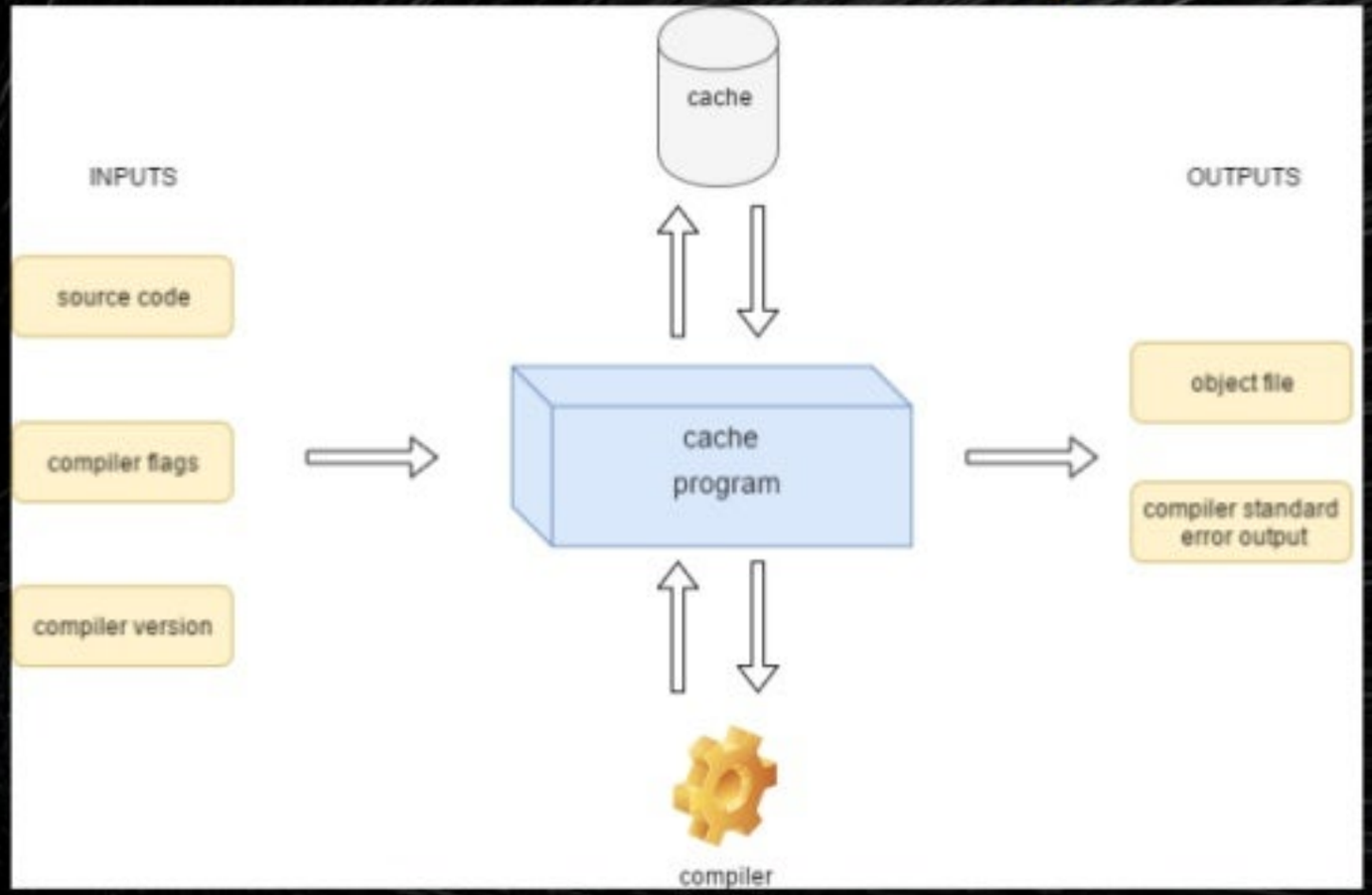
Linkers

- gold (unix)
  - comes with binutils starting from version 2.19
  - -fuse-ld=gold
  - not threaded by default
    - configure with "--enable-threads"
    - use: "--threads", "--thread-count COUNT", "--preread-archive-symbols"
- lld even faster than gold - https://lld.llvm.org/ - multithreaded by default
- Unix: Split DWARF - article - building clang/llvm codebase
  - 10%+ faster full builds and 40%+ faster incremental builds
- CMake - LINK_WHAT_YOU_USE

Linkers - LTO and other flags

- clang
  - ThinLTO: CppCon 2017: Teresa Johnson “ThinLTO: Scalable and Incremental Link-Time Optimization”
- GCC 8: link time and interprocedural optimization
- MSVC - lots of improvements, also regarding LTCG
  - lots of improvements in the recent years
    - Visual Studio 2017 Throughput Improvements and Advice
    - Speeding up the Incremental Developer Build Scenario
    - from x1.4 to x1.75 faster link times for Chrome with VS 2019
  - /debug:fastlink – generate partial PDB (new format)
  - /INCREMENTAL for debug
  - VS 2019 - huge speedups (~x2)
  - LTCG
    - /LTCG:incremental
    - also parallel - /cgthreads:8 instead of the default 4

Compiler caches

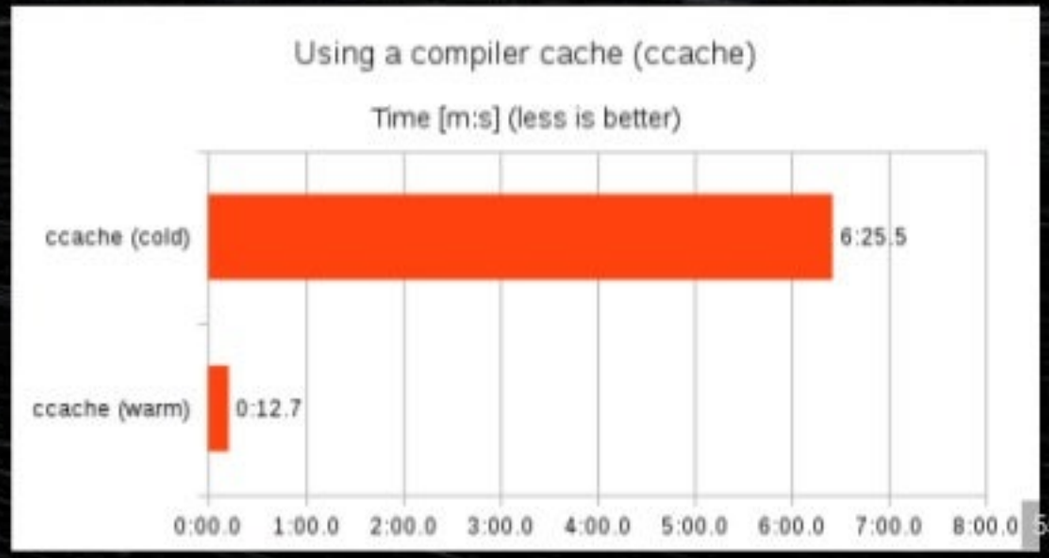


"compiler wrapper" -
maps a hash of:

- preprocessed source (TU)
- compilation flags
  & compiler version

==> to an object file

- First run (cold cache) is a bit slower than without the wrapper
- Consecutive >> FULL << rebuilds are much faster (warm cache)



Compiler caches

- For Unix alike: ccache (GCC, Clang), cachecc1 (GCC)
- For Windows: clcache (MSVC), cclash (MSVC)
- http://fastbuild.org/ - straight out of the build system!
- https://github.com/mozilla/sccache (GCC, Clang, MSVC)
  - ccache with cloud storage over AWS S3/GCS/Redis
- https://stashed.io/ (MSVC)
  - also distributed & with cloud storage
- zapcc - a Clang fork - not a wrapper! - mail thread
  - in memory caching server for requested compilations
  - biggest gains on heavy templates - article
  - recently open-sourced - future not clear
  - WebKit can compile x2~x5 times faster - more benchmarks

Compiler caches

- caches work best for full rebuilds - not minimal

ccache and CMake:

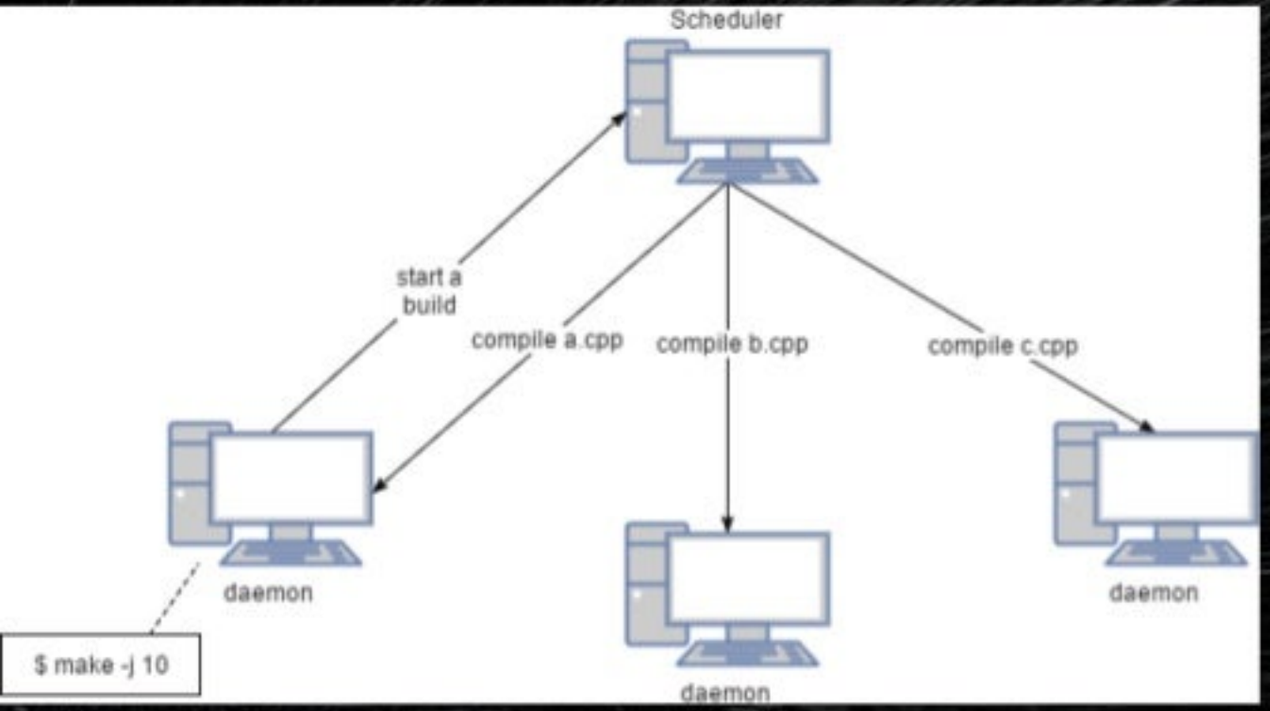
```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
    set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE "${CCACHE_PROGRAM}")
endif()

project(SomeProject)
```

- Using ccache with CMake
- Speed up C++ compilation, part 2: compiler cache

Other links:

- Faster C++ builds
- Speeding up the Build of C and C++ Projects

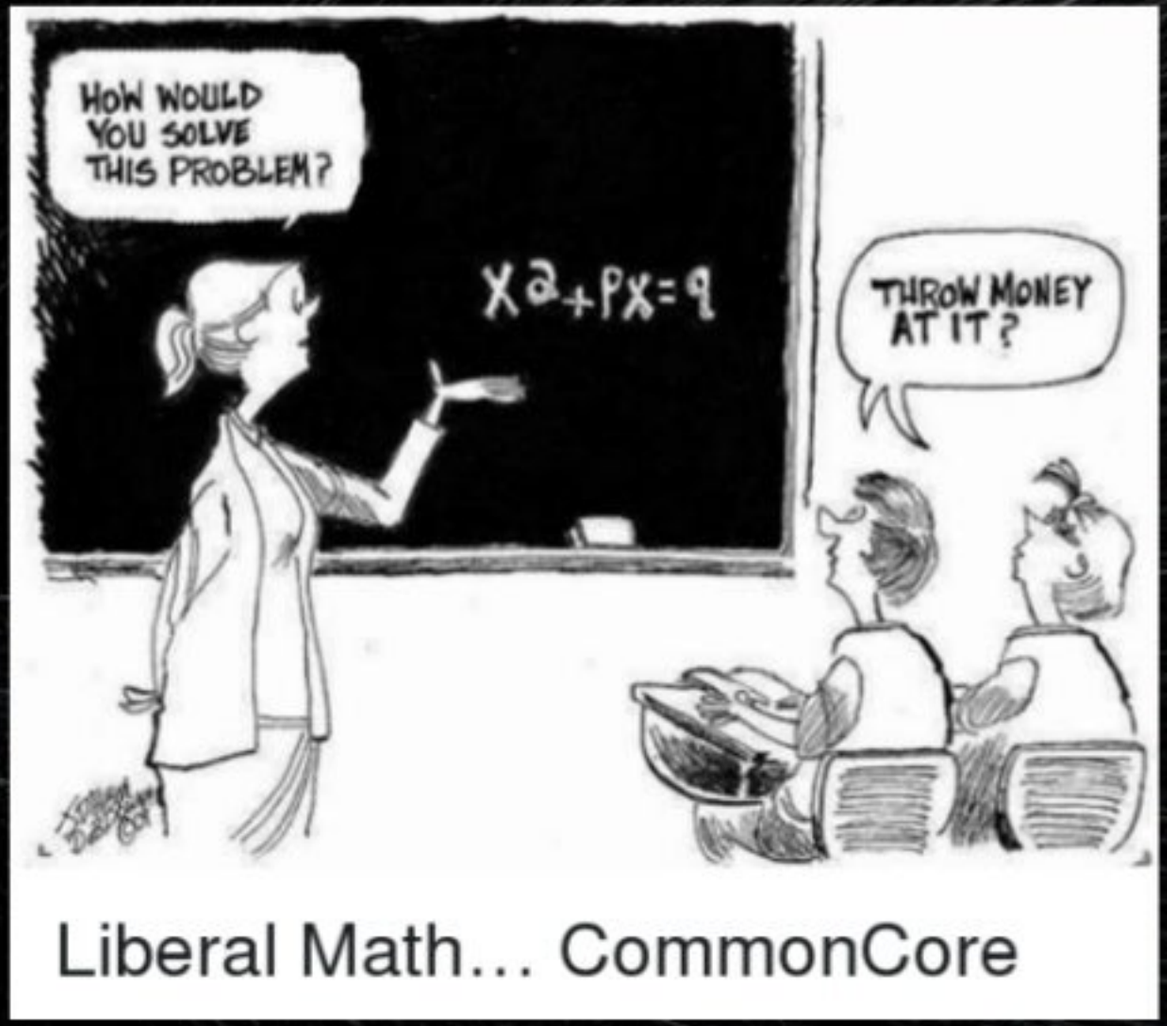
Distributed builds



- take advantage of unused processing power of computers
- prepend in front of compiler
  - just like caches: distcc gcc ...
- caching + distributed builds = ❤
  - here, here and here
  - `export CCACHE_PREFIX=DISTCC`
- watch out: network bottlenecks
  - include directories - sort em!
  - bandwidth

- https://github.com/distcc/distcc
- https://github.com/icecc/icecream
- https://www.incredibuild.com/
- OMG again that build system!!! ==> http://fastbuild.org/
- some article: Speed up C++ compilation, part 3: distributed compilation

Hardware



Liberal Math… CommonCore

Hardware

- more cores
  - make sure you have enough RAM && parallel builds setup
- RAM disks (file system drive in your RAM instead of ssd/hdd)
  - better than SSDs
  - don't expect huge gains though...
  - mount $TEMP as a RAM disk (%TEMP% for windows)
  - build output to RAM disk (most important - because of the link step)
  - also put the compiler there!
- CppCon 2014: Matt Hargett "Common-sense acceleration of your MLOC build"
  - touches on network I/O issues and many other topics

What if the fundament is bad

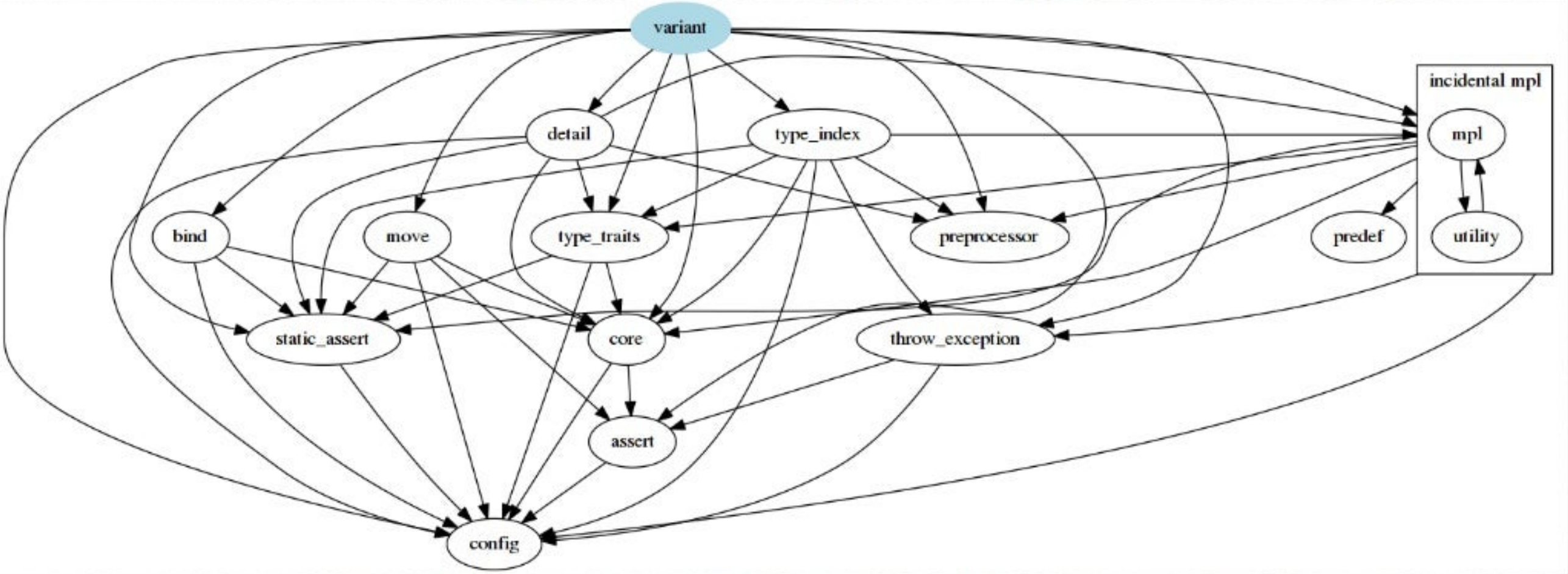
Physical design

- physical structure/design
  - often neglected
  - every project starts small... some end up big and it's too late
- reasoning about components and modularity - **IMPORTANT!!!**
  - physically (and logically): files, dependencies, includes
  - benefits: fast builds, easy to reason, easy to refactor
- Article: Physical Structure and C++ - A First Look
- John Lakos (Bloomberg)
  - book: "Large-scale C++ Software Design"
  - multi-part talk: "Advanced Levelization Techniques"
  - C++Now 2018: “C++ Modules & Large-Scale Development”
- Controversial article: Physical Design of The Machinery

Dependency analysis

- cmake --graphviz=[file]
  - options
  - visualize with Graphviz
  - some (most) build systems have such a feature (bazel)
- Most tools from the "Finding unnecessary includes" slide
  - Understand by scitools
  - CppDepend
  - cpp-dependencies - Graphviz
  - Doxygen - Graphviz
- maybe even SourceTrail
- IDE tools
  - Visual Studio Code Maps

Dependency analysis



Boost.Variant - Boost dependencies and bcp - by Stephen Kelly

- Boost.Spirit
- Boost.Geometry


MODULES

Modules - history

- 2004.11.05: first proposal: n1736 - Modules in C++
  - by Daveed Vandevoorde - working on EDG - the only compiler supporting exported templates (and aiding in their deprecation)
  - dropped from C++11 after a few revisions ==> headed for a TR
- 2011: work on first implementation in Clang - by Douglas Gregor
- 2014.05.27: standardization continues with n4047 - A Module System for C++
  - by Gabriel Dos Reis, Mark Hall and Gor Nishanov
  - after a few revisions - didn't enter C++17 - left as a TS
- latest work: will enter C++20
  - 2017.06.16: Business Requirements for Modules
  - 2018.01.29: n4720 - Working Draft, Extensions to C++ for Modules
  - 2018.03.06: Another take on Modules , p0924r0
  - 2018.05.02: Modules, Macros, and Build Systems
  - 2018.10.08: Merging Modules, p1218r0, p1242r0

Modules - motivation

- compile times...!!!
- preprocessor includes led to:
  - leaked symbols & implementation details
  - name clashes
  - making tooling hard
  - workarounds (include guards)
  - include order matters
- more DRY (don't repeat yourself)
  - forward declarations
  - code duplication in headers/sources

Modules - VS 2017 example

```
import std.core; // containers, string

export module M_1; // named M_1

export std::vector<std::string> getStrings() {
    return {"Plato", "Descartes", "Bacon"};
}
```

```
import std.core; // containers, algorithms, etc.
import M_1; // our module!

int main() {
    std::vector<std::string> v = getStrings();
    std::copy(v.begin(), v.end(),
        std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

```
cl.exe /experimental:module /EHsc /MD /std:c++latest /module:export             /c mod_1
cl.exe /experimental:module /EHsc /MD /std:c++latest /module:reference M_1.ifc /c main.
cl.exe main.obj mod_1.obj
```

Modules - how they work

- BMI (Binary Module Interface) files
  - consumed by importers
  - produced by compiling a MIU (Module Interface Unit)
  - (maybe) contains serialized AST (abstract syntax tree) with the exported parts of the parsed C++
- unlike precompiled headers:
  - composability of multiple modules
  - explicit code annotations to define visible interfaces

Modules - obsolete techniques

- PIMPL!!! (mostly)
  - unless you need to provide ABI stability of your APIs
- forward declarations (mostly)
- precompiled headers
- unity builds

Modules - the current state

- big companies are invested in them (think BIG)
- there are 3 implementations already (MSVC, Clang, GCC)
- coming in C++20
- when standardized and implemented in compilers:
  - do not expect astronomical gains in speed (for now)
  - build system support is hard
    - notably build2 supports modules!
  - slow adoption in existing software
    - updating toolchains
    - source code changes
      - third party dependencies
      - if it ain't "broke" don't fix it

Next steps

- consider runtime C++ compilation & hotswap
  - http://bit.ly/runtime-compilation-alternatives
- make source control checkouts faster
- tests
  - doctest? please!!! it's great, trust me :)
  - ctest has -j N
- Continuous integration
  - mtime_cache
- IDE integration of tools for better productivity
- change language :| Nim is the anwser *cough*

Q&A

- Slides: https://slides.com/onqtam/faster_builds
- Blog: http://onqtam.com
- GitHub: https://github.com/onqtam
-  : https://twitter.com/KirilovVik
- E-Mail: vik.kirilov@gmail.com

Other useful links:

- https://www.viva64.com/en/b/0549/
- http://www.bitsnbites.eu/faster-c-builds/
- https://github.com/green7ea/cpp-compilation