



THE QUESTIONS OF COMPONENTIAL C++ COMPILATION

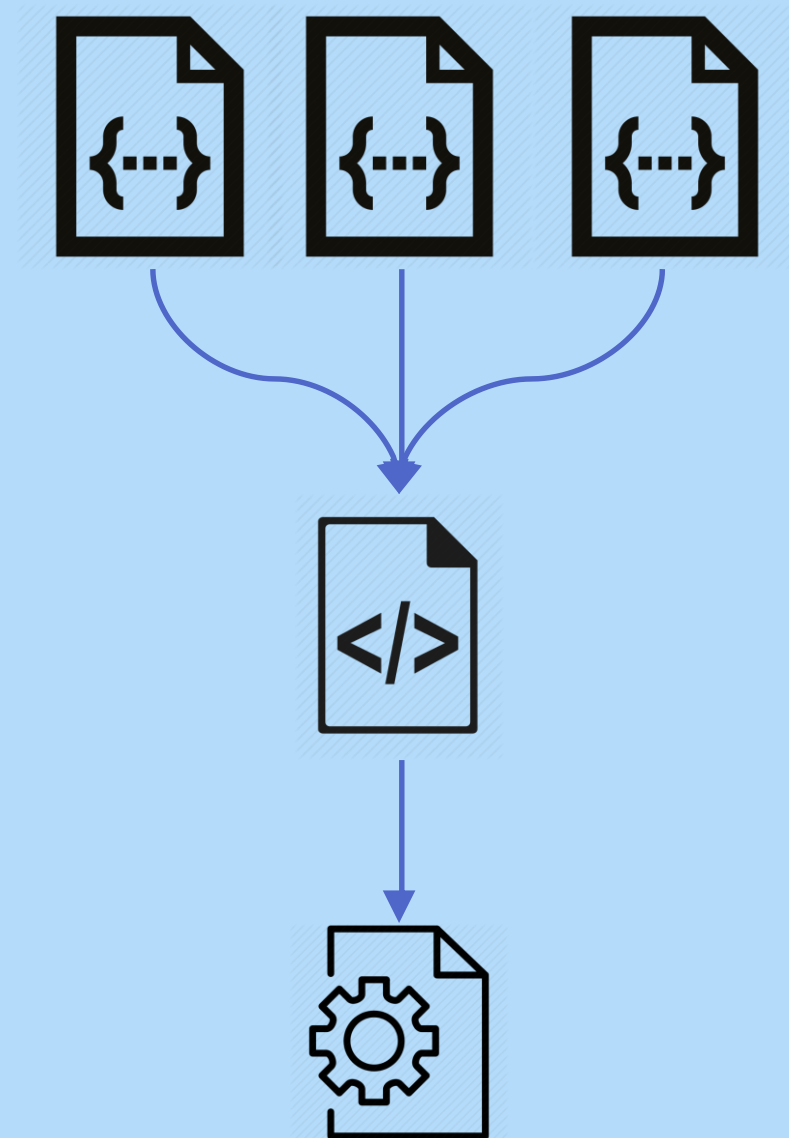
What the wild world can('t) do...

OUTLINE OF THE PROBLEM

Compiling C++ is hard ...
... and it keeps getting harder.

Compilation model

- Source file
 - *The entry point for a compilation, a translation unit*
- Headers
 - *Are just text files containing C++ code*
 - *Textually pulled into the compilation stream (via #include)*
- Preprocessor → Parsing and semantics → Lookups and semantic code generation → Compilation and output code generation
- Preprocessor supports string operations, such as textual emplace/replace (#define and #if(n)def)
- Compilation is done from a **single** textual buffer.



Headers are prevalent and important

- Set up interfaces and type information
 - ... *since C, and since ~40 years ago.*
- Contains actual executed code
 - *C++ inline methods*
- Contains blueprint for executed code
 - *C++ templates*
 - *Template instantiation also takes up resources.*

All source files	53
Depends on Thrift	13
Depends on ODB	30
Depends on LLVM	25
Depends on Boost	42

External library dependencies in CodeCompass.

- Headers are source text, they need to be understood every single time.
 - *And not just at every compilation...*
- Some of this generated code that should be executed will be removed later by the linker. This removal also has some overhead.
- One infinitesimal modification in any header → every dependent needs to be recompiled.

Headers are prevalent... a bit too much?

- One solution: *unity build*
 - Create a massive source text every time build is executed.
 - This eliminates copies of same header.
 - “Compile once, run once”?
 - Has more fallbacks than benefits (linkage issues, packaging errors, ...)
 - Is *not* modularisation.

Library	original LOC	preproc. LOC	preproc. impact	impact ratio
OpenSceneGraph	91205	16366213	17944%	} 4.90%
OpenSceneGraph unity		802053	879%	
wxWidgets	357642	40550041	11338%	} 3.85%
wxWidgets unity		1562461	437%	
Xerces	122396	2398884	1960%	} 9.87%
Xerces unity		236810	193%	

J. Mihalicza: *Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects*, Doctoral dissertation, Faculty of Informatics, ELTE, Hungary, 2014.

CACHING SOLUTIONS

Idea: Don't compile stuff that need not be compiled... multiple times



ccache

- First release in 2002.
- Overlays the compiler invocation from the user's perspective (supports most Unix-available compilers)
- Easy setup via *PATH* environment variable
- Caches the build output in a hash-based data storage
 - If the **same** build command is executed, fetch result from the cache and present it as result.
 - Otherwise, just call the compiler, then cache the result.

Compiler invocation command line
Dependent header paths and contents
List of output files
Compiler and environment metadata
Source file content hash

Compiler *stderr*
Compilation *output* file



ccache

- Does not support **linking**, or *precompiled headers* (later)
- Cache can be fine-tuned for file or size limit
- At hitting limit, the oldest not used file is eliminated.
- Cache is shareable after a small Unix-specific permission and environment setup.
- Only exact input match results in a cache hit.

Compiler invocation command line
Dependent header paths and contents
List of output files
Compiler and environment metadata
Source file content hash

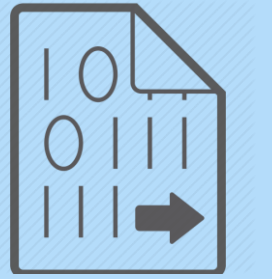
Compiler *stderr*
Compilation *output* file



ccache

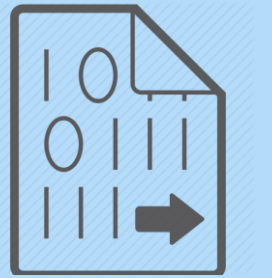
Project name	make	make clean; make	Change a header file with average prominency, make	Revert file, make
“Hello World”	2 sec	72 ms	1s 18ms	58 ms
LLVM, Clang, CTE 6.0 (Static linking, all targets)	48m 29s	20m 45s	8m 30s	5m
	2749 cache miss 5118 files (12.2 GiB) cache	50% cache hit	+ 203 cache miss + 203 files (2.3 GiB)	Back to 50% cache hit
LLVM, Clang, CTE 6.0 (Dynamic linking, X86 target, <i>gold</i> linker)	23m 6s	1m 4s	5m 11s	22s
	2226 cache miss 4071 files (360 MiB)	50% cache hit	+ 203 cache miss + 203 files (111 MiB)	Back to 50% cache hit

Precompiled headers



- Create a **compiled** *binary representation of the semantically analyzed and compiler instantiated* header contents.
- Use this header when compiling actual source files.
- Eliminate the need of having to parse the headers.

- Introduced in 1998 (MSVC), supported by GCC since 2004, Clang since 2009.
- **Not** a standard-backed thing, entirely driven by performance concerns.



Using precompiled headers

- `clang++ -xc++-header header.h -o header.h.pch`
 - *Creates the PCH output file.*
- `clang++ -include header.h source.cpp -o source.out`
 - *Automatically prefix the source code with `#include "header.h"`*
 - *If a `header.h.pch` exists, use that to load contents.*
 - *Otherwise fall back to standard compilation*
- Only **one** PCH can be used at translating one translation unit!
- If multiple `-include` specified: warning, use first.
- PCH contents **replace** the `<builtin>` stack.

Precompiled headers

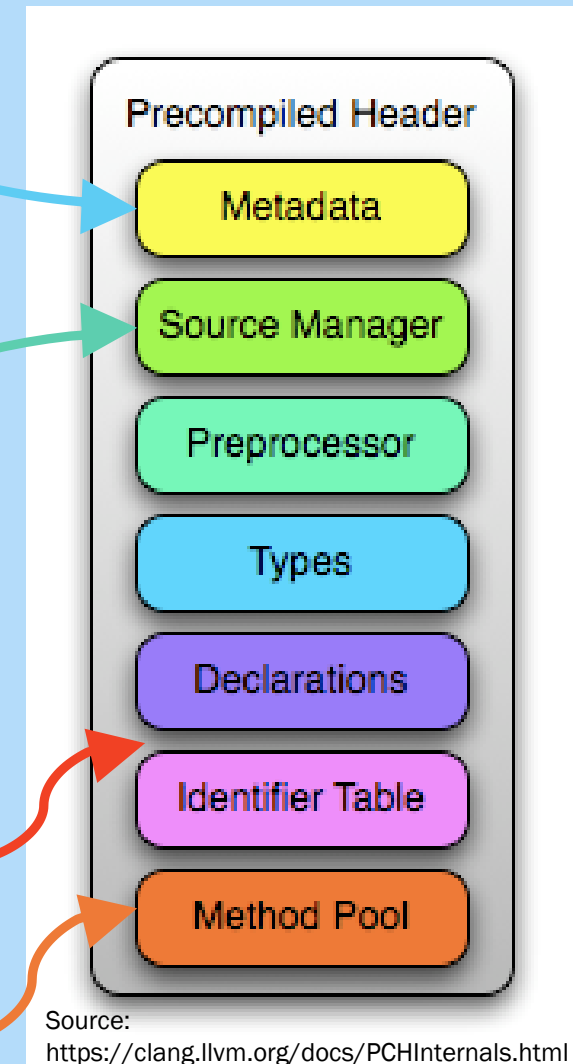
- Chained compilation can be done
 - *Still exactly one previous header + exactly one current header = exactly one header output.*
 - *Results in optimal representation if done with Clang.*
- GCC built-in around 10x the size of Clang's.
 - *GCH instead of PCH, most likely similar implementation.*
- Macros can bleed out and mess things up, thus caching PCHs is counter-intuitive.

Compiler setup,
target, file layout

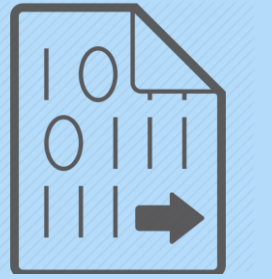
File locations, source ranges

Statements, expressions
(missing from Clang documentation image)

(Objective-C specific overload resolution)



Autogenerating PCHs?



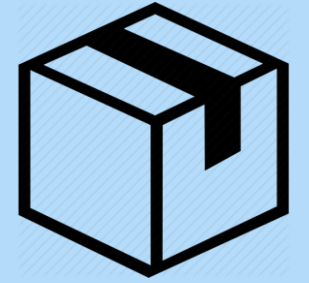
- PCHs are a maintenance burden – extra commands are needed to create them.
- *Cotire* ^[1]
 - *CMake plugin that auto-generates unity build and PCH setup for out-of-tree library headers*
- *aCC* (HP C++) compiler ^[2]
 - *For every translation unit, precompile the header area: the range from start of file to first non-preprocessor non-comment statement*
- *IncludeWhatYouUse* ^[3]
 - *Include usage analysis with libClang to eliminate unnecessary inclusion of headers which aren't used in the current translation.*
 - *Experimental, tailored for Google source code, and claimed to be buggy.*

[1]: <http://github.com/sakra/cotire>

[2]: T. Krishnaswamy: *Automatic Precompiled Headers: Speeding up C++ Application Build Times*, 2002.

[3]: <http://github.com/include-what-you-use/include-what-you-use>

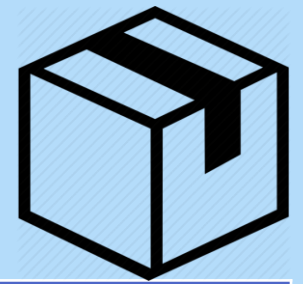
C++ modules



- Standard proposal to enable componential, compartmented compilation
 - *Existed as a proposal for 10+ years*
- Introduced as a new language construct, akin to Java, Python, etc. packages
- Incrementally coming into the language, *eventually* replacing *almost all* header mechanism.
 - *Designed for peaceful coexistence with the wild world.*
- Driven by performance **and** code quality concerns.

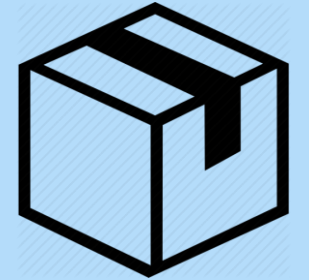
C++ modules

[1] G. D. Reis, M. Hall, G. Nishanov: A Module System for C++ (Revision 4), Open-STD document P0142R0



Standards proposal ^[1]	Clang 7.0 “Modules TS” implementation
Explicitly defined interface (<code>module</code> , <code>export</code> , ...) of a module that is understood by the compiler when a module is requested	Modules are made from headers, a header-to-module association is saved in a <i>module map</i>
Modules have a symbolic name and a way to specify usage	<code>import</code> (Clang), or <code>@import</code> in GCC
Modules do not introduce new scoping and name lookup rules	
1 module = 1 semantically separate translation <ul style="list-style-type: none">The TU’s preprocessor shouldn’t affect it	The compiler forks itself for every module with a fresh preprocessor (only inheriting command-line)
Modules are full-fledged TUs, importing one shouldn’t affect overload and template resolution	Using submodules in Clang is not working well and can cause issues
Order of import does not matter	It does for macros
Non-exported symbols can clash in two modules, but does not cause an issue in a TU importing both	Clang currently only performs a <i>minimal</i> check for ODR violation
It is NOT possible for modules to export macro definitions	Modules are internally translated to PCHs, which result in a weird behaviour regarding this

C++ modules



- Clang modules are translated to PCHs, which result in weird behaviour with macros.
 - *Every macro directive in the import order (which is not defined!) override previously visible (defined in the current (sub)module or TU) definitions...*
 - but an `#undef` always overrides a `#define`
 - *The active directives (those that are not overridden by the end of the importing) are inconsistent if the same name has different states (one undefines, the other defines, or two defines but differently)*
 - *In this case, the program is ill-formed.*
- `llvm-modularize` can help validate this “macro stack”

Let M1

```
#define X foo
```

and export this definition

Let M2

```
import M1;
```

```
#undef X
```

and export this explicit undefinition

Sources importing **both** M1 and M2 *in any order* are well-formed (??) and will see X as **undefined**.

This is an example from the documentation!



zapcc



- Compilation server architecture forked from Clang 5
- Around 400k lines of diff (manually reducible to 170k)
 - **Zero** documentation and project history, was supposed to be proprietary!
- Drop-in replacement for Clang
- For a build run: `zapcc` (client) forwards arguments and files to a `zapccs` server (started automatically in the local shell), which handles builds
- Source files **should not** change in the same *make* invocation but between two compilations.
- The server handles caching in memory...


zapcc



- The server handles caching:
 - *Macro states and pre-processor stack*
 - *Template instantiations (subtrees)*
 - This is claimed to be the most important overhead of compilation
 - *On-the-fly header modifications*
 - *IR builder intrinsics and diagnostic details*
 - *Frontend data and Modules (LLVM IR modules, **not C++ modules**)*
 - LLVM IR modules means globals, functions, co-dependencies, symbol tables, and target-specific details
- The server process uses the rewritten Clang as library
- Cache lost when memory usage gets high, or when the server is killed
- Carefully evicts parts of cache between two TUs

zapcc



	G++ 7.3 (Ubuntu 18.03 default)	Clang 5.0 release	zapcc
LLVM, Clang, CTE 6.0 (Dynamic linking with <i>gold</i> , every target)	26m 53s	22m 15s	10m 39s 

However, *zapcc* has issues!

- Macro definition conflict errors happen all the time.
- Subsequent compilations of the same project (without modifications) result in different errors.

Compilation “attempt”	Macro error count	Linker error action count	Linker error symbol count
1.	42	1	8 in X86
2.	29	2	5+6 in ARM
3.	40	0	

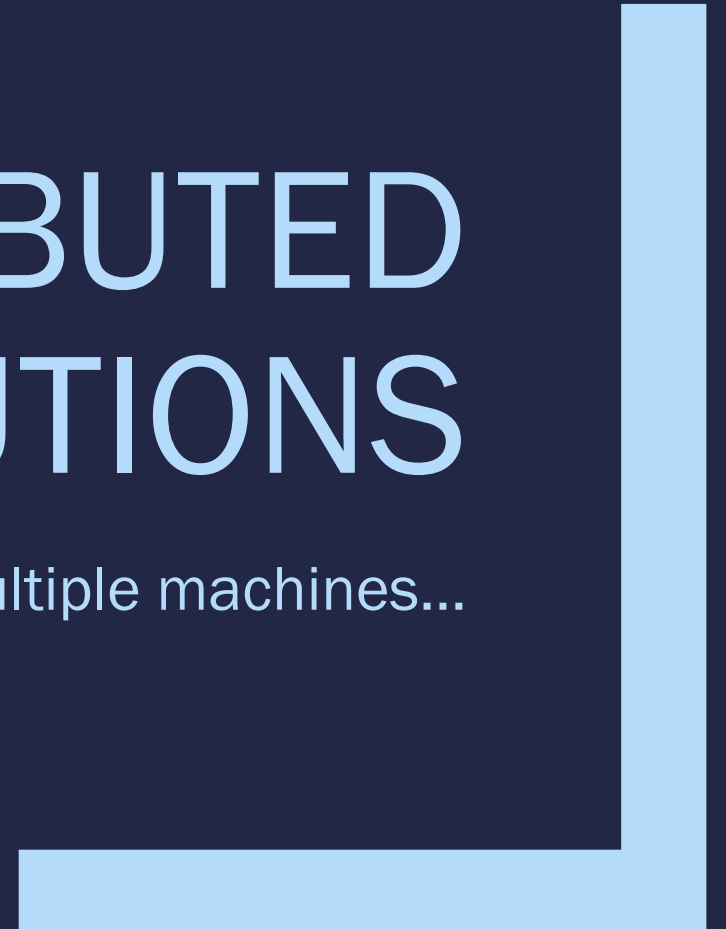
Attempt in a single install

Compilation “attempt”	Macro error count	Linker error action count	Linker error symbol count
1.	36	1	8 in X86
2.	35	1	6 in ARM
3.	32	3	6 in AArch64 5 in Arm 5 in Hexagon

Reattempt with a clean install again

DISTRIBUTED COMPILATION SOLUTIONS

If we need to build, at least do it on multiple machines...



distcc

- C++ compilation is single-thread, and there is only so much a single machine can do
- First release in 2002, last major release (3.0) in 2008
- Set up servers with a C/C++ compiler and assembler on them
- *distcc* wraps over compilers on local machine to
 - *Send preprocessed output to server*
 - *Receive compiled objects*
- (On the servers, *ccache* can work in conjunction with *distcc*!)
- *Pump-mode*: Send sources and headers directly to servers, and use incremental include analysis on the local machine to figure out what is needed.
 - *Uses a **local** Python implementation*
 - *No full preprocessing done for every TU*
 - *Servers handle preprocessing*

Bazel Build



- Open-source version of tool used by Google
- Written with a Java backend
- Custom DSL for a *very high level* description of how the build should take place
 - *Orders of magnitude more abstract than CMake's scripting language*
 - *But can do fancies, like fetching dependencies mid-build*
 - *cc_library(), cc_executable() and cc_test()*
- Locally, only the latest build information is cached
 - *The build's input SHOULD not change by build-to-build conditions, or mid-compilation!*
- Remote caching to many backends (nginx, SSH, Google Cloud) without any garbage collection
 - *This cache works akin to ccache, but much more detailed*
 - *No management of the cache – garbage collection is done via `rm -rf`*
- Bazel Remote Server
 - *Experimental and unsupported, but allows for garbage collection of old builds*