IMPROVING COMPILATION TIMES

Tools & Techniques

purecpp 2023

24th October 2023

Vittorio Romeo

mail@vittorioromeo.com

@supahvee1234

Bloomberg Engineering

TechAtBloomberg.com

Careers

A bit about me

- I've been working with C++ for over 10 years
  - Started thanks to game development 🕹️🎮
- 6+ YoE at Bloomberg
  - Currently teaching Modern C++
- Co-authored *"Embracing Modern C++ Safely"*
  - J. Lakos, R. Khlebnikov, A. Meredith, and many other contributors
- Participating in ISO C++ standardization
  - Member of the Italian national body
- Many open-source side projects, including:
  - Modernizing SFML from C++03 to C++17
  - Game development: Open Hexagon, Quake VR
  - Tools & libraries: majsdown, ecst, scelta
  - Video tutorials on YouTube
  - Articles on vittorioromeo.com

About this talk

- Why should we care about compilation times?
- Improving compilation times: a flowchart-based approach
  - *Part 1*: Low-hanging fruits
  - *Part 2*: Profiling and dealing with bottlenecks
- A few remarks on C++20 modules
- Benchmarks and examples from SFML 3.x
- Goals:
  - ☐ **Understand what can negatively impact build times**
  - ☐ **Provide *actionable* points to improve your compilation times**
  - ☐ **Call to action: improve your favorite open-source project's build**
  - ☐ **Spark some interesting discussion!**

Before we begin...

- Assumptions
  - You are somewhat familiar with C++'s build model
    - Parallel compilation, header/source files, linking, ...
  - You are somewhat familiar with C++ 🙄
    - Declarations, definitions, templates, overload resolution, ODR, ...

- Disclaimer
  - This talk focuses mostly on CMake and UNIX
  - 100% applicable to GNU/Linux, WSL, MSYS2 + MinGW
  - Some details not applicable to MSVC or other build systems
  - Will mostly cover techniques and tools I am familiar with

- *"There's no such thing as a stupid question"*
  - Feel free to interrupt me

- Measurements and assertions have sources and references [src]
  - Slides available at: https://github.com/vittorioromeo/purecpp2023

Why care about compilation times?

- C++ has the reputation to be slow to compile
  - Especially compared to languages like C
  - *"Zero-cost abstractions"* can have a large build time cost

- Compilation times matter
  - *"Time is money"* – we could do the math
  - Not only developer time, but also CI time/power usage
  - Often overlooked: programmer motivation and experimentation
    - **Q:** have you ever felt frustrated?
  - Short iteration times $\longrightarrow$ better products and development experience

- Build times can get out of hand easily, even on modern hardware
  - Chromium takes around ~50min on a very strong setup [src]
  - GCC takes on average ~90min [src]
  - LLVM takes on average ~30min [src]

- Build times can very often be improved significantly
  - Especially if you have never cared that much before!

Why can C++ compilation times be poor?

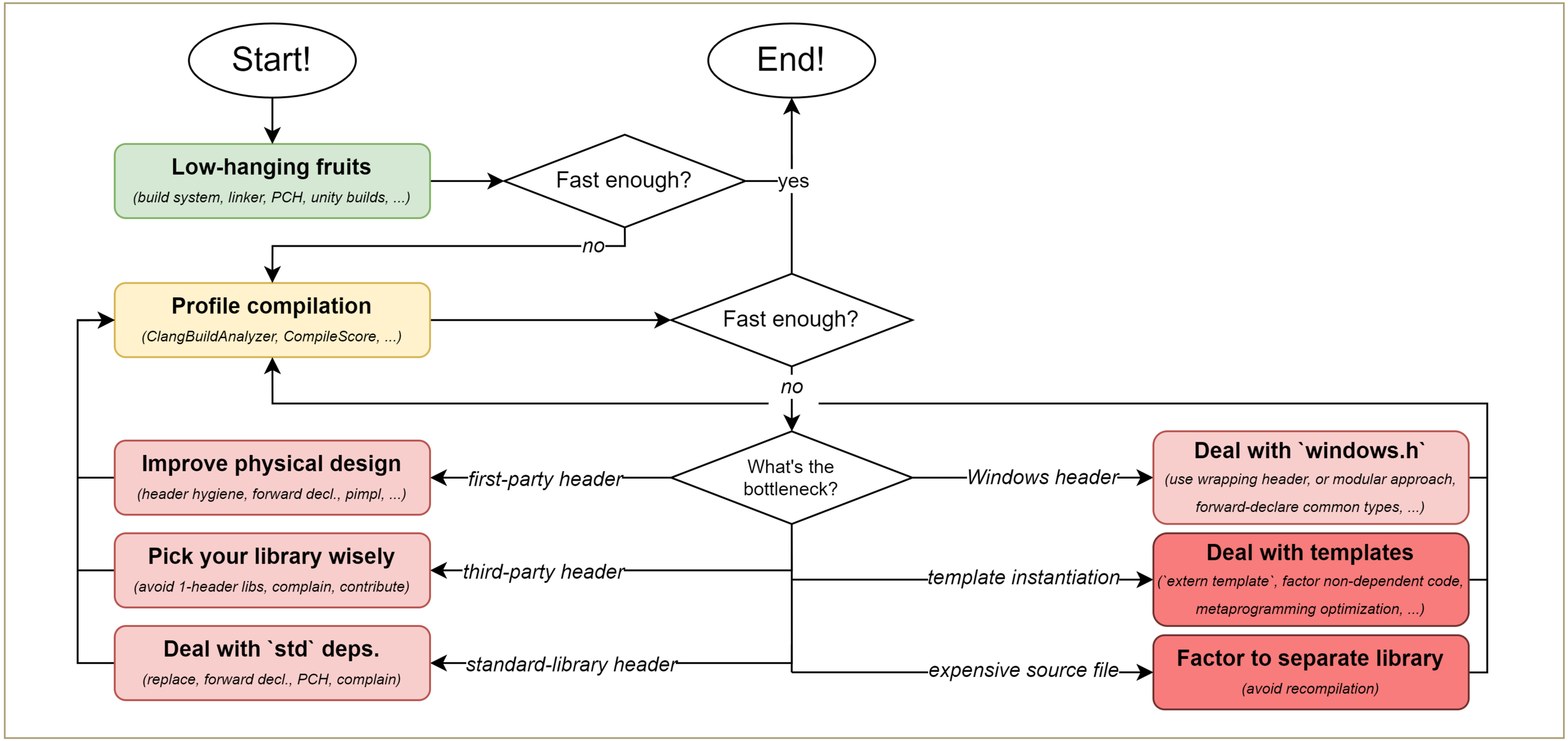
- Build model and textual `#include` system is archaic
- The language itself is complicated
  - Overload resolution, template instantiation, SFINAE, etc...
- Highly generic and abstracted libraries tend to be bulky
  - Think about Boost or the Standard Library
  - Many reasons: backwards compatibility, build time not a priority, etc...
- Poor “physical design”
  - E.g., `#include` when forward declaration is enough
  - E.g., templates unnecessarily defined in a header file
- Compilation times are often not a priority for low-level libraries
  - This includes the Standard Library
  - Needs a “cultural” change

What about modules?

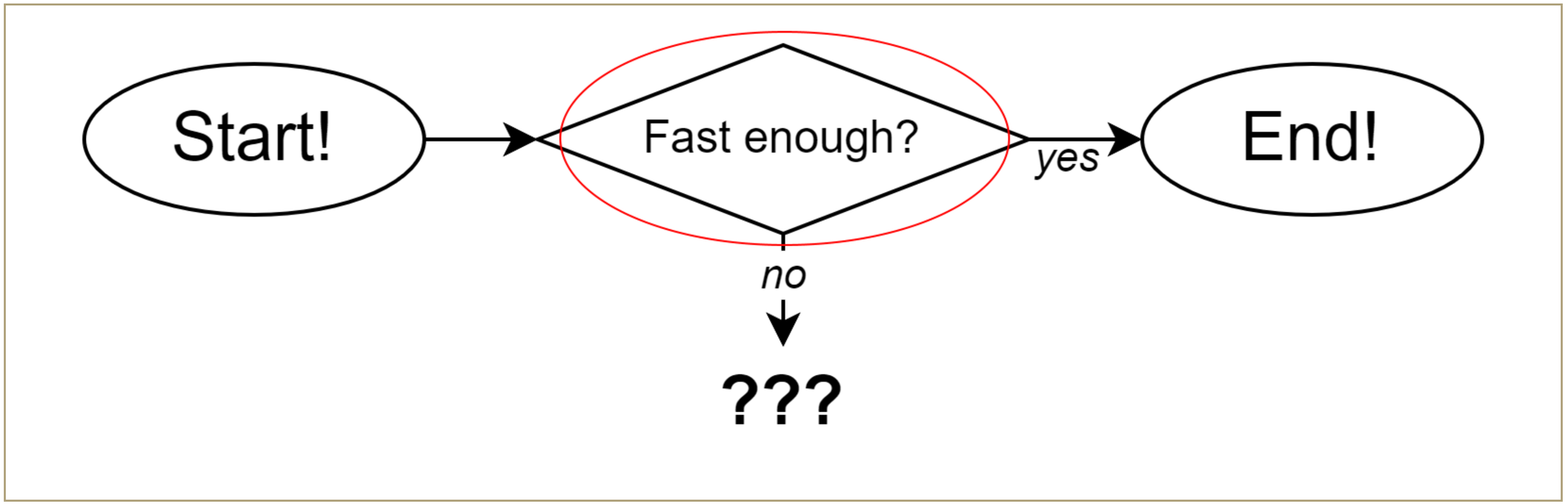
- In short: they *will* help, but we're not there yet
- Compiler support for modules is still limited [src]
- Libraries and projects need to migrate
  - It takes considerable effort
- Promising compilation time speedups [src]
- Being actively worked on
  - e.g., Bloomberg sponsoring Kitware [src]
- More information at the end of the talk
  - Let's focus on what you have control over *today*

The flowchart

Sneak peek – complete flowchart

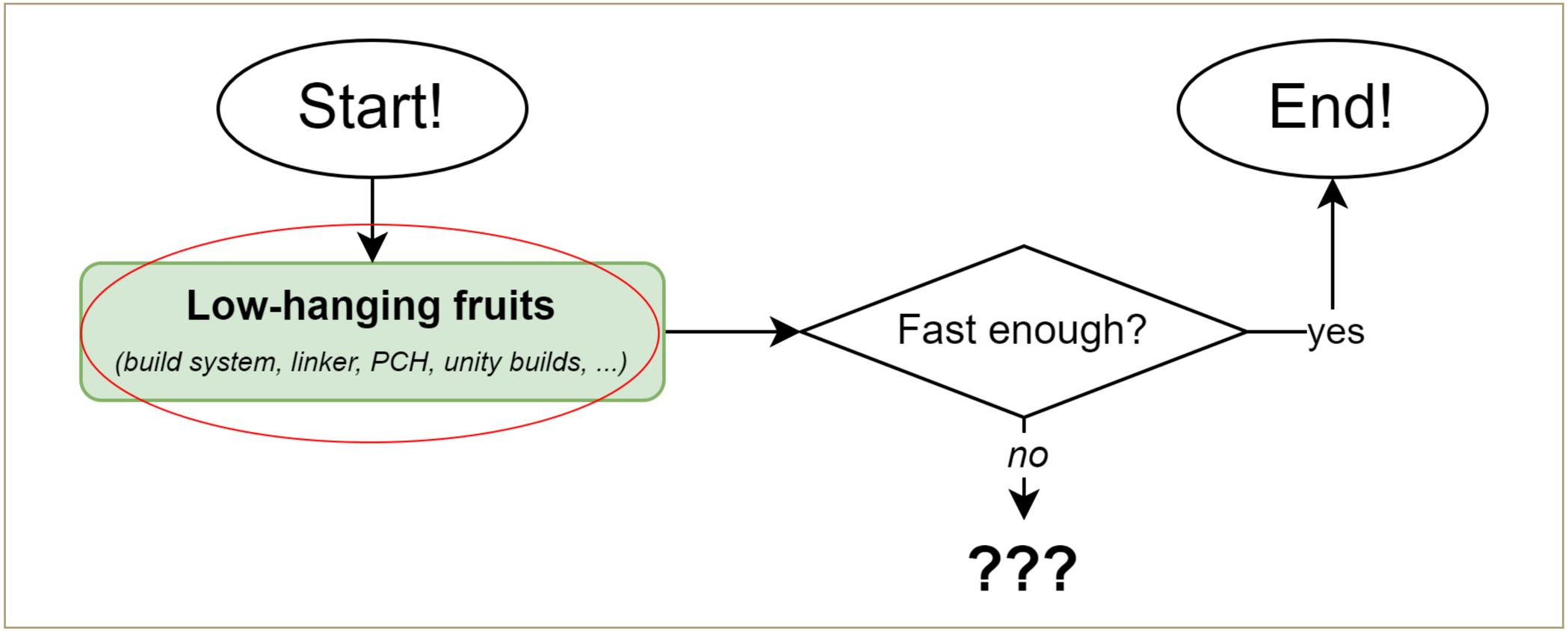


Building the flowchart (0)



- Defining “fast enough” is subjective
- Possible metrics:
  - Time/cost analysis, including CI time and developer time
  - Frustration/motivation
  - Reputation/marketing for library developers

Building the flowchart (1)



- Low-hanging fruits
  - Coarse-grained: they affect the entire project
  - Generally easy to introduce in an existing project
  - Their impact can be quite significant

Low-hanging fruits – build system (0)

- Compared to `make`, alternatives such as `ninja` have:
  - Superior scheduling algorithms
  - Better dependency tracking [src]
  - More robust file change detection [src]

- `ninja` is available on all major platforms
  - Enabling it through CMake is trivial

```
cmake -GNinja
```

- Invoke `ninja` instead of `make -jX`
  - By default, `ninja` will use all your available cores

```
❯ ninja
[12/100] Building CXX object src/SFML/Window/CMakeFiles/sfml-window.dir/Win32/InputImpl.cpp.obj
...
```


Low-hanging fruits – build system (1)

- SFML
  - *"Simple and Fast Multimedia Library"*
  - Fairly small project: around ~225 source files

- Build environment
  - Intel Core i9-9900K @ 3.6GHz base, 5GHz turbo
  - Corsair C15 32GB DDR4 SDRAM @ 1500MHz, dual channel
  - Samsung SSD 970 EVO Plus 1TB NVMe drive
  - MSYS2/MinGW
  - `hyperfine` benchmarking tool

```
# Using `clang++`, `ccache`, and `make` -- full rebuild
Benchmark 1: mingw32-make clean && mingw32-make -j16
  Time (mean ± σ):      5.951 s ±  0.231 s    [User: 5.990 s, System: 16.748 s]
  Range (min … max):    5.787 s …  6.114 s    2 runs

# Using `clang++`, `ccache`, and `ninja` -- full rebuild
Benchmark 1: ninja clean && ninja
  Time (mean ± σ):     948.7 ms ±  29.2 ms    [User: 1569.1 ms, System: 4458.8 ms]
  Range (min … max):   928.1 ms … 969.3 ms    2 runs
```


Low-hanging fruits – linker (0)

- The default linker (`ld`) and `gold` are very slow compared to `lld`
  - Inferior threading model, allocation scheme, and data structure choices [src]

Performance

This is a link time comparison on a 2-socket 20-core 40-thread Xeon E5-2680 2.80 GHz machine with an SSD drive. We ran gold and lld with or without multi-threading support. To disable multi-threading, we added `-no-threads` to the command lines.

Program	Output size	GNU ld	GNU gold w/o threads	GNU gold w/threads	lld w/o threads	lld w/threads
ffmpeg dbg	92 MiB	1.72s	1.16s	1.01s	0.60s	0.35s
mysqld dbg	154 MiB	8.50s	2.96s	2.68s	1.06s	0.68s
clang dbg	1.67 GiB	104.03s	34.18s	23.49s	14.82s	5.28s
chromium dbg	1.14 GiB	209.05s [1]	64.70s	60.82s	27.60s	16.70s

[llvm.org -- LLD - The LLVM Linker]

Low-hanging fruits – linker (1)

- `lld` is a drop-in replacement for `ld`
  - Easiest way to enable it is to pass `-fuse-ld=lld` to the compiler

```
cmake -GNinja -DCMAKE_CXX_FLAGS="-fuse-ld=lld"
```

- Roughly ~3s (~20%) speedup on full *unity* SFML 3.x rebuild with `clang++`
- However, even `lld` is slow when compared to `mold` …

Low-hanging fruits – linker (2)

- `mold` is a "modern" open-source linker
  - by Rui Ueyama, the same author who started the development of `lld`

- Extremely fast due to high parallelization & good choice of algorithms/data structures
  - In-depth technical comparsion versus `lld` available here

Program (linker output size)	**GNU gold**	**LLVM lld**	**mold** [src]
Chrome 96 (1.89 GiB)	53.86s	11.74s	2.21s
Clang 13 (3.18 GiB)	64.12s	5.82s	2.90s
Firefox 89 libxul (1.64 GiB)	32.95s	6.80s	1.42s

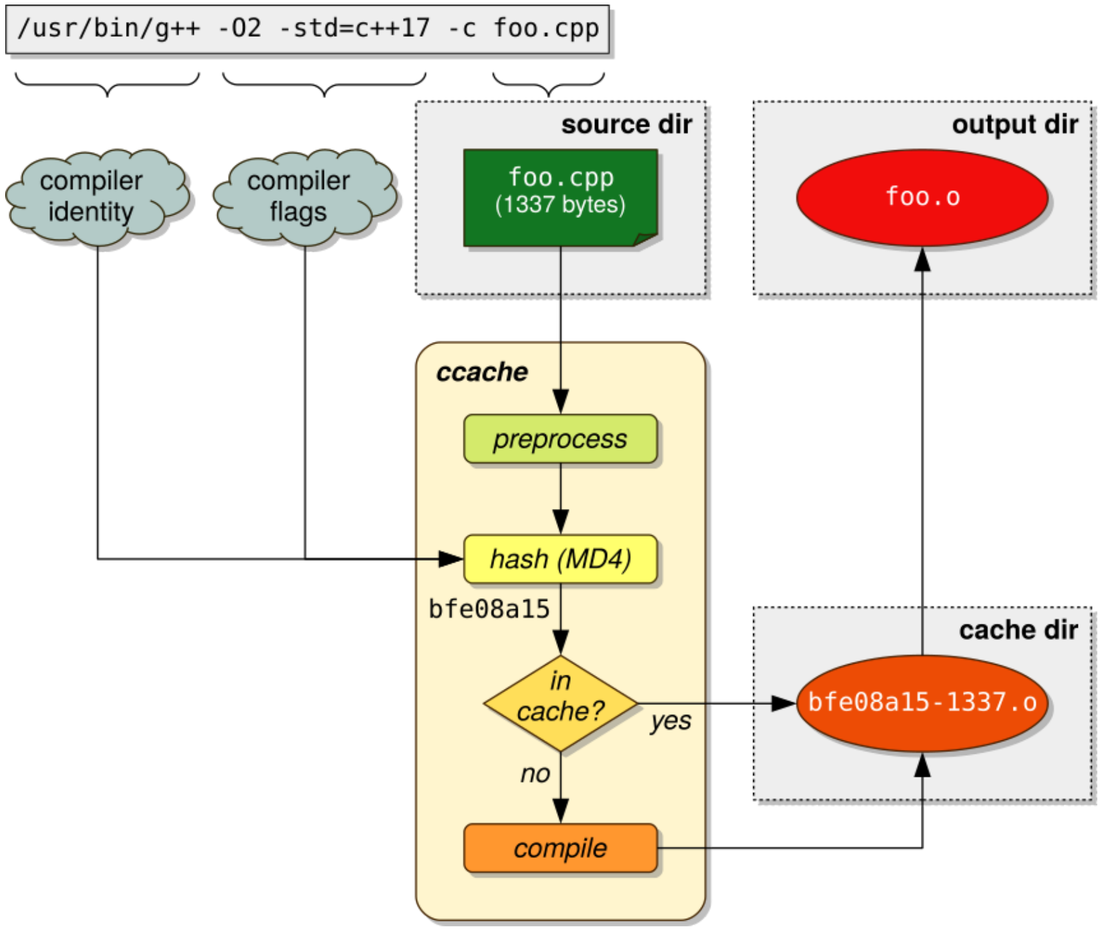
- `mold` only targets UNIX-like platforms under the AGPL license

- `sold`, a commercial version of `mold` supports MacOS and *will* support Windows
  - Licensing: paid on a per-user, per-month/year basis [src]

Low-hanging fruits – compilation cache (0)

- Avoid recompiling unchanged source files
  - Even in fresh new builds
  - Map `(compiler, flags, file_hash)` ⇒ `.obj`
- Common tools: `ccache`, `sccache`
  - Others: FASTBuild, IncrediBuild
- `ccache.c` benchmark [src]

	Elapsed time	Percent	Factor
Without ccache	0.6988 s	100.00 %	1.00 x
ccache 3.7.1 prepr., first time	0.7251 s	103.77 %	0.96 x
ccache 3.7.1 prepr., second time	0.0247 s	3.53 %	28.33 x
ccache 3.7.1 direct, first time	0.7268 s	104.01 %	0.96 x
ccache 3.7.1 direct, second time	0.0048 s	0.69 %	145.39 x
ccache 3.7.1 depend, first time	0.7102 s	101.64 %	0.98 x
ccache 3.7.1 depend, second time	0.0051 s	0.73 %	137.81 x



[Von Christoph Erhardt -- The C/C++ Developer's Guide to Avoiding Office Swordfights]

Low-hanging fruits – compilation cache (1)

- Enabling `ccache` in CMake is trivial
  - SFML PR available here

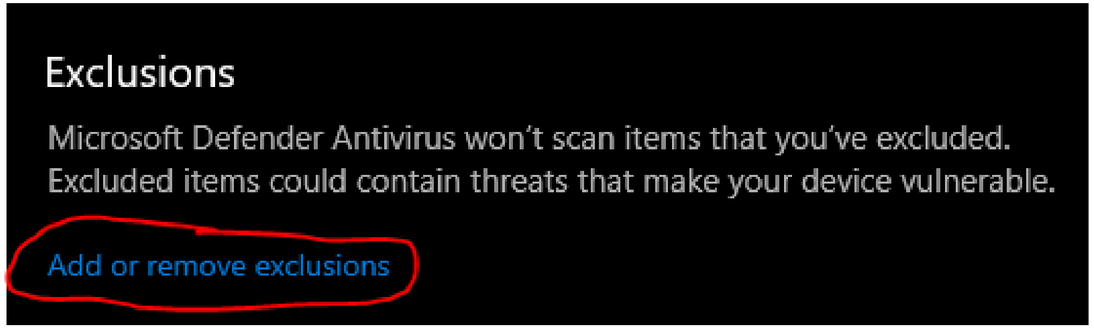
```
# use ccache if available
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
    message(STATUS "Found ccache in ${CCACHE_PROGRAM}")
    set(CMAKE_CXX_COMPILER_LAUNCHER "${CCACHE_PROGRAM}")
endif()
```

- `CMAKE_CXX_COMPILER_LAUNCHER` prepends `ccache` to compiler invocations
- Roughly ~35-50x speedup for SFML 3.x rebuild!
  - Great when working with multiple builds at the same time
- Other notes about `ccache`:
  - Supports HTTP and Redis storage backends out of the box [src]
  - Supports precompiled headers [src]

Low-hanging fruits – build machine configuration

- Platform-specific
  - For Linux, check the relevant Arch Linux wiki page
  - For Windows, look for “gaming optimizations” or specific tools (e.g., Optimizer)

- Building locally on Windows?
  - Boost your compilation times by 10% [src] with this **one easy trick**!



- Add your build directories as Windows Defender exclusions
  - Windows Defender can use up to ~20% CPU during compilation
  - Find offending directories with `procmon`

- Generally a good idea to look at processes during compilation

Low-hanging fruits – precompiled headers (0)

- The same header `x.hpp` is usually processed anew in every source files that uses it
  - Very wasteful work: preprocessing, tokenizing, parsing, etc...
- Compilers can preprocess some commonly used headers of our choice
  - They are translated to some intermediate representation *once*
  - That IR is then prepended to every compiled source file
- Time savings can be massive!
- SFML results: [PR #1895] [PR #2488]
  - 34.032s - *baseline*
  - 28.787s - *PCH (without reuse – 5 targets, 5 PCHs)*
  - 19.381s - *PCH (with reuse – 5 targets, 1 PCH)*
- "Reuse": using the same PCH for multiple targets built in the same project
  - A bit of extra work required to set it up, but usually worth it

Low-hanging fruits – precompiled headers (1)

- Using PCH with CMake:
  - 1. Create `PCH.hpp` file with commonly used includes
  - 2. Use `target_precompile_headers(<target> PRIVATE "PCH.hpp")`

- For reuse:
  - 1. Pick (or create) a target that all other targets depend from
  - 2. `target_precompile_headers(<base_target> PRIVATE "PCH.hpp")`
  - 3. `target_precompile_headers(<other_target> REUSE_FROM <base_target>)`

- "Reuse" caveats:
  - Compiler flags have to *perfectly* match, or PCHs will be ignored
  - Sometimes this means having to pass flags/defines nonsensical for a target
  - Or ugly hacks, such as manually renaming PDB files [src]

- Good idea: make PCHs toggleable through a flag

```
option(SFML_ENABLE_PCH FALSE BOOL
       "TRUE to enable precompiled headers for SFML builds")
```


Low-hanging fruits – precompiled headers (2)

- What to put in `PCH.hpp`?

```cpp
// PCH.hpp
#pragma once

// Commonly-used first-party headers (e.g., logging, assertions, basic components)
#include <SFML/System/Err.hpp>
#include <SFML/System/String.hpp>
#include <SFML/System/Time.hpp>
#include <SFML/System/Vector2.hpp>

// Expensive headers, like `windows.h`
#ifdef SFML_SYSTEM_WINDOWS
#include <SFML/System/Win32/WindowsHeader.hpp>
#endif

// Commonly used Standard Library or third-party headers
#include <algorithm>
#include <filesystem>
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>
#include <vector>
/* ... */
```


Low-hanging fruits – precompiled headers (3)

- When to use PCHs?
  - They scale well with a lot of source files and frequently used expensive headers
  - Any change to `PCH.hpp` requires a full recompilation
    - Avoid putting in headers that might change
  - PCHs growing too large might hit diminishing returns
    - Remember that the PCH IR gets included in *every* source file

- How to determine what headers are used frequently?
  - `grep`, `sort`, and `wc` worked well for me
  - Tools such as `include-what-you-use` work well – good CMake integration

- How to determine what headers are expensive?
  - `ClangBuildAnalyzer` – we will cover it soon

- Header hygiene
  - Do not remove existing headers from source files! *(i.e. don't rely on PCHs for correctness)*
  - Build with PCHs disabled from time to time (or have a CI job) to catch missing includes

Low-hanging fruits – unity builds (0)

- Coalesce multiple source files into fewer larger ones
  - Imagine a sort of automatic `#include`, but for `.cpp` files

```
// unity_0_cxx.cxx (generated by CMake)

#include "my_source0.cpp"
#include "my_source1.cpp"
#include "my_source2.cpp"
/* ... */
```

- Surprisingly large amount of benefits!
  - Commonly included headers are parsed/compiled fewer times
  - Fewer redundant template instantiations
  - Much less work for the linker *(e.g., less symbol de-duplication and stiching)*
    - Incremental builds might actually be *faster* because of that
  - Fewer invocations of the compiler and creation of `.obj` files
  - More optimization opportunities – the wish.com version of LTO
  - Catch ODR violations at compile-time *(example later)*
  - Enforce header hygiene best practices such as `#pragma once` *(example later)*

Low-hanging fruits – unity builds (1)

- Enabling unity builds in CMake:

```
cmake -GNinja -DCMAKE_UNITY_BUILD=ON -DCMAKE_CXX_FLAGS="-fuse-ld=lld"
```

- Enabling on a per-target basis is also easy:

```
set_target_properties(<target> PROPERTIES UNITY_BUILD ON)
```

- Can also skip a problematic file via `SKIP_UNITY_BUILD_INCLUSION`

- Various knobs to tweak and related utilities:
  - `set(UNITY_BUILD_BATCH_SIZE <n_sources>)`
  - `set(UNITY_BUILD_CODE_BEFORE_INCLUDE <code>)`
  - `set(UNITY_BUILD_CODE_AFTER_INCLUDE <code>)`

- Recommendations:
  - Enable file-by-file with small batch size, fix any arising issue
  - Keep a non-unity CI build to catch issues *(e.g., missing header)*

Low-hanging fruits – unity builds (2)

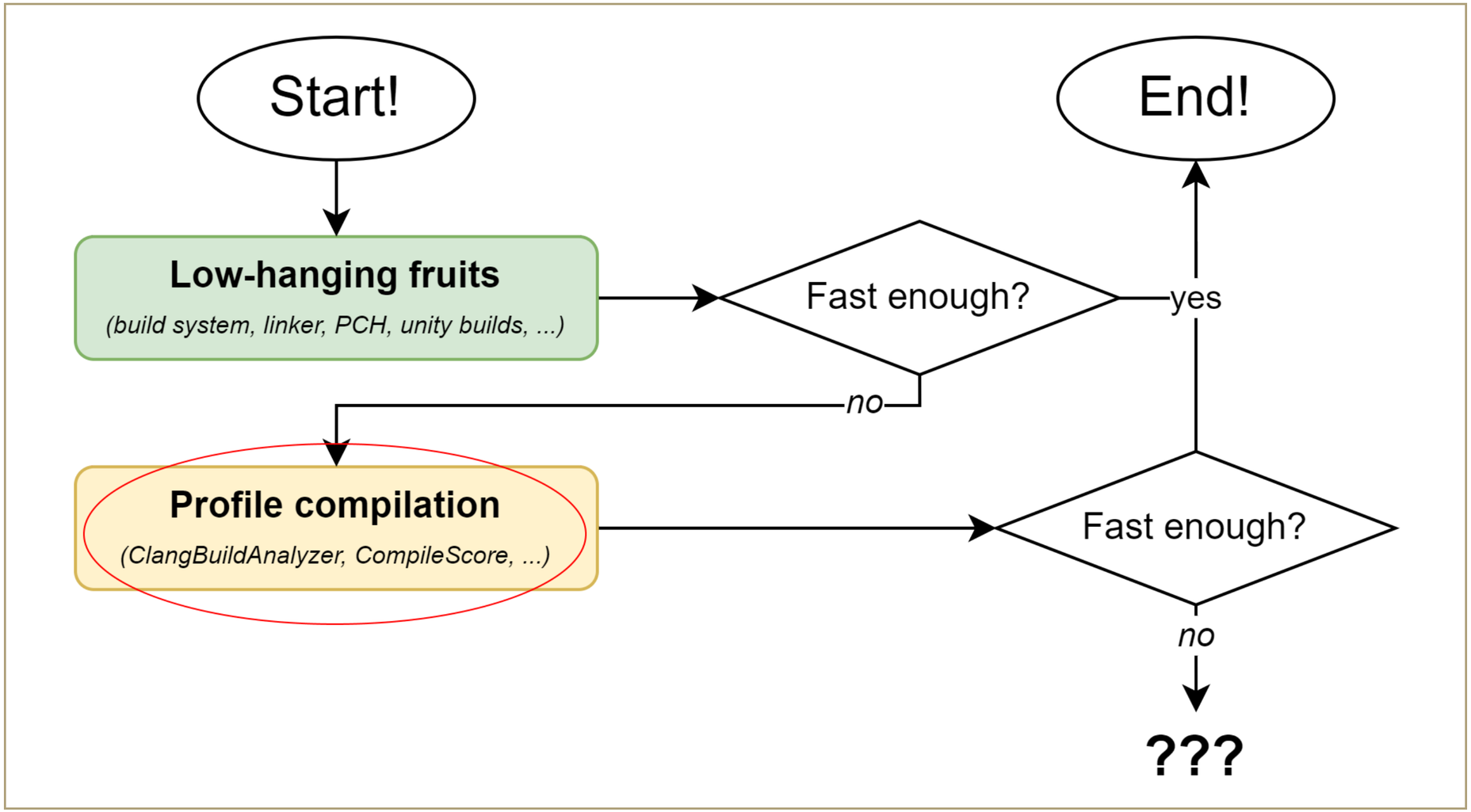
- Many possible drawbacks:
  - Symbols having internal linkage and same name will clash *(example later)*
  - Source-scope `using namespace` can cause collisions **(*)**
  - Preprocessor defines in `.cpp` files can clash – must use `#undef` **(*)**
  - Smaller one-source changes can slow down iteration times
    - Some files can be excluded from the unity build (or it can be disabled during development)
  - Loss of parallelism if the choice of N for `UNITY_BUILD_BATCH_SIZE` is poor
  - Generally higher memory usage
  - Possible behavior change *(e.g., introducing an overload that is a better match)* **(*)**
  - Can't always apply to third-party libraries
    - And maintainers might not be willing to make changes to support unity builds...

- Arguably, drawbacks marked with **(*)** promote good practices

- Roughly ~3-4s (~29%) speedup on SFML 3.x full rebuild *(with `clang++` and `lld`)*
  - SFML PR here

- Also massive ~3x speedup on our `clang-tidy` CI job 🤦
  - SFML PR here *(by Chris Thrasher)*

Building the flowchart (2)



- How can we profile compilation?

Profiling compilation – `ClangBuildAnalyzer` (0)

- Free & open-source tool developed by Aras Pranckevičius
  - Parses Clang's `-ftime-trace` output and produces a human-friendly report
  - The report provides *actionable* information

- `-ftime-trace`
  - Developed by Aras himself, merged upstream since Clang 9 [src]
  - Produces Chrome Tracing `.json` files for each compiled object file
  - No equivalent in GCC or MSVC

- How to use
  - Use `clang++` as your compiler, passing `-ftime-trace` to your compiler flags
  - Compile everything you want to profile
  - Run `ClangBuildAnalyzer` in the build directory

```
cmake -GNinja -DCMAKE_UNITY_BUILD=ON -DCMAKE_CXX_COMPILER=clang++
      -DCMAKE_CXX_FLAGS="-fuse-ld=lld -ftime-trace"

./ClangBuildAnalyzer.exe --all . analysis.bin
./ClangBuildAnalyzer.exe --analyze analysis.bin > analysis.txt && explorer analysis.txt
```


Profiling compilation – ClangBuildAnalyzer (1)

```
Analyzing build trace from 'analysis.bin'...
**** Time summary:
Compilation (171 times):
  Parsing (frontend):           128.9 s
  Codegen & opts (backend):      29.9 s

**** Files that took longest to parse (compiler frontend):
  3320 ms: ./src/SFML/Window/CMakeFiles/sfml-window.dir/WindowImpl.cpp.obj
  3239 ms: ./src/SFML/Window/CMakeFiles/sfml-window.dir/Win32/JoystickImpl.cpp.obj
  2912 ms: ./src/SFML/Window/CMakeFiles/sfml-window.dir/Win32/WindowImplWin32.cpp.obj
  2826 ms: ./src/SFML/Window/CMakeFiles/sfml-window.dir/GlContext.cpp.obj
  2710 ms: ./src/SFML/Window/CMakeFiles/sfml-window.dir/Win32/WglContext.cpp.obj
  2458 ms: ./src/SFML/Graphics/CMakeFiles/sfml-graphics.dir/Font.cpp.obj
  ...

**** Files that took longest to codegen (compiler backend):
  5623 ms: ./src/SFML/Graphics/CMakeFiles/sfml-graphics.dir/ImageLoader.cpp.obj
  2562 ms: ./src/SFML/Graphics/CMakeFiles/sfml-graphics.dir/GLExtensions.cpp.obj
  1356 ms: ./src/SFML/Graphics/CMakeFiles/sfml-graphics.dir/Font.cpp.obj
  1326 ms: ./src/SFML/Audio/CMakeFiles/sfml-audio.dir/SoundFileReaderMp3.cpp.obj
  1197 ms: ./src/SFML/Graphics/CMakeFiles/sfml-graphics.dir/Shader.cpp.obj
  1032 ms: ./src/SFML/Network/CMakeFiles/sfml-network.dir/Ftp.cpp.obj
  ...
```


Profiling compilation – ClangBuildAnalyzer (2)

```
**** Templates that took longest to instantiate:
   693 ms: std::unique_ptr<std::filesystem::path::_List::_Impl, std:... (34 times, avg 20 ms)
   582 ms: std::__uniq_ptr_data<std::filesystem::path::_List::_Impl,... (34 times, avg 17 ms)
   576 ms: std::__uniq_ptr_impl<std::filesystem::path::_List::_Impl,... (34 times, avg 16 ms)
   495 ms: std::basic_string<char> (65 times, avg 7 ms)
   458 ms: __gnu_cxx::__to_xstring<std::basic_string<wchar_t>, wchar_t> (65 times, avg 7 ms)
   428 ms: std::basic_string<char32_t> (65 times, avg 6 ms)
   424 ms: std::basic_string<char16_t> (65 times, avg 6 ms)
   380 ms: std::basic_string<wchar_t> (65 times, avg 5 ms)
   368 ms: std::basic_string<char16_t>::basic_string (85 times, avg 4 ms)
   363 ms: std::basic_string<char32_t>::basic_string (85 times, avg 4 ms)
   353 ms: std::filesystem::path::string<char32_t, std::char_traits<... (34 times, avg 10 ms)
   ...

**** Template sets that took longest to instantiate:
  3393 ms: std::__and_<$> (2185 times, avg 1 ms)
  2332 ms: std::unique_ptr<$> (109 times, avg 21 ms)
  1893 ms: std::__uniq_ptr_data<$> (109 times, avg 17 ms)
  1874 ms: std::__uniq_ptr_impl<$> (109 times, avg 17 ms)
  1735 ms: std::basic_string<$> (261 times, avg 6 ms)
  1430 ms: std::is_convertible<$> (1555 times, avg 0 ms)
  1096 ms: std::chrono::duration<$> (632 times, avg 1 ms)
  1028 ms: std::basic_string<$>::basic_string (414 times, avg 2 ms)
   998 ms: std::basic_string<$>::_M_construct<$> (394 times, avg 2 ms)
   ...
```


Profiling compilation – ClangBuildAnalyzer (3)

```
**** Functions that took longest to compile:
   489 ms: gladLoadGLUserPtr(void (* (*)(void*, char const*))(), void*) cpp)
   431 ms: sf_glad_gl_find_extensions_gl(int)
   156 ms: mp3dec_decode_frame
   111 ms: stbi__create_png_image_raw(stbi__png*, unsigned char*, unsigned int,...
    86 ms: stbi__load_main(stbi__context*, int*, int*, int*, int, stbi__result_...
    78 ms: sf::Ftp::getResponse()
    72 ms: stbi__jpeg_load(stbi__context*, int*, int*, int*, int, stbi__result_...
    60 ms: sf::priv::WglContext::createContext(sf::priv::WglContext*) cpp)
    60 ms: stbi__do_zlib(stbi__zbuf*, char*, int, int, int)
    57 ms: sf::priv::JoystickImpl::openDInput(unsigned int)
    ...

**** Function sets that took longest to compile / optimize:
   107 ms: std::vector<$>::_M_default_append(unsigned long long) (11 times, avg 9 ms)
    91 ms: std::basic_ostream<char, std::char_traits<char> >& std::__de... (10 times, avg 9 ms)
    69 ms: bool std::__do_str_codecvt<$>(wchar_t const*, wchar_t const*... (10 times, avg 6 ms)
    56 ms: std::_Hashtable<$>::_M_rehash_aux(unsigned long long, std::i... (9 times, avg 6 ms)
    52 ms: std::_Hashtable<$>::_M_insert_unique_node(unsigned long long... (9 times, avg 5 ms)
    43 ms: std::__cxx11::basic_string<$> std::filesystem::__cxx11::path... (10 times, avg 4 ms)
    37 ms: void std::_Hashtable<$>::_M_assign<$>(std::_Hashtable<$> con... (3 times, avg 12 ms)
    29 ms: sf::priv::RenderTextureImplFBO::create(sf::Vector2<$> const&... (1 times, avg 29 ms)
    28 ms: sf::Ftp::upload(std::__cxx11::basic_string<$> const&, std::_... (1 times, avg 28 ms)
    ...
```

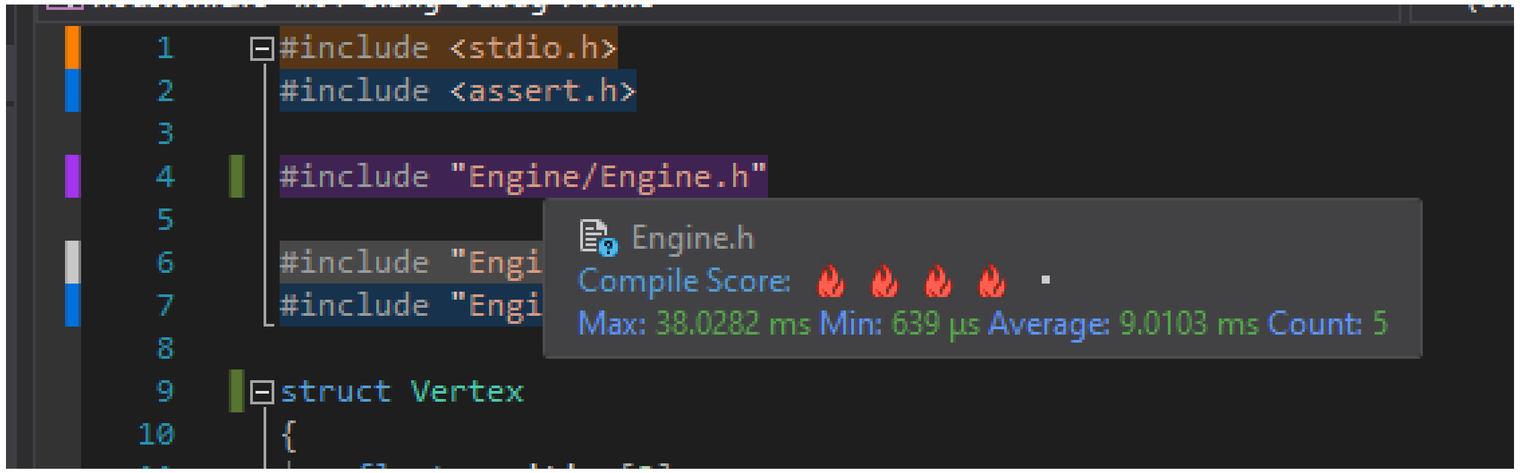

Profiling compilation – ClangBuildAnalyzer (4)

```
**** Expensive headers:
26760 ms: SFML/System/Win32/WindowsHeader.hpp (included 22 times, avg 1216 ms), included via:
  UdpSocket.cpp.obj SocketImpl.hpp SocketImpl.hpp  (1321 ms)
  Packet.cpp.obj SocketImpl.hpp SocketImpl.hpp  (1296 ms)
  IpAddress.cpp.obj SocketImpl.hpp SocketImpl.hpp  (1294 ms)
  VideoModeImpl.cpp.obj  (1292 ms)
  Joystick.cpp.obj JoystickManager.hpp JoystickImpl.hpp JoystickImpl.hpp  (1291 ms)
  ...

10748 ms: SFML/Network/Win32/SocketImpl.hpp (included 8 times, avg 1343 ms), included via:
  SocketImpl.cpp.obj  (1466 ms)
  TcpListener.cpp.obj SocketImpl.hpp  (1456 ms)
  UdpSocket.cpp.obj SocketImpl.hpp  (1377 ms)
  Packet.cpp.obj SocketImpl.hpp  (1366 ms)
  IpAddress.cpp.obj SocketImpl.hpp  (1323 ms)
  ...

10713 ms: SFML/Graphics/GLCheck.hpp (included 10 times, avg 1071 ms), included via:
  RenderTextureImplFBO.cpp.obj  (1262 ms)
  RenderTarget.cpp.obj  (1218 ms)
  TextureSaver.cpp.obj TextureSaver.hpp  (1210 ms)
  GLCheck.cpp.obj  (1209 ms)
  Texture.cpp.obj  (1182 ms)
  ...
```


Profiling compilation – `CompileScore` and VCPerf

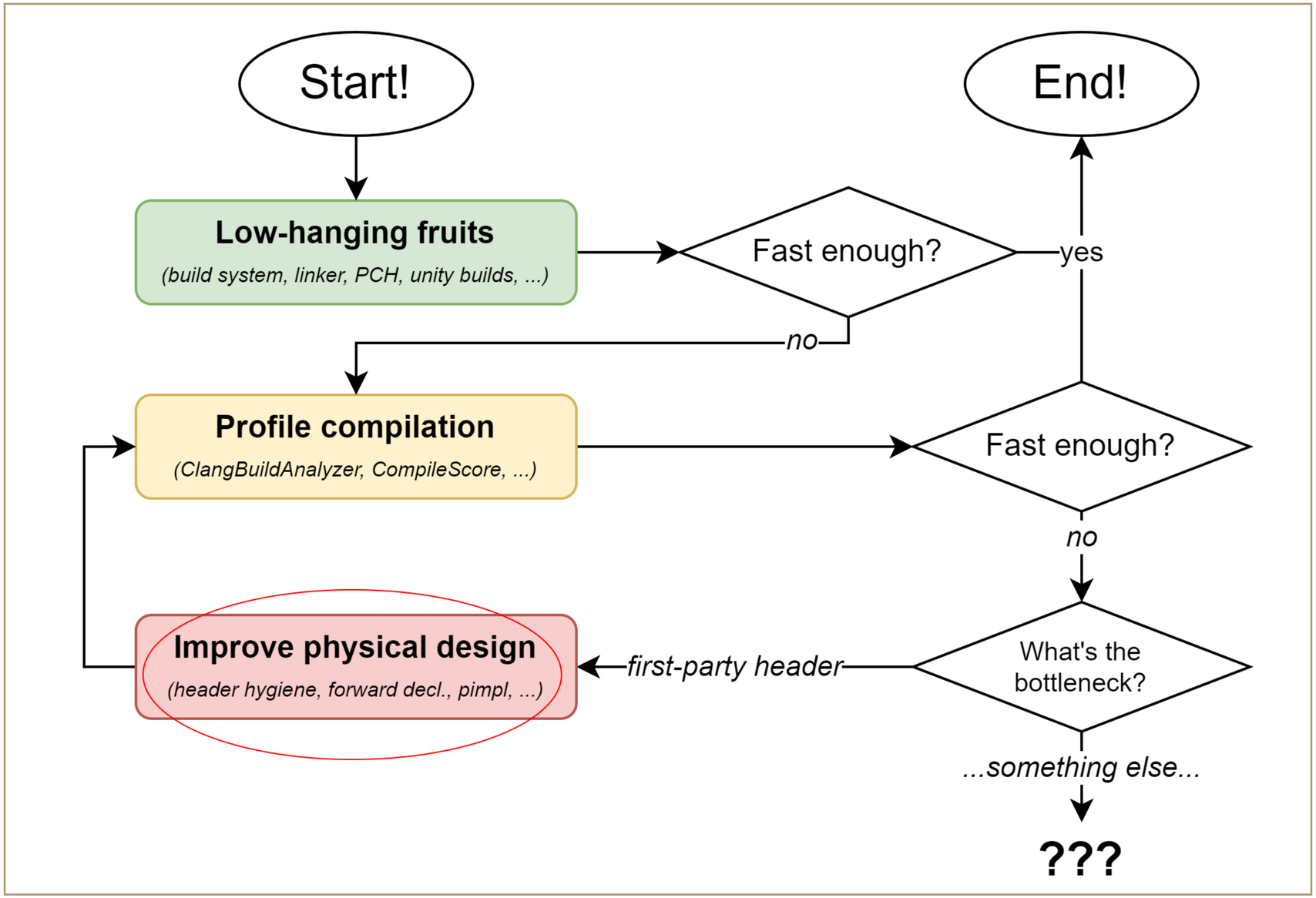
- Visual Studio extension based on `clang-cl's -ftime-trace`
  - Developed by Ramon Viladomat

- Many features:
  - Profile compilation
  - Text highlights on include costs
  - Complation flamegraph
  - Include graph



- VCPerf (MSVC) also has a `/timetrace` flag

Building the flowchart (3)

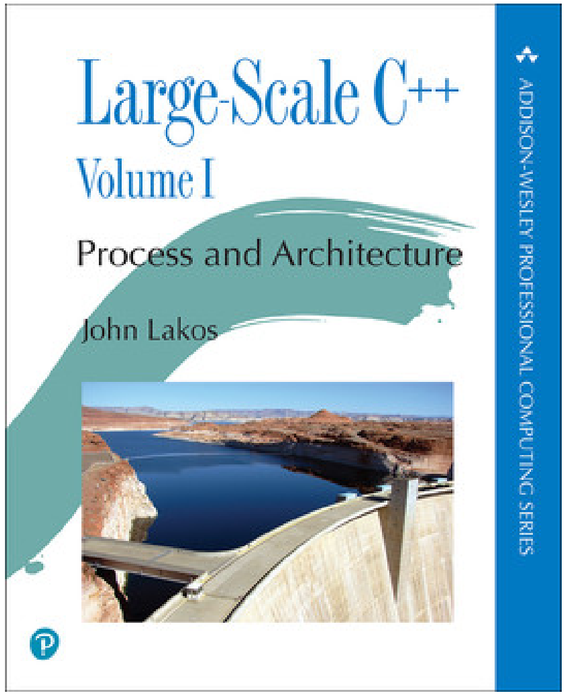


Improving physical design – components and levelization

- Split your software into physically self-contained components
  - Component: header + source *(along with an associated standalone test driver)*
  - Limit dependencies among components *(no cyclic physical dependencies)*
    - Lakos20 (Sec. 3.5) provides 9 levelization techniques for removing design cycles

- Levelization
  - Components reside in *levels* due to their relative physical dependencies
  - (Local) leaf components are defined to be at level 1
  - Components at level N depend on components at levels $N-1$ (and lower)
  - With proper testing $\rightarrow$ new components need depend on only tested ones
  - Group components together into packages *(and packages into groups)*

- John Lakos' work on physical design is a fantastic reference
  - Amazon: Large-Scale C++ Software Design
  - YouTube: "Advanced Levelization Techniques" - John Lakos [CppCon 2016]
  - YouTube: "C++ Modules and Large-Scale Development" - John Lakos [ACCU 2019]
  - YouTube : "Lakos'20: The "Dam" Book is Done!" - John Lakos [ACCU 2021]



Improving physical design – declarations and definitions

- Basic advice, but always good to be reminded of it
- Always put the definition of a function in a `.cpp` file
  - Unless you intentionally want to `inline` the function

```h
// component.h
#pragma once
class component { void f(); };
```

```cpp
// component.cpp
#include <component.h>
void component::f() { /* ... */ }
```

- Did you know you can also put `= default` in the source file?

```h
// example.h
#pragma once
class example { example(); };
```

```cpp
// example.cpp
#include <example.h>
example::example() = default;
```

- For more information, check out Embracing Modern C++ Safely
  - Section 1.1. “Defaulted Functions”

Improving physical design – forward declarations (0)

- For first-party types, prefer *forward declarations* rather than headers
  - They can be used in more places than you might expect! [src]

```
// zoo.h
#pragma once

class animal;

class zoo
{
public:
    animal get_random_animal();
    void add_animal(animal);

private:
    std::vector<animal> _animals;
};
```

```
// zoo.cpp
#include "zoo.h"
#include "animal.h"

animal zoo::get_random_animal() { /* ... */ }

void zoo::add_animal(animal x)
{
    _animals.push_back(x);
}
```

- **Q:** is the code above valid?

```
ERROR: 'std::is_complete_or_unbounded(std::type_identity<animal>{})':
       template argument must be a complete class or an unbounded array
```


Improving physical design – forward declarations (1)

```h
// zoo.h
#pragma once

class animal;

class zoo
{
public:
    animal get_random_animal();
    void add_animal(animal);

    zoo();  // <==
    ~zoo(); // <==

private:
    std::vector<animal> _animals;
};
```

```cpp
// zoo.cpp
#include "zoo.h"
#include "animal.h"

animal zoo::get_random_animal() { /* ... */ }

void zoo::add_animal(animal x)
{
    _animals.push_back(x);
}

zoo() = default;  // <==
~zoo() = default; // <==
```

- The code above is now completely valid!
  - `animal.h` is only included in `zoo.cpp`
- Using forward declarations can greatly reduce header dependencies
  - Libraries should provide *"forward headers"* for their users

Improving physical design – forward declarations (2)

- The same trick can be used for `std::unique_ptr`
  - Allows safe memory management of forward-declared polymorphic types

```
// subsystem.h
#pragma once

class dependency;

class subsystem
{
public:
    subsystem(std::unique_ptr<dependency>&&); // defined or defaulted in `.cpp`
    ~subsystem();                             //    "    "       "     "    "

private:
    std::unique_ptr<dependency> _dependency;
};
```

- Careful!
  - The user of `subsystem.h` will often have to include `dependency.h` themselves
    - Error messages might be unintuitive, like on the previous slides
    - Problematic for a public-facing header, but possibly worthwhile for an implementation one

Improving physical design – PImpl (0)

- Common technique, still very useful
  - Completely isolates implementation in a `.cpp` file
  - Can use a forward declaration + `std::unique_ptr` 😳

```cpp
// steam_manager.hpp
#pragma once
#include <memory>

class steam_manager
{
private:
    struct impl;
    std::unique_ptr<impl> _impl;

public:
    steam_manager();
    ~steam_manager();

    void api0();
    void api1();
};
```

- Example from Open Hexagon

```cpp
// steam_manager.cpp
#include "steam_manager.hpp"
#include "steam_api.hpp" // possibly expensive
#include "windows.h"     // very expensive

struct steam_manager::impl
{
    void impl0() { /* ... */ }
    void impl1() { /* ... */ }
};

steam_manager::steam_manager()
    : _impl(std::make_unique<impl>()) { }

steam_manager::~steam_manager() = default;
    // ^ must be in the source file!

void steam_manager::api0() { _impl->impl0(); }
void steam_manager::api1() { _impl->impl1(); }
```


Improving physical design – PImpl (1)

- When to use?
  - PImpl should be a default for components used infrequently or outside of the hot path
  - Don’t use for anything that must be cache-friendly or invoked in a hot loop

- Drawbacks:
  - SO. MUCH. BOILERPLATE.
  - Cost of pointer indirection *(suprisingly avoidable)*

- Avoiding indirection:
  - Aligned array of `std::byte` as a buffer with a max size instead of `std::unique_ptr`
  - `static_assert` that `impl` fits in the buffer in the `.cpp`
  - Create the `impl` via placement-`new` in the buffer
  - Only do this if you can prove that you needed to via profiling

Improving physical design – header hygiene

- Good practices for header hygiene:
  - Do not rely on transitive inclusion
  - Do not rely on header inclusion order
  - Group headers together logically
  - Sort header groups by dependency levels
  - Sort headers in groups alphabetically
  - Include fine-grained headers, not catch-all ones [SFML PR #2489]

```
#include <SFML/Window/Event.hpp> // First-party headers (level 1)
#include <SFML/Window/Keyboard.hpp>

#include <SFML/System/Clock.hpp> // First-party headers (level 2 -- Window depends on System)
#include <SFML/System/Time.hpp>

#include <boost/stacktrace.hpp> // Third-party headers
#include <glad/gl.h>

#include <filesystem> // C++ Standard Library
#include <vector>

#include <cstdlib> // C Standard Library
#include <cstdio>
```

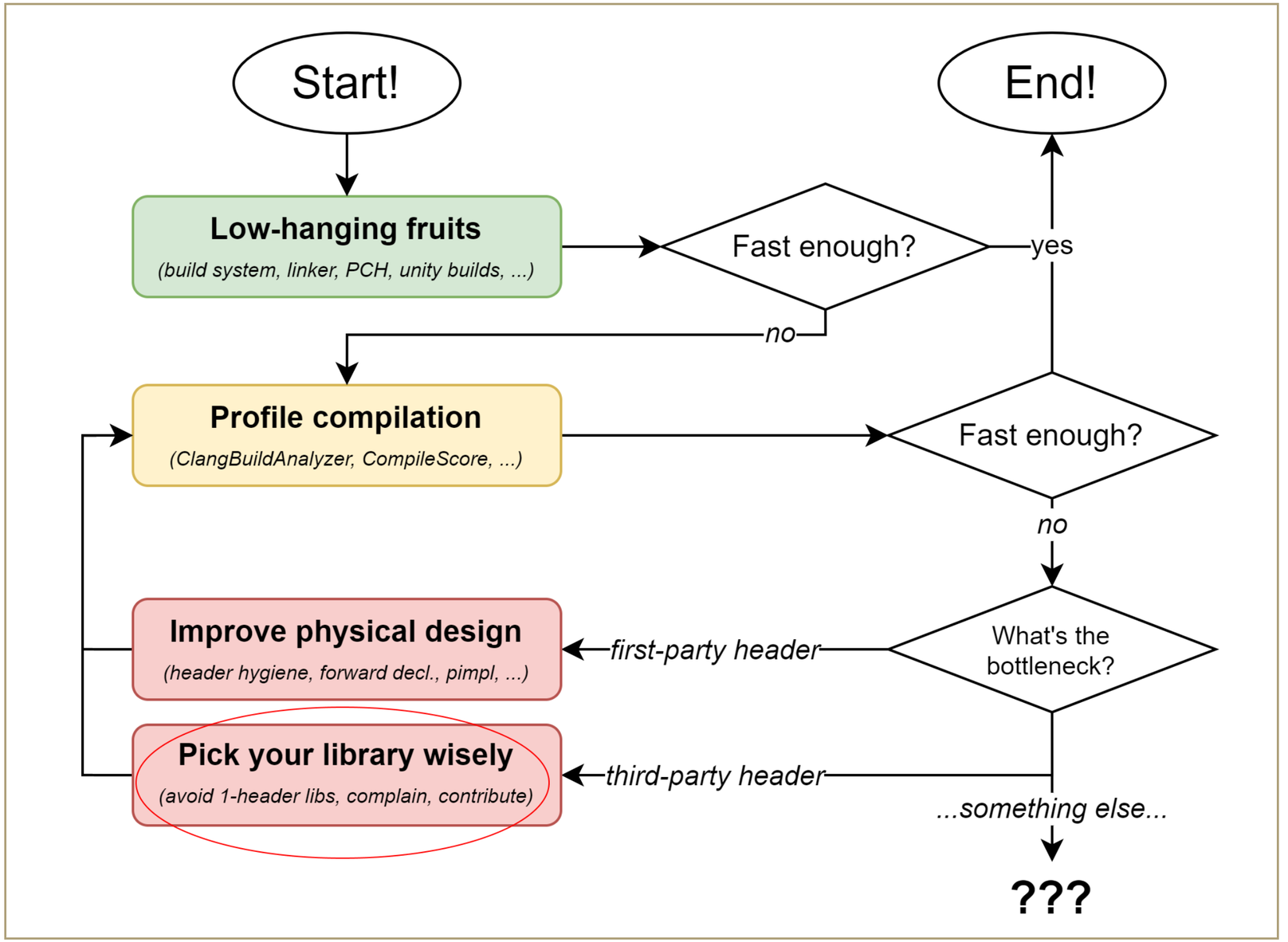

Improving physical design – unnecessary includes

- How to figure out what includes are unnecessary?
  - `include-what-you-use` – `clang`-based tool [SFML PR #1917] [SFML PR #2002] [SFML PR #2013] [SFML PR #2021] [SFML PR #2425]
- Native support in CMake
  - Point `CMAKE_CXX_INCLUDE_WHAT_YOU_USE` to the IWYU executable
  - Can also just use compilation database via `CMAKE_EXPORT_COMPILE_COMMANDS`

```
cmake -GNinja -DCMAKE_UNITY_BUILD=OFF # remember to turn unity builds off
      -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_CXX_FLAGS="-fuse-ld=lld"
      -DCMAKE_CXX_INCLUDE_WHAT_YOU_USE=include-what-you-use
```

```
[3/100] Building CXX object src/SFML/System/CMakeFiles/sfml-system.dir/Sleep.cpp.obj

C:/OHW/SFML/src/SFML/System/Sleep.cpp should add these lines:
#include "SFML/Config.hpp"                  // for SFML_SYSTEM_WINDOWS
#include "SFML/System/Time.inl"             // for operator>=, Time::Zero

C:/OHW/SFML/src/SFML/System/Sleep.cpp should remove these lines:

The full include-list for C:/OHW/SFML/src/SFML/System/Sleep.cpp:
#include <SFML/System/Sleep.hpp>
#include <SFML/System/Time.hpp>             // for Time
#include <SFML/System/Win32/SleepImpl.hpp>  // for sleepImpl
...
```

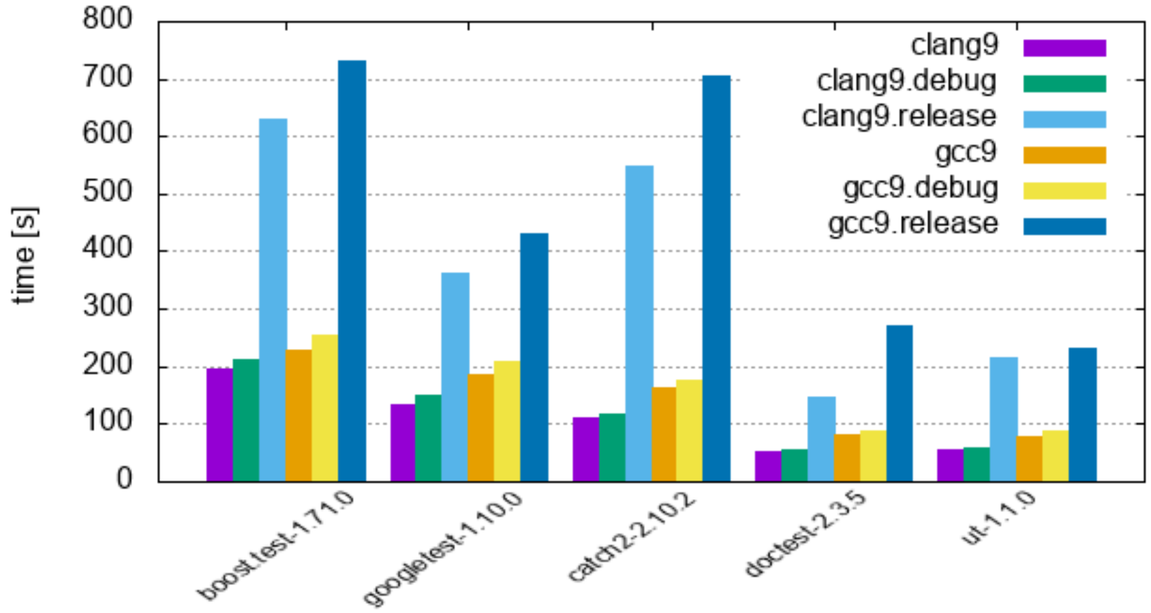

Building the flowchart (4)



3rd-party libraries – general tips

- Physical design considerations and PCH apply
  - Don't `#include` unnecessarily
  - Consider PImpl or isolating in `.cpp` *(e.g., `sqlite_orm` in Open Hexagon)*
  - Consider forward-declaring or check if the library provides a `fwd` header

- Pick your libraries carefully
  - Libraries fast to compile often advertise it 🙄 [SFML PR #1921]
    - e.g., `doctest` or `boost-ext.ut`

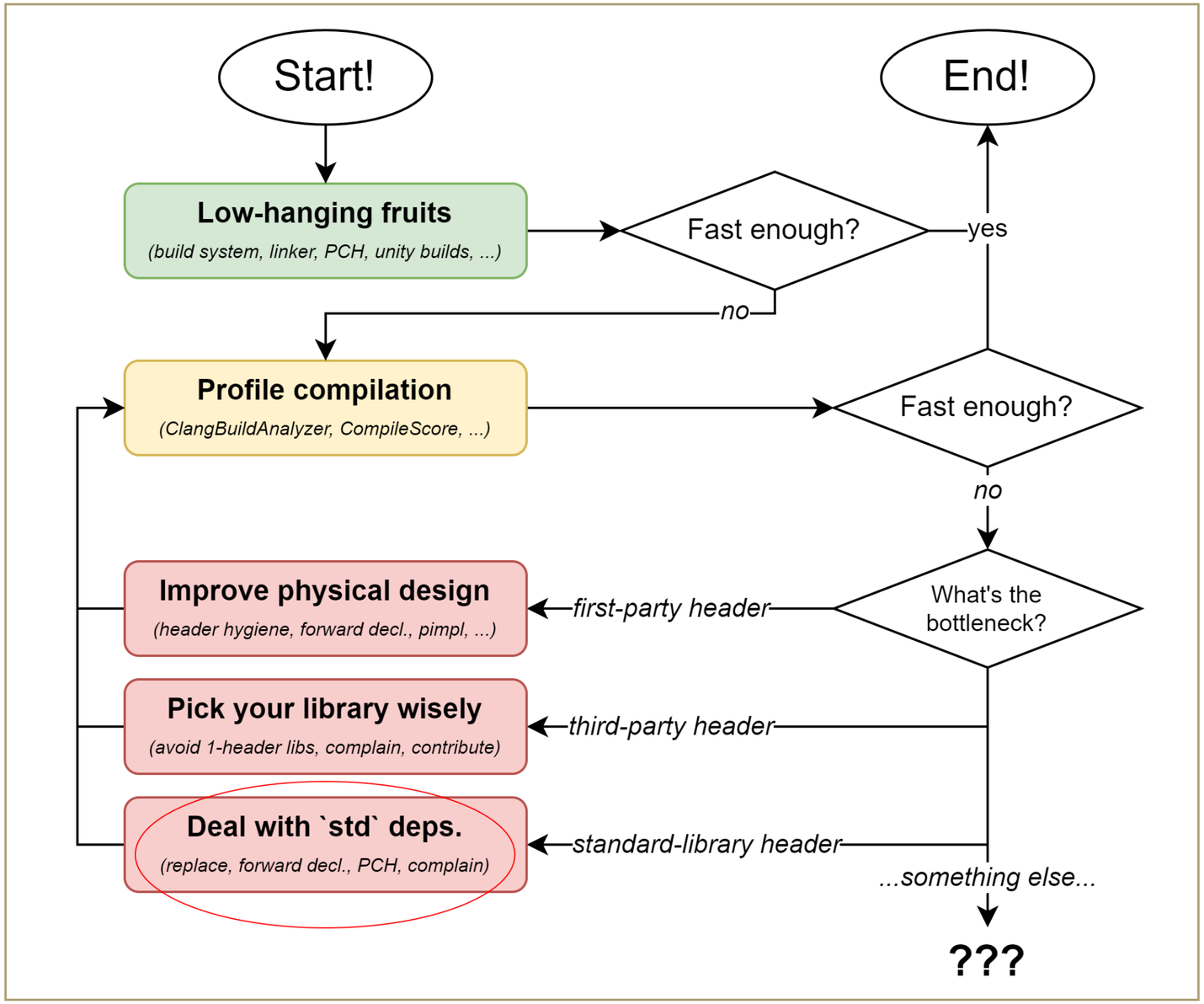


[boost.ut -- 10000 tests, 20000 asserts, 100 cpp files]

3rd-party libraries – complaining, contributing, competing

- If a library that you need doesnt care much about compilation times…
  - ~~Consider complaining~~
  - Considering politely requesting improvements
- Do you know what's better than a request…?
- **Call to action!**
  - If you learned something from this talk, contribute to an open-source project!
    - *Example:* SFML PRs
    - *Example:* R.E.L.I.V.E. project PRs
    - *Example:* `sqlite_orm` PR
- 🙋 🙋🏿 🙋 🙋🏽 🙋🏿 🙋🏽
- Alternatively, consider *competing*
  - Fast-to-compile lightweight libraries are sought after, especially in game development

Building the flowchart (5)



Standard Library – per-header cost and impact

- C++ Compile Health Watchdog by Philip Trettner
  - Let's take a look together!

- *Example:* the cost of `<algorithm>`
  - Removing it and hardcoding `min` resulted in ~260ms speedup on a `.cpp` (MSVC) [src]
  - Similar improvements in SFML, but PR #1783 initially rejected!
    - Sneaked it in as part of PR #1909 😉

- `<algorithm>` gets bigger and slower with every standard [src]
  - C++14: 0.09s
  - C++17: 0.29s *(~3x slower)*
  - C++20: 0.70s *(~7.5x slower than C++14!)*

- Every time you `#include` a Standard Header, you may be paying a big price – why?
  - Compilation speed not a priority for Standard Library implementers
  - Headers are implemented with backwards-compatibility in mind
  - The C++ Standard Library is heavily templated and overly complicated at times
  - Check it out yourself – look at `libstdc++`, `libc++`, or Microsoft headers

Standard Library – tips, complaining, contributing, competing

- Physical design considerations and PCH apply here as well
  - E.g., prefer `<iosfwd>` to `<iostream>`; place heavy-hitters in PCH
  - …can you forward-declare Standard Library types? *(covered in next slides)*

- Requesting improvements in this area is difficult
  - Standard Library implementers are mostly volunteers
  - Massive backlog for features and bug fixes – build time takes low priority 😔
  - Keeping backwards compatibility is important and extremely difficult

- Even contributing improvements is very difficult!
  - I tried removing `<tuple>` from `<memory>` for `libstdc++` [diff]
    - `std::tuple<T*, TDeleter>` is used to store the state of `std::unique_ptr`
    - I changed it to a custom pair
  - Dreams crushed by Jonathan Wakely:
    - *"No, it would be a ABI break. It would change the layout for a deleter with the `final` specifier."*
    - GDB pretty-printers might also stop working
    - Generally opposed to small compilation time optimizations

- "Competing" suprisingly often makes sense *(covered in next slides)*

Standard Library – forwarding headers

- ⚠️ **ENTERING UNDEFINED BEHAVIOR TERRITORY** ⚠️

- It is UB to add declarations to `std` unless explicitly permitted by the C++ Standard
  - *"Rules are made to be broken"* 😎
  - It is quite hard to get the forward declarations right!

- To the rescue: Standard Library forward declaration libraries
  - `cpp-std-fwd` by Philip Trettner – outdated, but started the idea
  - `stdfwd` by Oleh Fedorenko – successor fork, supports all of C++17 – see also: C++Now 2021 lightning talk

```
#include <stdfwd/string> // <== use `stdfwd/XXX`
#include <stdfwd/vector>

struct Conference
{
    virtual std::string getName() const = 0;
    virtual stdfwd::vector<Person> getParticipants() const = 0;
        //  ^~~~~~~~~~~~~~~~~~~~~~
        // for classes with default template arguments, use `stdfwd::XXX`
};
```

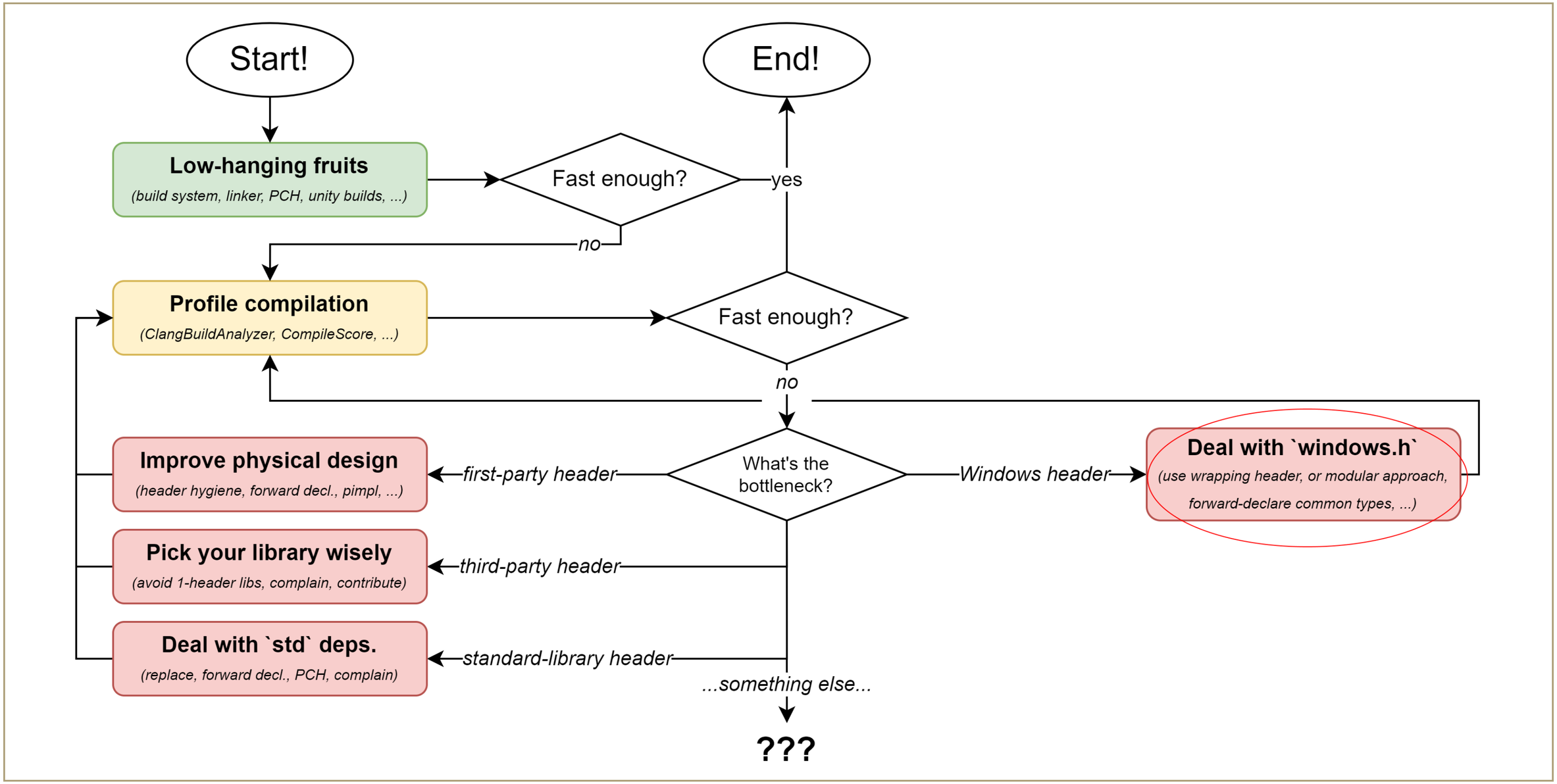

Standard Library – hand-written replacements

- In certain situations, writing your own replacement makes sense

- *Example:* `std::unique_ptr`
  - Fairly easy to write a replacement that exposes only what you need
  - Save ~92-166ms per source file including `<memory>` [src]
  - Zero-dependency barebones implementation in Open Hexagon here

- Consider using 3rd-party alternatives written by compile-time–aware people
  - `magnum-singles` by Vladimír Vondruš
    - Provides self-contained single-header replacements for `std::unique_ptr`, `std::optional`, and more…
    - Excellent blog post with in-depth benchmarks here
    - `Pointer.h` is around ~3x faster than `<memory>` (`std::unique_ptr`)
    - `Reference.h` is around ~5x faster than `<functional>` (`std::reference_wrapper`)
  - `corrade` by Vladimír Vondruš
    - Multiplatform utility and container library
    - *"complementing STL features with focus on compilation speed, ease of use and performance"* [src]
    - Another great article on why his libraries compile quickly here

Building the flowchart (6)



`windows.h` – wrapper header

- Wrap the inclusion of `windows.h` in a separate header [SFML PR #1896]
  - The header will define a few preprocessor symbols before including `windows.h`

```
// WindowsHeader.hpp (from SFML)
#pragma once

#ifndef NOMINMAX
#define NOMINMAX
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

// ...

#include <windows.h>
```

- `NOMINMAX` prevents `min` and `max` macros from escaping
- `WIN32_LEAN_AND_MEAN`
  - Excludes less common APIs such as Cryptography, DDE, RPC, Shell, and Windows Sockets [src]

`windows.h` – forwarding declarations

- It is technically possible to forward-declare `windows.h` types
  - See Stefan Reinalter's tweet (@molecularmusing)
  - See Sebastian Aaltonen's tweet (@SebAaltonen)

```
using HANDLE = void*;
using WPARAM = unsigned long long;
using LPARAM = long long;
using LRESULT = long long;

#define FORWARD_DECLARE_HANDLE(name) struct name##__; using name = name#__*

FORWARD_DECLARE_HANDLE(HINSTANCE);
FORWARD_DECLARE_HANDLE(HWND);
FORWARD_DECLARE_HANDLE(HDC);
FORWARD_DECLARE_HANDLE(HGLRC);
```

- Portable?
  - Probably, these defines have never changed on Windows
  - Might need `#if` for 32-bit support or weird targets like ReactOS

windows.h – modular replacement

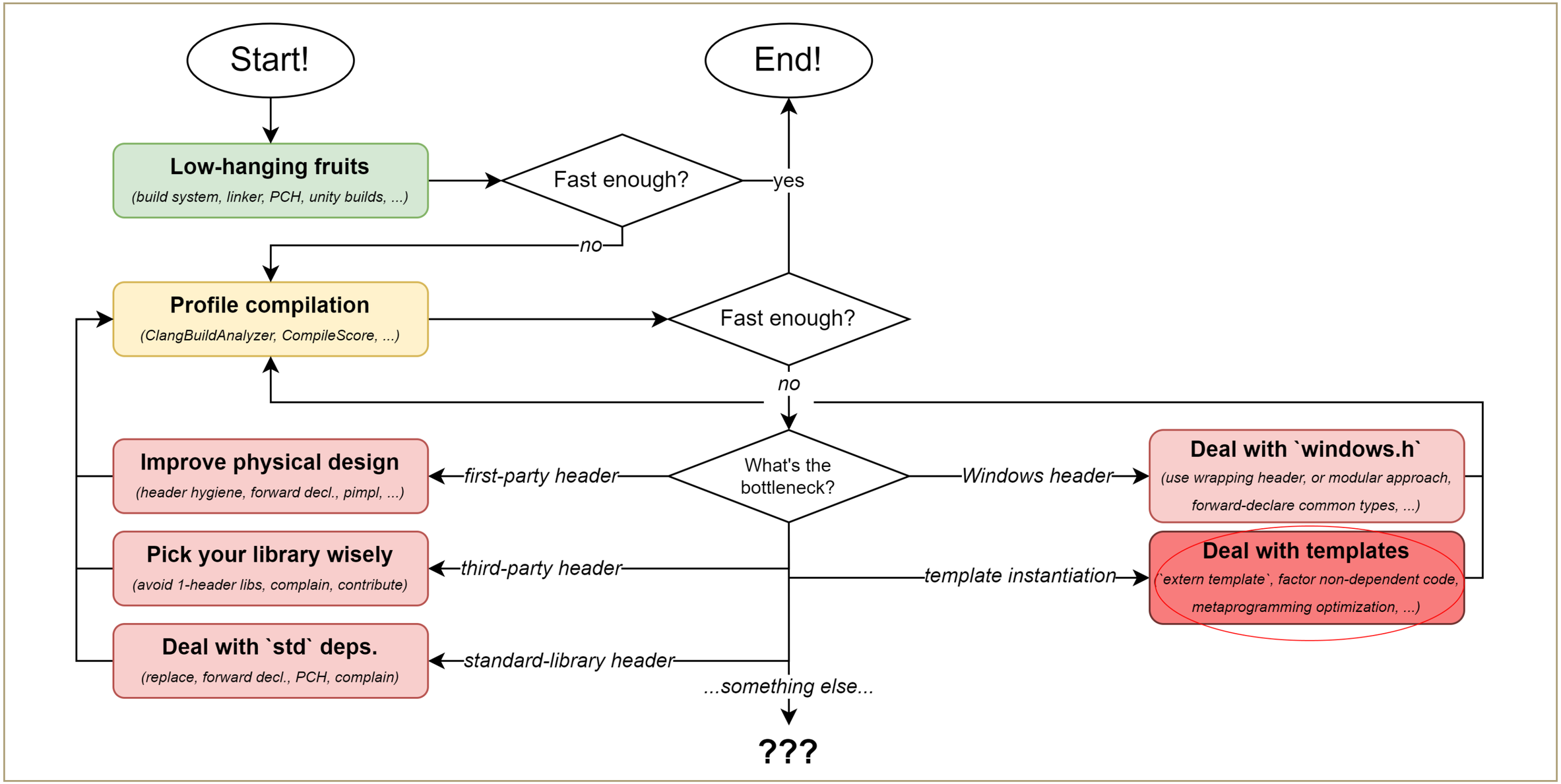
- Free and open-source modular replacement for `windows.h`
  - `WindowsHModular` by Arvid Gerstmann

```
windows_base.h  file.h  process.h
atomic.h        gdi.h   sysinfo.h
dbghelp.h       io.h    threads.h
dds.h           misc.h  window.h
```

- Caveats:
  - Not officially supported by Microsoft
  - Still work-in-progress: some APIs missing (contributions welcome)
  - Unicode/ASCII functions missing generic macro (must use `A` or `W` suffix explicitly)

- Might try using this for SFML 3.x and measure impact in the future

Building the flowchart (7)



Templates – general tips

- Some guidelines:
  - Prefer `constexpr` functions to template metaprogramming
  - Consider run-time polymorphism *(e.g., type erasure)* instead [src]
  - Avoid recursive variadic template instantiation, prefer `...` or fold tricks [src]
    - Used a fold over `=` to optimize `sqlite_orm` reverse tuple iteration [src]
  - Prefer `if constexpr` to SFINAE whenever possible
  - Define templates in source files whenever possible

- Factor out non-dependent code:

```cpp
template <typename T>
class my_optional
{
private:
    alignas(T) std::byte _buf[sizeof(T)];
    bool _has_value;

public:
    bool has_value() const;
    // ...
};
```

```cpp
struct my_optional_base
{
    bool _has_value;
    bool has_value() const;
};

template <typename T>
class my_optional : my_optional_base
{
    /* ... */
};
```


Templates – `extern template (0)`

- *"Explicit instantiation declaration"*
  - Suppresses implicit generation of object code for a particular template specialization in a TU
  - Must be paired with *"explicit instantiation definitions"*

```
// vec3_util.hpp

template <typename T>
vec3<T> cross(const vec3<T>& lhs, const vec3<T>& rhs) { /* ... */ }

extern template vec3<float> cross(const vec3<float>&, const vec3<float>&);
extern template vec3<double> cross(const vec3<double>&, const vec3<double>&);
```

- *"Explicit instantiation definition"*
  - Forces generation of object code for a particular template specialization in a TU

```
// vec3_util.cpp
#include "vec3_util.hpp"

template vec3<float> cross(const vec3<float>&, const vec3<float>&);
template vec3<double> cross(const vec3<double>&, const vec3<double>&);
```


Templates – `extern template` (1)

```cpp
// source0.cpp
#include "vec3_util.hpp"
void f0() { cross(vec3{...}, vec3{...}); }
```

```cpp
// source1.cpp
#include "vec3_util.hpp"
void f1() { cross(vec3{...}, vec3{...}); }
```

```cpp
// source2.cpp
#include "vec3_util.hpp"
void f2() { cross(vec3{...}, vec3{...}); }
```

```cpp
// source3.cpp
#include "vec3_util.hpp"
void f3() { cross(vec3{...}, vec3{...}); }
```

- What happens?
  - Each source file gets an `extern template` by including `vec3_util.hpp`
  - If we don't link against `vec3_util.o`, linking will fail!
  - If we do, none of the `sourceX.cpp` files trigger an instantiation of `cross`
    - They simply link against the instantiations forced in `vec3_util.cpp`
- What's the benefit?
  - No `extern template` → 4 duplicate instantiations of `cross`, coalesced by linker
  - With `extern template` → 1 single instantiation of `cross`, no extra work by linker

Templates – `extern template` (2)

- General strategy [src]
  - In the header, place template definition + `extern template` for common specializations
  - In the source, place explicit `template` instantiations for those same specializations

- One of my experiences
  - I measured `glm` template instantiations being a bottleneck [src]

```
// glmwrapper.h
#pragma once
#include <glm.hpp>
extern template struct glm::vec<4, float, glm::packed_highp>;
```

```
// glmwrapper.cpp
#include "glmwrapper.hpp"
template struct glm::vec<4, float, glm::packed_highp>;
```

- **Q:** how much was my compilation time speedup?
  - **A:** 0% 😩

Templates – `extern template` (3)

- Quotes by Davis Herring (ISO C++ Committee): [src]

> *"explicit instantiation declaration of a class template doesn't prevent (implicit) instantiation of that template;* ***it merely prevents instantiating its non-inline, non-template member functions****"*
>
> *"****code which requires that the class be complete still needs to know its layout and member function declarations*** *(for overload resolution), and in general there's no way to know those short of instantiating the class"*

- In short:
  - `extern template` helps a lot with non-inline functions expensive to instantiate
  - Doesn't help much if the bottleneck is the instantiation of a class template itself

- Modules will help!

> *"an explicit instantiation definition in a module interface allows caching the instantiation in the module data, avoiding both parsing and instantiation in importing translation units"*
>
> *"modules remove the implicit `inline` on class members and friends defined in the class increasing the number of functions for which `extern template` prevents implicit instantiation"*

Templates – `extern template` (4)

- More pain points:
  - Easy to get puzzling linker errors
  - Easy to not get any benefit
  - Still a few bugs around… [GCC #109387] [GCC #109380]
  - Syntax can be verbose and unintuitive [src]
  - Duplicate explicit instantiations cause linker errors [src]
    - Very annoying with platform-specific type aliases!
  - Attempts to simplify the features were shut down by EWGI in 2019 [src]
  - Want to support all platforms + MinGW + older MSVC versions? Good luck! [SFML PR #2496]

- Worth it?
  - As always, *measure*!
  - Minor ~0.6s (~5.4%) speedup in SFML test suite [SFML PR #2424]

Templates – metaprogramming (0)

- Rule of Chiel
  - Named after Chiel Douwes [src]
  - Presented by Odin Holmes

- From most expensive to least expensive:
  - SFINAE
  - Instantiating a function template
  - Instantiating a type
  - Calling an alias
  - Adding a parameter to a type
  - Adding a parameter to an alias call
  - Looking up a memorized type

- Let's put it in practice!
  - Reimplementing `std::conditional_t` (example from Odin Holmes [src])

[@chieltbest]

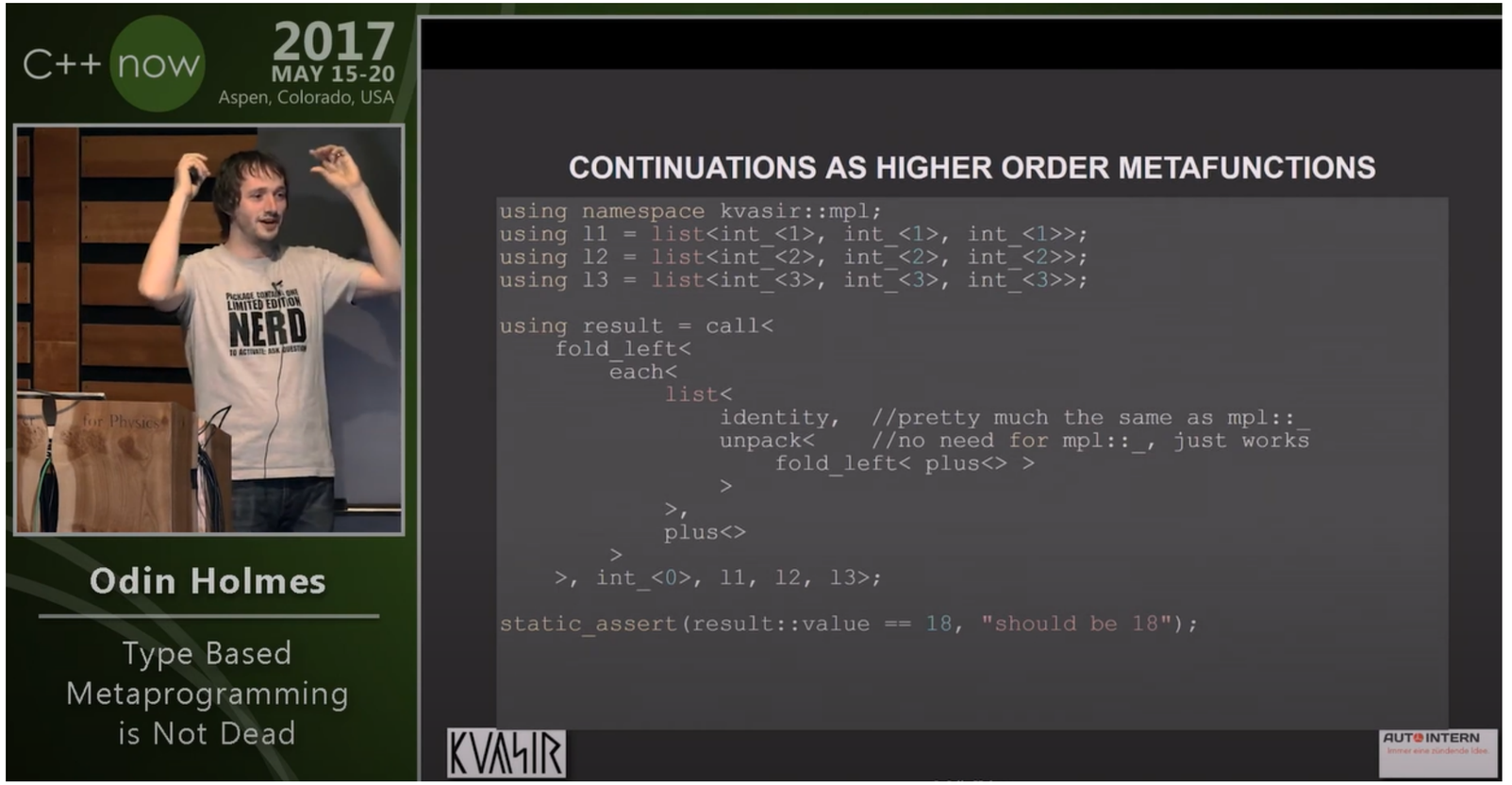
Templates – metaprogramming (1)

```
template <bool B, typename T, typename F>
struct conditional { using type = T; };

template <class T, typename F>
struct conditional<false, T, F> { using type = F; };

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B, T, F>::type;
```

```
template <bool B>
struct conditional
{
    template <typename T, typename F> using f = T;
};

template <>
struct conditional<false>
{
    template <typename T, typename F> using f = F;
};

template <bool B, typename T, typename F>
using conditional_t = typename conditional<B>::template f<T, F>;
```


Templates – metaprogramming (2)

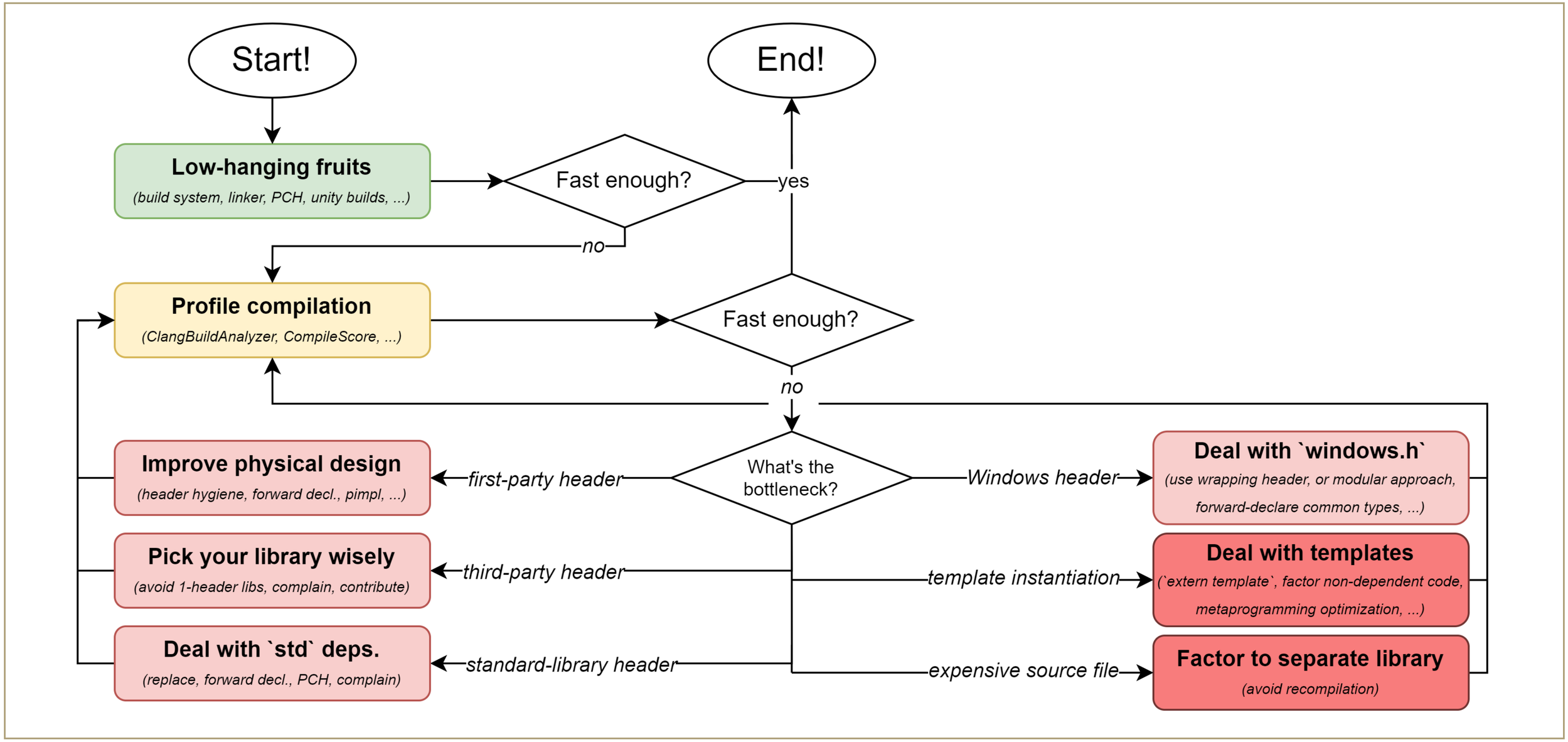
- This metaprogramming style can be chained (*"zero-cost composition"*)



[Odin Holmes -- "Type Based Template Metaprogramming is Not Dead" [C++Now 2017]]

- Incredibly fast – `kvasir` blows the competition away
  - Check out http://metaben.ch/

The completed flowchart



Low-hanging fruits – checklist

- ☐ Pick a better build system *(e.g., use `ninja`)*
- ☐ Pick a better linker *(e.g., use `lld` or `mold`/`sold`)*
- ☐ Use a compilation cache *(e.g., use `ccache`)*
- ☐ Check your build machine configuration and running processes *(e.g., anti-virus)*
- ☐ Improve the hardware of your build machine *(e.g., CPU, RAM, NVMe drives)*
- ☐ Enable precompiled headers *(e.g., `PCH.hpp` + `target_precompile_headers`)*
- ☐ Enable unity builds *(e.g., `-DCMAKE_UNITY_BUILD=ON`)*
- ☐ Prefer dynamic linking to static linking *(e.g., `-DBUILD_SHARED_LIBS=ON`)* – see Extras

Conclusion

Thank you for attending!

- Goals:
  - ☑ **Understand what can negatively impact build times**
  - ☑ **Provide *actionable* points to improve your compilation times**
  - ☑ **Call to action: improve your favorite open-source project's build**
  - ☐ **Spark some interesting discussion!**
- Detailed analysis of Modern C++ features: **EMC++S**!
  - 📕 https://emcpps.com
  - No opinions: just *facts*, *use cases*, *pitfalls*, and *annoyances*
- SFML 3.x – *"Simple and Fast Multimedia Library"*
  - https://sfml-dev.org/
- Open Hexagon – open-source arcade game made with SFML
  - https://openhexagon.org/
- Let's keep in touch!
  - mail@vittorioromeo.com

- Thanks!
  - 🙋 Questions? 💁 Comments? 🙆 Criticism? 🤳 Stories?
  - https://vittorioromeo.com | @supahvee1234 | https://github.com/vittorioromeo | mail@vittorioromeo.com

References

- References and further reading/viewing material
  - `magnum` blog - “Forward-declaring STL container types”
  - `magnum` blog - “Lightweight but still STL-compatible unique pointer”
  - “The Hitchhiker’s Guide to Faster Builds” - Viktor Kirilov - CoreHard Spring 2019
  - Tobias Hieta: “Compiling C++ is slow - let’s go faster” - SwedenCpp
  - Arne Mertz - “Reduce Compilation Times With extern template”
  - Annileen Devlog #2 - “C++20 and Modules”
  - Arthur O’Dwyer - “SCARY metafunctions”
  - QT Blog - “Precompiled Headers and Unity (Jumbo) Builds in upcoming CMake”
  - Christoph Heindl - “Reducing Compilation Time: Unity Builds”
  - virtuallyrandom - “C++ Compilation: What’s Slowing Us Down?”
  - Roy Jacobson - “C++20 Modules Status Report”
  - David Röthlisberger - “Benchmarking the Ninja build system”
  - Viktor Kirilov - “CMake 3.16 added support for precompiled headers & unity builds - what you need to know”
  - Viktor Kirilov - “A guide to unity builds”
  - methodpark - “The C/C++ Developer’s Guide to Avoiding Office Swordfights – Part 1: ccache”
  - Philip Trettner - “C++ Compile Health Watchdog”
  - Aras Pranckevičius - “Investigating compile times, and Clang -ftime-report”
  - Aras Pranckevičius - “time-trace: timeline / flame chart profiler for Clang”
  - Aras Pranckevičius - Clang Build Analyzer
  - CppCon 2016: John Lakos “Advanced Levelization Techniques”
  - “C++ Modules and Large-Scale Development” - John Lakos [ACCU 2019]
  - J. Lakos, V. Romeo, R. Khlebnikov, A. Meredith - “Embracing Modern C++ Safely”
  - Jonathan Müller - “Nifty Fold Expression Tricks”
  - Kevin Cadieux - “Introducing vcperf /timetrace for C++ build time analysis”

Extras

Extras – More information on modules (0)

- From Roy Jacobson's C++20 Modules Status Report
- MSVC has a complete implementation since VS2022 17.5
  - Many open bug reports
- GCC still has an incomplete implementation
  - Many open bug reports
  - Progress stalled until September 2022
  - Module scanning protocol (P1689R5) ETA: 2024 [src]
- Clang still has an incomplete implementation
  - Many open bug reports
- CMake has *experimental* support via magic flag
  - Relies on the module scanning protocol from P1689 – GCC unsupported
  - Many open bug reports

```
set(CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API "2182bf5c-ef0d-489a-91da-49dbc3090d2a")
```

Extras – More information on modules (1)

- Victor Zverovich achieved 4x speedup with modules for `fmtlib` [src]
  - Not compared against PCH
  - Of course, issues with `extern template` [src]
- Nice blog posts on the subject:
  - Kitware - “import CMake; C++20 Modules”
  - Victor Zverovich - “Simple usage of C++20 modules”
- Modules are being actively worked on
  - E.g., Bloomberg is sponsoring Kitware to work on CMake support [src]
- Full support for modules + CMake on Matt Godbolt’s Compiler Explorer
  - *Example:* https://godbolt.org/z/aTr8crhcE
    - by Bill Hoffman and Kitware
- My opinion:
  - As with most other things in C++, headers will never truly disappear
  - Compilation speed optimization techniques will not become obsolete due to modules
  - Modules will eventually make our lives easier

Extras – *Low-hanging fruits* – dynamic linking (0)

- Dynamic linking is usually faster than static linking
  - Especially due to symbol visiblity
- Enable globally in CMake via `-DBUILD_SHARED_LIBS=ON`
  - Alternatively, select on a per-case basis with `add_library`
- Symbol visibility:
  - On Windows, only marked symbols are exported in `.dll` files
    - Use portable macros to export/import symbols
  - On UNIX, all symbols are exported *(poor default!)*
    - Use `-fvisibility=hidden` to change the default, or CMake

```
set_target_properties(${target} PROPERTIES
                      CXX_VISIBILITY_PRESET hidden
                      VISIBILITY_INLINES_HIDDEN YES)
```

- No measurable difference on SFML 3.x full rebuild 😕
  - SFML does use hidden visibility by default
  - YMMV – significantly helps in some cases *(e.g., a lot of template-generated symbols)* [src]

Extras – *Low-hanging fruits* – dynamic linking (1)

- *Example:* portable macro-based symbol visibility API

```
#if defined(SFML_STATIC)
    #define SFML_API_EXPORT
    #define SFML_API_IMPORT
#else
    #if defined(SFML_SYSTEM_WINDOWS)
        #define SFML_API_EXPORT __declspec(dllexport)
        #define SFML_API_IMPORT __declspec(dllimport)
    #else // Linux, FreeBSD, Mac OS X
        #define SFML_API_EXPORT __attribute__((__visibility__("default")))
        #define SFML_API_IMPORT __attribute__((__visibility__("default")))
    #endif
#endif
```

```
#if defined(SFML_SYSTEM_EXPORTS)
    #define SFML_SYSTEM_API SFML_API_EXPORT
#else
    #define SFML_SYSTEM_API SFML_API_IMPORT
#endif
```

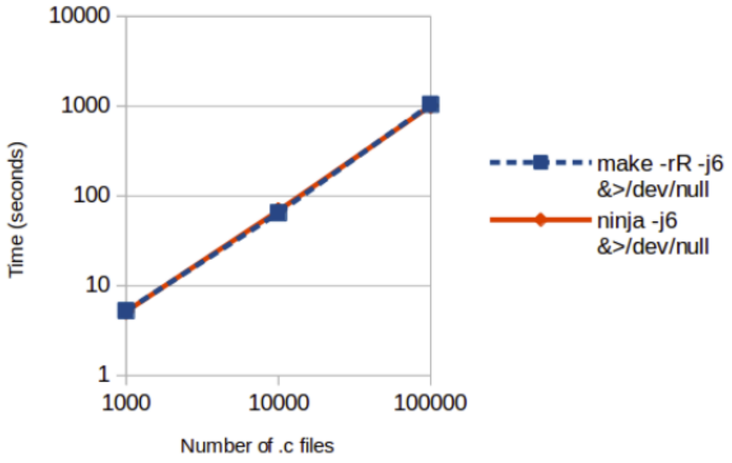
```
set_target_properties(${NAME} PROPERTIES DEFINE_SYMBOL ${NAME_UPPER}_EXPORTS)
```

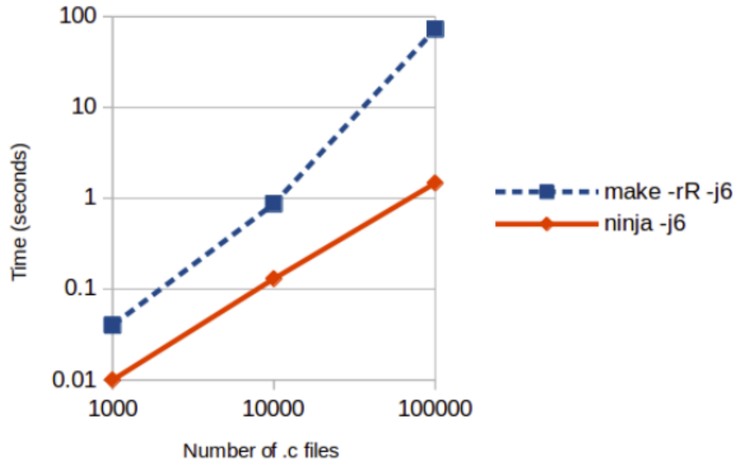

Extras – *Low-hanging fruits* – more on `ninja`

- Origin of the name `ninja`: *"quiet and strikes quickly"* [src]
- Originally created for Chrome
  - No-op build (all targets up-to-date) with `make` took ~10s, less than ~1s with `ninja` [src]

Fresh build		
# of .c files	**Make**	**Ninja**
1,000	5.3s	5.2s
10,000	65s	69s
100,000	17m24s	16m56s

No-op build		
# of .c files	**Make**	**Ninja**
1,000	0.04s	0.01s
10,000	0.87s	0.13s
100,000	73s	1.5s

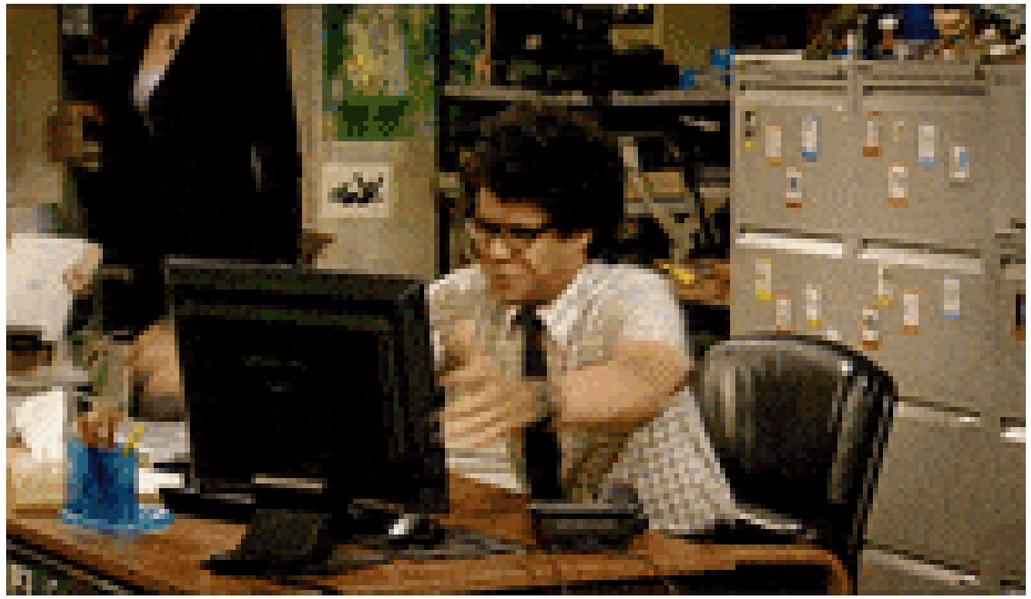




[David Röthlisberger -- Benchmarking the Ninja build system]

Extras – *Low-hanging fruits* – build machine hardware

- For completeness, but yeah – can always throw more hardware at it



[Channel 4 -- The IT Crowd [Tenor.com]]

- Not sure what hardware to purchase?
  - Phoronix offers CPU compilation benchmarks (e.g., Godot, LLVM, Linux)
  - TechPowerUp offers some as well
  - On YouTube: Hardware Unboxed & GamersNexus [src]
  - Phoronix sometimes also has NVMe drive benchmarks

Extras – *Appendix* – unity builds (1)

- *Example:* clashing symbols

```cpp
// source0.cpp

namespace
{
    void f() { /* ... */ }
}
```

```cpp
// source1.cpp

namespace
{
    void f() { /* ... */ }
}
```

- If `source0.cpp` and `source1.cpp` end up in the same unity build chunk...
  - ...the program will fail to compile, because `f` will be defined twice
- The fix is easy, but manual:
  - Rename one function to `f0`, the other to `f1`, and update usages
- Annoying requirement: all `static` symbols must be uniquely named
  - In the same `namespace`, at least...

Extras – *Appendix* – unity builds (2)

- *Example:* catching ODR violations

```cpp
// source0.cpp

inline int f() { return 0; }
int call_f();

int main() { return f() + call_f(); }
```

```cpp
// source1.cpp

inline int f() { return 1; }
int call_f() { return f(); }
```

- The behavior of the program above is *undefined*
- If `source0.cpp` and `source1.cpp` end up in the same unity build chunk…
  - …the program will fail to compile, revealing a nasty ODR violation!
- This is very good 😳

Extras – *Appendix* – unity builds (3)

- *Example:* enforcing header guards

```
// bad_header.hpp
/* ...missing include guard or `#pragma once`... */

inline void foo() { /* ... */ }
```

```
// source0.cpp
#include "bad_header.hpp"
/* ... */
```

```
// source1.cpp
#include "bad_header.hpp"
/* ... */
```

- If `source0.cpp` and `source1.cpp` end up in the same unity build chunk...
  - ...the program will fail to compile, revealing the missing header guard
- This is also good 🙂

