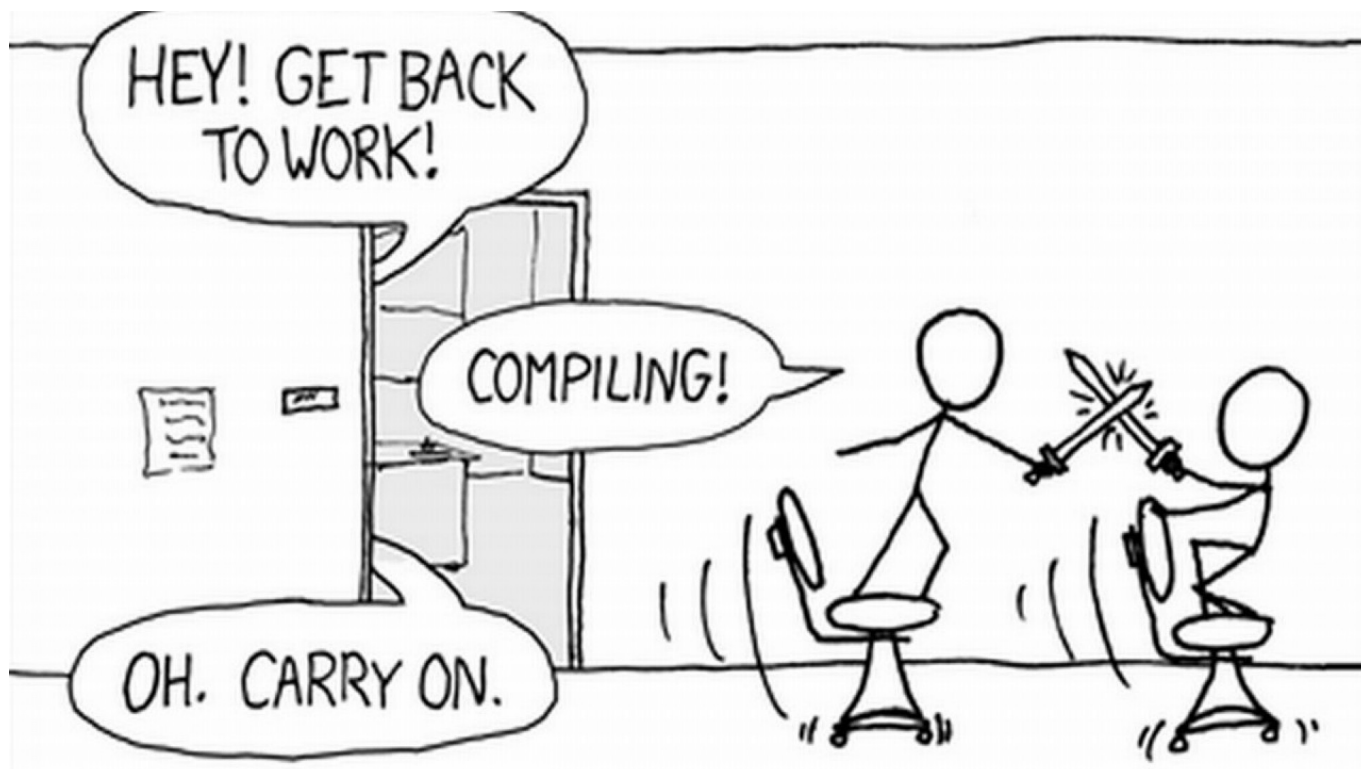


## Ускорение сборки КОМПАСа с помощью rsh



Началось всё с того, что нам потребовался linux, который за собой потянул переход на CMake. Используемая система сборки умеет только под Windows, точнее позволяет её лицензия. Для всего остального и без того немалая стоимость удваивается. А это уже получается хороший билд сервер. Можно конечно генерировать обратно в sln и собирать им, но уж очень хотелось поменьше лишних телодвижений. Поэтому решили что-то делать.

На руках были результаты прошлогодней давности, когда пытались сделать что-то [похожее](#).

Тогда все уперлось во время парсинга и решили оставить ~~пока не припечет~~ до лучших времен.

К счастью пока делали CMake, прибыл новенький билд сервер на 128 ядер (256 потоков) и 1 ТиБ оперативной памяти.

Было где развернуться.

Переход на CMake в качестве бонуса дал доступ к [Ninja](#), которых значительно лучше распределяет задачи сборки, по сравнению с MSBuild.

Создаем RAM drive, кладем туда исходники, tmp и прочее. Отрубаем все лишние процессы. Первый запуск сборки и ... получаем время более 10 минут и только половину загрузки CPU. Сказать что это медленно для такой машины, ничего не сказать.

Тут явно что-то не так. Для начала попробуем разобраться, почему так долго. Посмотрим на исходные данные.

Что есть ?

- ~3 миллиона строк

- ~4500 .cpp & ~4000 .h
- 55 проектов
- boost, stl, winapi

Проект достаточно давний, при этом активно развивается.

Начнем с самого простого.

Зная что в основном тормозит парсинг, попробуем найти что именно.

Первое что приходит на ум, распарсить все include, посчитать строки и попытаться избавиться от самых тяжелых headers.

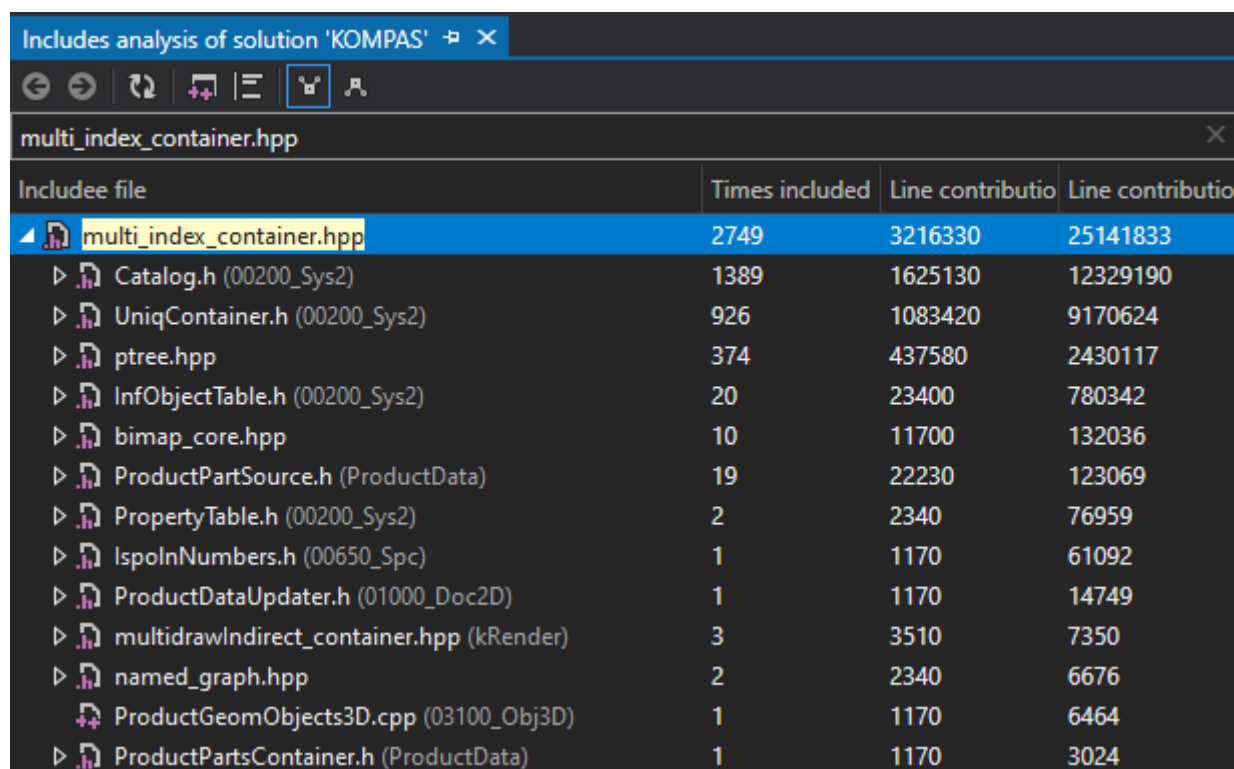
Парсить все руками, разбирать все макросы, учитывать настройки компилятора. Задача выглядит достаточно распространенной, может уже есть готовый инструмент ?

Оказывается что есть

- Resharper C++ Analyze Includes

И делает именно то что требуется. Запускаем и ждем, ждем и еще немного. Парсинг c++ дело не быстрое.

## Resharper C++ Analyze Includes



Includee file	Times included	Line contributio	Line contributio
<b>multi_index_container.hpp</b>	<b>2749</b>	<b>3216330</b>	<b>25141833</b>
▶ Catalog.h (00200_Sys2)	1389	1625130	12329190
▶ UniqContainer.h (00200_Sys2)	926	1083420	9170624
▶ ptree.hpp	374	437580	2430117
▶ InfObjectTable.h (00200_Sys2)	20	23400	780342
▶ bimap_core.hpp	10	11700	132036
▶ ProductPartSource.h (ProductData)	19	22230	123069
▶ PropertyTable.h (00200_Sys2)	2	2340	76959
▶ IspolnNumbers.h (00650_Spc)	1	1170	61092
▶ ProductDataUpdater.h (01000_Doc2D)	1	1170	14749
▶ multidrawIndirect_container.hpp (kRender)	3	3510	7350
▶ named_graph.hpp	2	2340	6676
▶ ProductGeomObjects3D.cpp (03100_Obj3D)	1	1170	6464
▶ ProductPartsContainer.h (ProductData)	1	1170	3024

И что мы видим, boost. его много и он всюду. Один хедер дает в семь строчек больше чем весь исходный код проекта.

Пробуем с этим что-то сделать. Пока что трогать исходники не сильно хочется.

PCN

[PCH](#) позволяет распарсить один раз включаемые заголовочные файлы, а после при каждом обращении выдавать уже готовое внутреннее представление (обычно это сериализованный ast).

Замеры будем проводить на типичной машине простых смертных, т.к. на ней собирать каждый день.

Попробуем добавить самые жирные хедера. И Получаем результат. Ускорение примерно в 2 раза. Похоже на чудо. Ничего особо не делая, ускоряем компиляцию в 2 раза, с 50 минут до 25.

Но этого мало. Кажется что можно быстрее.

Смотрим чтобы еще ускорить. Обращаем внимание на сами pch. Размер мягко говоря немаленький, 500 МиБ и более.

Тут уже придется поработать скальпелем. Начинаем с самых жирных кусков и смотрим, действительно ли они там нужны.

Как оказалось что нет. multi\_index\_container.hpp используется в интерфейсном заголовке, но почему то включен полностью. Сам интерфейс используется достаточно часто, точнее почти везде. Попробуем немного подсократить.

Для многих тяжелых хедеров существует версия с предварительным объявлением, с суффиксом \_fwd. Это справедливо как для boost, так и stl и прочих.

Заменяем, пробуем собрать. Расставляем в .cpp полный хедер по необходимости и спустя несколько часов попыток добиваемся компиляции проекта.

Смотрим на топ 10 тяжелых хедеров и видим что boost там уже стало меньше. Повторяем расстановку хедеров в pch и снова собираем. И видим что время сборки уменьшилось до 20 минут.

Уже лучше. Относительно небольшими усилиями сократили время сборки. Теперь оно сопоставимо использованию системы параллельной сборки.

Хм. А можно еще лучше ? Конечно. Повторяем операцию по замене топовых хедеров на fwd версию.

Но чем дальше, тем сложнее. В некоторых местах используется реализация. И тут на помощь приходит [pimpl](#). Суть метода в том, что вместо тяжелой структуры используем указатель на предварительно объявленную структуру. Компилятору достаточно знать размер данных. А размер указателя всегда известен.

И тут натываемся на неприятный момент. С сырыми указателями работать совсем не хочется, есть же умные указатели.

Попробуем поставить std::unique\_ptr. И Неожиданно получаем ошибку компиляции.

```
class Foo
{
private:
    std::unique_ptr<struct Bar> _bar;
public:
    Foo();
    ~Foo() = default;
}
```

В чем же дело ? Вроде бы нигде не создаем ничего лишнего, но все равно ругается на incomplete type. Присмотримся внимательней. И что мы видим ? Деструктор то остался в заголовке, а значит что для всех полей также должны быть деструкторы. Поэтому верно ругается что не может удалить, т.к. тип неизвестен. Решением является перенос реализации деструктора в .cpp файл.

Подобным образом удаляем остальные тяжелые хедеры. Спустя пару недель... получаем ускорение еще на 5 минут. 15 минут, почти как в лучшие времена с использованием incredibuild, но это на типовой локальной машине.

А что дальше ? Можем ведь лучше. Точно знаю.

А дальше остаются более мелкие boost заголовки, stl и собственные хедеры и прочая мелочь, которая в совокупности набирает немалый вес. Их уже слишком много чтобы разность по rimpl и fwd. Вот их бы и запихнуть вpch да поплотнее.

И тут Resharper C++ начинает нехватать. Он выдает статистику по числу строк, но это не всегда соответствует времени компиляции.

Нужно что-то помощней. Вот бы взять какой нибудь полноценный профилировщик, который выдавал все как есть. И как оказалось для msvc относительно недавно microsoft выкатили такой.

## Include What You Use (iwyu)

- удаляет лишние include
- добавляет forward declaration

Также для автоматизации есть инструмент [iwyu](#), который поможет немного сократить число лишних инклюдов, расставляя forward declaration и удаляя лишние include

Для работы нужно чтобы проект компилировался под clang, иначе результат будет некорректным. Собираем сам iwyu. Собираем llvm, с включенными опциями.

- `llvm`
- `-DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra"`

Теперь собираем сам iwyu. Для работы так же нужна compile\_commands.json. Запускаем.

```
// найти лишние include
iwyu_tool -j 256 -p compile_commands.json -- -w > iwyu_res.cpp
```

На выходе получаем список исправлений. Напрямую его применять не советую. Сама программа находится в beta версии и может поудалять чего лишнего.

```
// применить исправления
fix_includes.py < iwyu_res.cpp
```

Применять лучше по проектно, просматривая предлагаемые изменения. Всяко это будет быстрее чем руками, но полностью автоматически результат не гарантирован.

## C++ Build Insights

Инструмент, позволяющий заглянуть в процесс сборки, понять что все это время делает компилятор и линковщик. Начиная от времени парсинга отдельных файлов вплоть до анализа времени генерации и отдельных функций.

Прежде чем воспользоваться, нужно настроить систему и установить несколько инструментов.

Для этого нам понадобится свежий ADK и Visual Sdtudio 2019.

После установки ADK нужно настроить. Для этого лезем в нутрь студии и ищем `perf_msvcbuildinsights.dll`. Он отвечает за отображение результатов. Теперь заходим в WPA копируем его туда. Так же нужно прописать в настройках `perfcore.ini` add `perf_msvcbuildinsights.dll`

## C++ Build Insights

- Visual Studio 2019
- [latest Windows ADK](#)
- `perf_msvcbuildinsights.dll` -> WPA directory
- `perfcore.ini` add `perf_msvcbuildinsights.dll`

Начнем замерять. Для этого от имени администратора стартуем консоль разработчика x64 `Native Tools Command Prompt for VS 2019` (vcvars64) от имени администратоа. И стартуем сессию `vcperf /start MySessionName`

Сам процесс выглядит следующим образом. Запускаем профилировщик. Глобально. Неважно из какой директории, главное чтобы хватало место и прав на запись.

Далее стартуем сборку. Захватываются все события системы. Поэтому не должно быть ничего лишнего. Также потребуется немало оперативной памяти, чем больше проект тем больше. В данном случае хватило 32 Гиб.

После завершения компиляции устанавливаем профилировщик `vcperf /stop MySessionName outputFile.etl`

## C++ Build Insights

- `vcperf /start MySessionName`
- compile your project
- `vcperf /stop MySessionName outputFile.etl`

Если все было установлено и настроено правильно, то при открытии отчета видим дополнительные вкладки

## C++ Build Insights

Line #	Activity Name	Included Path	Inclusive Duration (s)	Sum	Wall Clock Time Responsibility (s)	Exclusive	Start Time (ns)	Inclusive Duration (s)	Legend
1	▼ Parsing		291,248,166,000,000		794,341,000,000				
2		A:\src\Source\ProductData\PropertyData.h	4,084,544,000,000		13,616,000,000				
3		A:\src\Source\2D\FSys\Option.h	2,800,288,000,000		9,320,000,000				
4		A:\src\Tools\Boost\boost\variant\variant.hpp	2,689,562,000,000		8,409,000,000				
5		A:\src\Source\2D\FSys\Cfgopt.h	2,526,610,000,000		8,335,000,000				
6		A:\src\Source\2D\DocT\DocT\DocTedit.h	2,301,613,000,000		8,173,000,000				
7		A:\src\Source\2D\ObjS\Obj.h	2,572,620,000,000		8,169,000,000				
8		A:\src\Source\2D\Options\EmbodimentsTree.h	2,518,660,000,000		8,049,000,000				
9		A:\src\Source\ProductData\PropRefsBase.h	2,409,359,000,000		7,561,000,000				
10		A:\src\Source\ProductData\MetaProductData.h	2,218,591,000,000		7,046,000,000				
11		A:\src\Source\2D\ObjS\Complobj.h	2,017,424,000,000		6,670,000,000				
12		A:\src\Source\2D\Options\Wsdef.h	2,101,165,000,000		6,636,000,000				
13		A:\src\Source\2D\ObjS\Objlist.h	1,949,444,000,000		6,472,000,000				
14		A:\src\Source\2D\TedB\Ted_base.h	1,915,549,000,000		6,115,000,000				
15		A:\src\Source\2D\DocS\DocS\Doccont.h	1,712,567,000,000		5,975,000,000				
16		A:\src\Tools\Boost\boost\multi_index_container.hpp	1,981,786,000,000		5,868,000,000				
17		A:\src\Source\2D\Options\Komdoc.h	1,707,304,000,000		5,686,000,000				
18		A:\src\Source\2D\FSys\Prjopt.h	1,699,975,000,000		5,379,000,000				
19		A:\src\Source\2D\DocE\DocE\Docedit.h	1,473,623,000,000		5,256,000,000				
20		A:\src\Source\3D\Include\io_tape.h	1,861,150,000,000		5,167,000,000				
21		A:\src\Source\3D\K3dObj01\Ifcompnt.h	1,522,513,000,000		5,070,000,000				
22		A:\src\Source\2D\DocT\DocT\DocTedit.h	1,470,662,000,000		5,066,000,000				
23		A:\src\Source\3D\K3dObj01\HotPoint.h	1,495,312,000,000		4,981,000,000				
24		A:\src\Source\2D\FSys\Catalog.h	1,618,259,000,000		4,959,000,000				
25		A:\src\Source\2D\DocS\DocS\Drfndpt.h	1,414,098,000,000		4,794,000,000				
26		A:\src\Source\2D\Options\Wsdef.h	1,365,502,000,000		4,658,000,000				
27		A:\src\Source\3D\K3dObj01\PARTOBJ.H	1,376,792,000,000		4,593,000,000				
28		A:\src\Source\2D\DocS\Ifdrfsrv.h	1,343,431,000,000		4,513,000,000				
29		A:\src\Source\3D\K3dObj01\foundobj.h	1,291,557,000,000		4,316,000,000				
30		A:\src\Source\ProductData\MetaProductSignals.h	1,268,790,000,000		4,297,000,000				
31		A:\src\Source\2D\ObjE\Leader.h	1,315,306,000,000		4,125,000,000				
32		A:\src\Tools\Boost\boost\signals2.hpp	1,218,208,000,000		4,122,000,000				
33		A:\src\Source\2D\ObjS\Tcurve.h	1,344,972,000,000		3,995,000,000				
34		A:\src\Source\2D\ObjS\AppearanceObj.h	1,265,915,000,000		3,968,000,000				
35		A:\src\Source\2D\FSys\UniqContainer.h	1,233,713,000,000		3,602,000,000				
36		A:\src\Source\2D\ObjS\Annsymb.h	1,125,564,000,000		3,564,000,000				
37		A:\src\Source\API\API5_2D\IAPViewsAndLayers.h	947,529,000,000		3,538,000,000				
38		A:\src\Source\3D\K3dObj01\OBJECT3D.H	1,055,877,000,000		3,500,000,000				
39		A:\src\Source\3D\K3dObj01\OBJ3DTYP.H	1,057,743,000,000		3,494,000,000				
40		A:\src\Source\3D\K3dObj01\IfGenerativeDimensionsOwner.h	998,825,000,000		3,284,000,000				

В данном случае нас интересует время парсинга файлов. Смотрим и видим что boost еще в топ 10, хотя по числу строк его там нет. Почему так ? Смотрим, а чего же там такого тяжелого ? Вроде ничего. Один boost хедер включает другой, а потом еще один и еще и еще... И так до 1500.

Вспомним как работает компилятор. С берем настройки и начинаем парсить. На каждый файл производится операция открытия, чтения и закрытия. И так каждый раз. А если таких файлов очень много ? Windows просто утопает в создании процесса и открытии файлов. Т.е. получается что наши хедера попали в том потому что застреваем в IO.

Используя предыдущий опыт, так же пробуем избавиться от самых тяжелых, пока не останется много мелочевки. Её попытаемся затолкать в rch.

Так же берем топ N хедеров по проектам и расписываем в rch.

Но так делать каждый раз мягко говоря утомительно. Проект стремительно развивается. Что-то добавляется а что-то удаляется. Вот бы сани-хали-сами, rch сам генерировался.

Для этого нужно как-то достать инклюды из etl, и записать в rch.

Оказывается всё уже есть.

<https://devblogs.microsoft.com/cppblog/analyze-your-builds-programmatically-with-the-c-build-insights-sdk/>

Собираем <https://github.com/microsoft/cpp-build-insights-samples/tree/master/TopHeaders>

Натравливаем на получившийся etl и получаем заветные хедера, которые после обработки записываем в rch

Остается дело за малым. Собираем каждый проект по отдельности, сохраняем отчет. После чего извлекаем топ N хедеров и вставляем их рch.

## Попытка 1

Берем топ 50 для каждого узла. Делаем еще один замер и получаем ускорение еще в несколько минут. Неплохо, но кажется что можно еще лучше.

Внимательней посмотрим на список. Оказывается в топе попадают хедеры которые включены в другие хедеры из того же топа. Т.е. топ 50 на самом деле вовсе не 50, а как повезет.

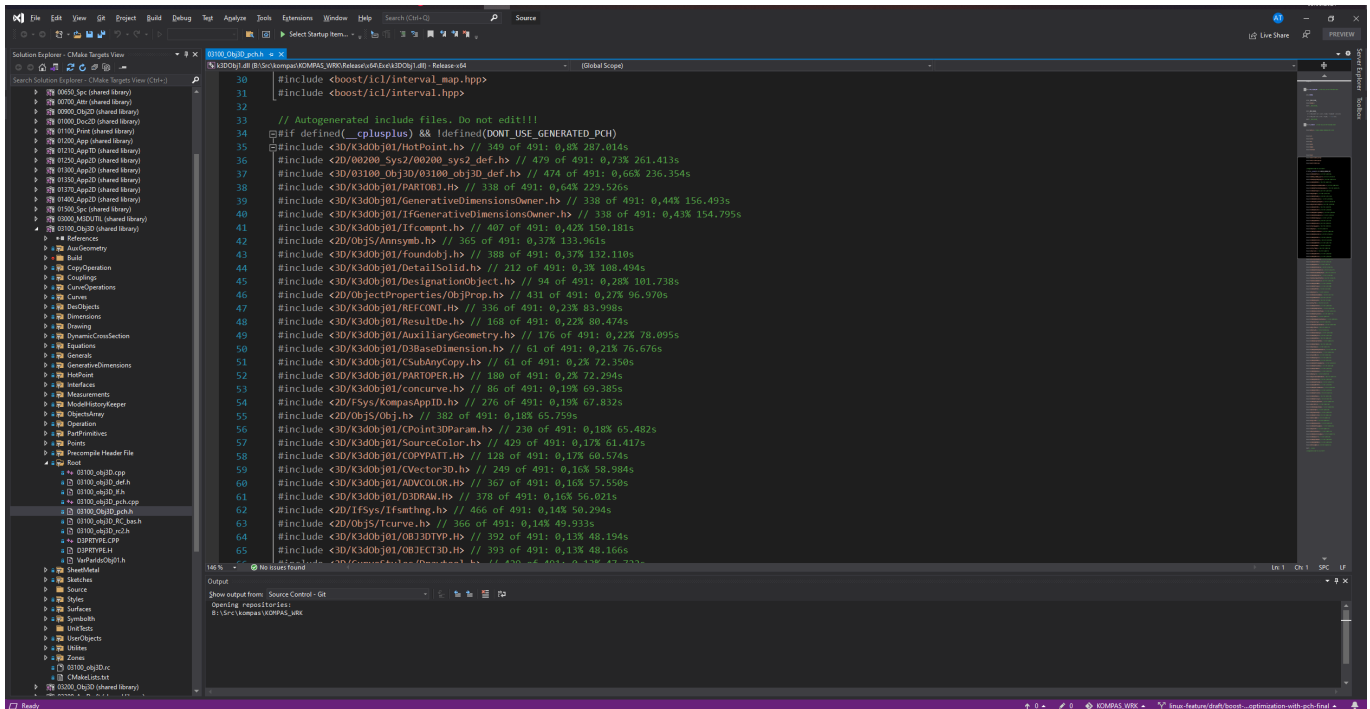
К тому же, как оказалось не все можно взять и просто так включить. Где-то попадают одинаковые typedef с разными значениями или внезапно обнаружились определения макросов, которые пересекаются с именами классов. Проект все таки давний и вряд ли авторы задумывались что кто-то захочет свалить столько хедеров в кучу.

По идее нам нужны только те хедеры, которые включают в себя все остальные.

Оказывается необходимая информация в виде дерева инклюдов есть в etl отчете, которую также можно извлечь.

Далее все просто. Берем полное дерево инклюдов, так же берем топ N инклюдов. Строим граф. Ищем топовые вершины первого уровня. В итоге из 50 осталось около 10, остальные так или иначе включают друг друга.





## Попытка 2

Пробуем, еще минус пару минут. Теперь можно поиграться с настройками. Нужно понять сколько же брать. Чем больше хедеров пихаем в `rch`, тем дольше время сборки на машине со множеством ядер (Ерус на 256 потоков), но на обычной наоборот, получается быстрее. Дело в том, создание `rch` происходит в один поток, в это время остальные ждут, но зато парсинг происходит практически мгновенно. Тут нужно подбирать в зависимости от проекта или написать скрипт, который сделает это сам )

Итак, делаем еще один замер и еще минус пару минут.

Итого с более чем 50 минут, получили около 10 на типовой машине. Естественно результаты могут и отличаться на каждой машине из-за наличия фоновых процессов, работающего антивируса, количества открытых вкладок в хrome. Но в целом получилось быстрее чем с использованием системы параллельной сборки, при этом намного проще чем у коллег из [питера](#) 😊

Еще одним приятным бонусом стала возможность работать не используя всю вычислительную мощь компании, что актуально для удаленных сотрудников, которых стало значительно в такое время.

## Результаты

22m10s	
18m13s	2dc76351284f52c6290026bf752d1ce1b0dc5efd
16m35s	0a63db91fe413b3ac22208b493b0cadd192050b
15m00s	92cb9c7e328b49ff62da04771db19e210947cd86
13m41s	adaf388f816fc9354054e5b8aa4c7c37c115d460
12m59s	d1517387eb1b23c2ed2cb78188712250006fa4b1
12m25s	e6222d14c44eaff476afa89faf662a488e3575c7

Спустя полгода



За все время использования скорость сборки почти не изменилась. Делалось несколько регенераций rch. Об этом несколько подробнее.

Для генерации нужно делать замеры на чистой сборке, без rch. Т.к. туда попадают внутренние хедера, то происходит некоторое смещение и каждой единице трансляции становится видно чуть больше чем раньше. При компиляции без rch те места, где символы попадали из rch, начинают отваливаться. Такие случаи пока что приходится править руками. Хорошо что не часто, раз в пару месяцев или после крупных изменений.

Что дальше ?

Поддерживать инклюды все же сложновато. Вся надежда остается на модули. Пока что нет полноценной поддержки со стороны cmake ([C++ modules support?](#)), поэтому о результатах говорить рано. Все равно нужно будет переработать код и нет полных гарантий что это будет сильно быстрее rch, но зато можно надеяться что после правильной разбивки регресс будет происходить значительно реже.

- [Analyze your builds programmatically with the C++ Build Insights SDK](#)
- [C++ Build Insights SDK samples](#)
- [Faster builds with PCH suggestions from C++ Build Insights](#)
- [Finding build bottlenecks with C++ Build Insights](#)
- [LLVM](#)
- [Include What You Use \(IWYU\)](#)