

# УСКОРЯЕМ СБОРКУ C++ ПРОЕКТОВ. ПРАКТИКА ИСПОЛЬЗОВАНИЯ UNITY-СБОРОК

ЛАПИЦКИЙ АРТЁМ

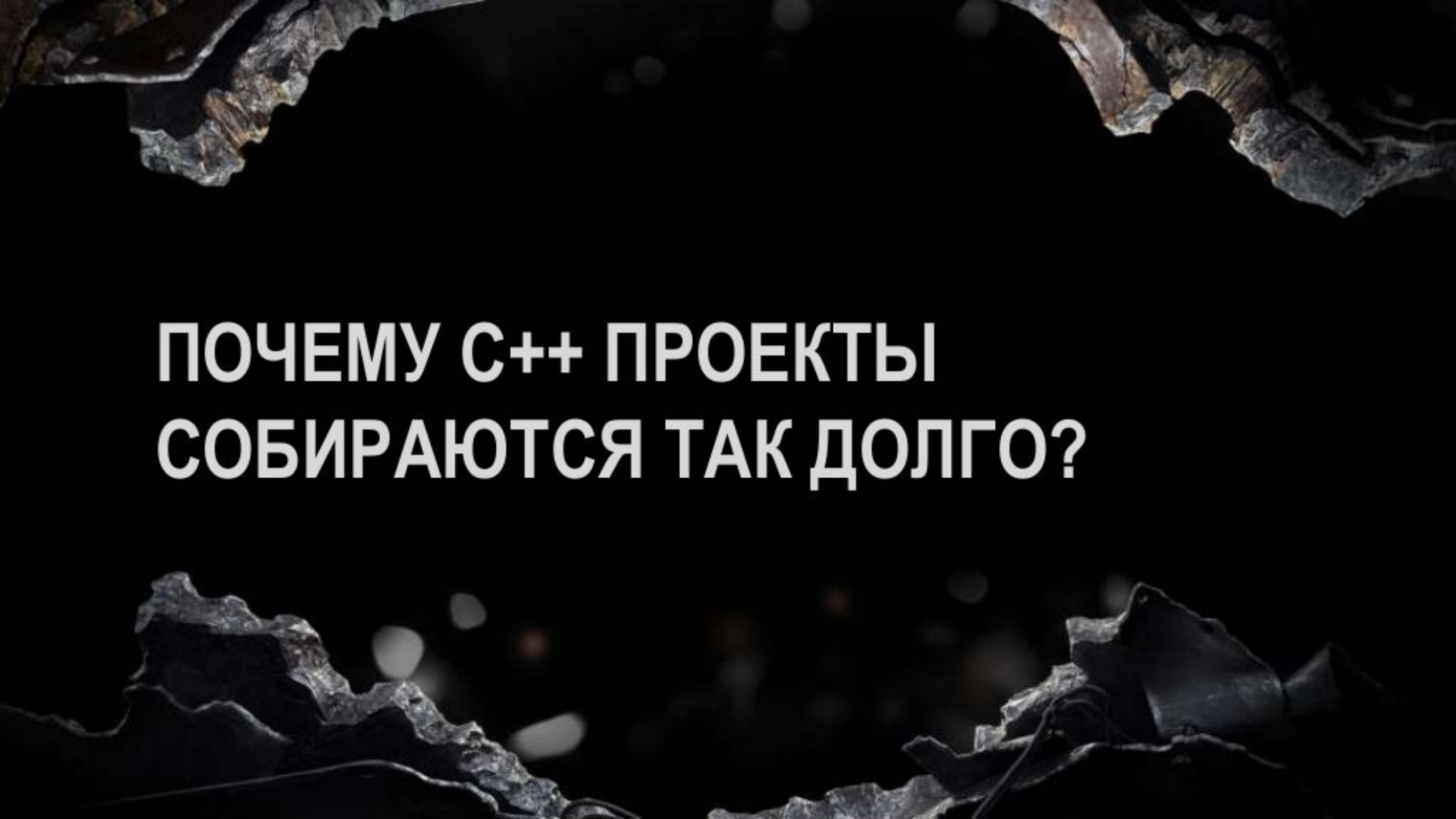
LAPITSKY.ARTEM@GMAIL.COM



WARGAMING.NET  
LET'S BATTLE

# О ЧЁМ БУДЕМ ГОВОРИТЬ

- Почему C++ проекты собираются так долго?
- Какие есть способы ускорения сборки C++ кода
- Что такое Unity сборки и как они работают
- Практика использования Unity сборок



**ПОЧЕМУ C++ ПРОЕКТЫ  
СОБИРАЮТСЯ ТАК ДОЛГО?**

# ЧЕМ ЗАНИМАЕТСЯ КОМПИЛЯТОР ПРИ СБОРКЕ КОДА

- › Bjarne Stroustrup “The C++ Programming Language. Special Edition”, 2017

hello.cpp

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```



## ЧЕМ ЗАНИМАЕТСЯ КОМПИЛЯТОР ПРИ СБОРКЕ КОДА (2)

### ➤ Результат препроцессинга G++

```
$ gcc hello.cpp -E > hello.gcc.preprocessed.cpp
$ stat -x hello.gcc.preprocessed.cpp
  File: "hello.gcc.preprocessed.cpp"
  Size: 1233499      FileType: Regular File
$ cloc hello.gcc.preprocessed.cpp
  1 text file.
  1 unique file.
  0 files ignored.
```

```
github.com/AlDanial/cloc v 1.74 T=0.25 s (4.0 files/s, 152999.5 lines/s)
```

Language	files	blank	comment	code
C++	1	6824	0	30995

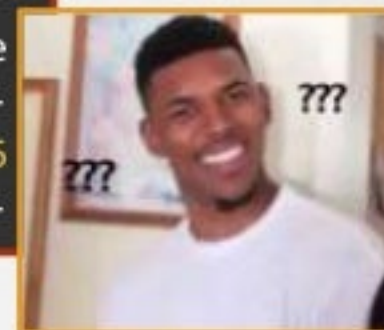
## ЧЕМ ЗАНИМАЕТСЯ КОМПИЛЯТОР ПРИ СБОРКЕ КОДА (3)

### ➤ Результат препроцессинга MSVC++

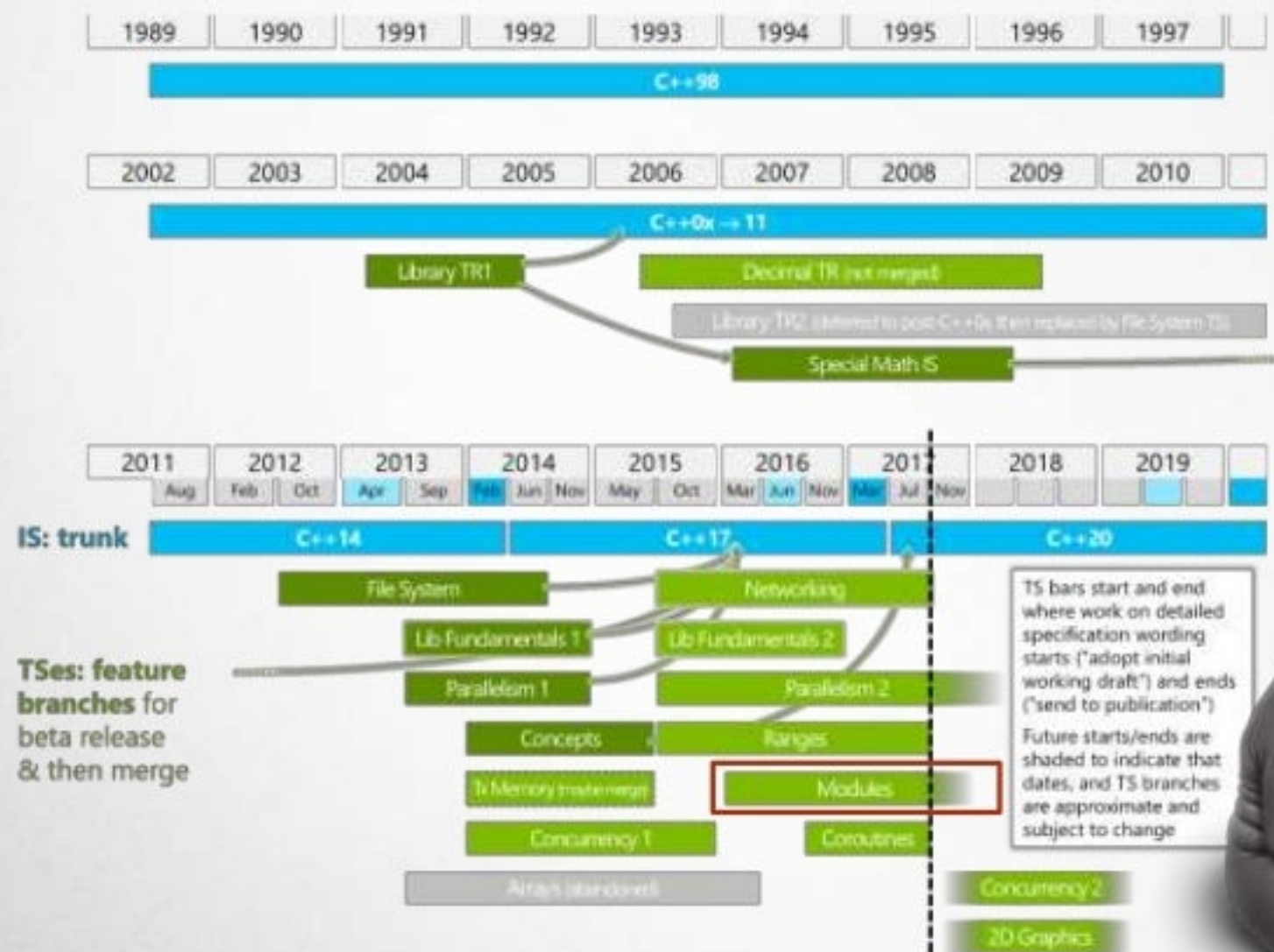
```
> cl /E hello.cpp > hello.vc.preprocessed.cpp
> cloc hello.vc.preprocessed.cpp
    1 text file.
    1 unique file.
    0 files ignored.
```

```
github.com/AlDanial/cloc v 1.74 T=0.26 s (3.8 files/s, 192740.0 lines/s)
```

Language	files	blank	comment	code
C++	1	23331	0	27286



# C++ И МОДУЛИ





## ОСОБЕННОСТИ ПРОЦЕССА ТРАНСЛЯЦИИ C++ КОДА

- Каждый этап трансляции полностью зависит от результата предыдущих
- Трансляция unit'а не может быть распараллелена
- Требуется минимум 3 прохода по исходному коду





# МОДУЛЬНОСТЬ ЧЕРЕЗ ЗАГОЛОВОЧНЫЕ ФАЙЛЫ

- `#include` делает текстовую подстановку содержимого заголовочного файла
- Смысл подставленного текста полностью зависит от всего, что было в исходном файле до этого

```
#include <foo.h>

...

#define true (bool(__LINE__ % 2))

#include <foo.h>
```

## МОДУЛЬНОСТЬ ЧЕРЕЗ ЗАГОЛОВОЧНЫЕ ФАЙЛЫ (2)

- Невозможно использовать повторно результат обработки заголовочного файла (даже в рамках одного translation unit)
- Лишние `#include` директивы создают массу бессмысленной работы для компилятора



# TEMPLATE METAPROGRAMMING

- Выполнение метапрограмм может занимать значительную часть времени трансляции
- Инстанцирование шаблонов происходит в каждом translation unit отдельно (используйте extern template)







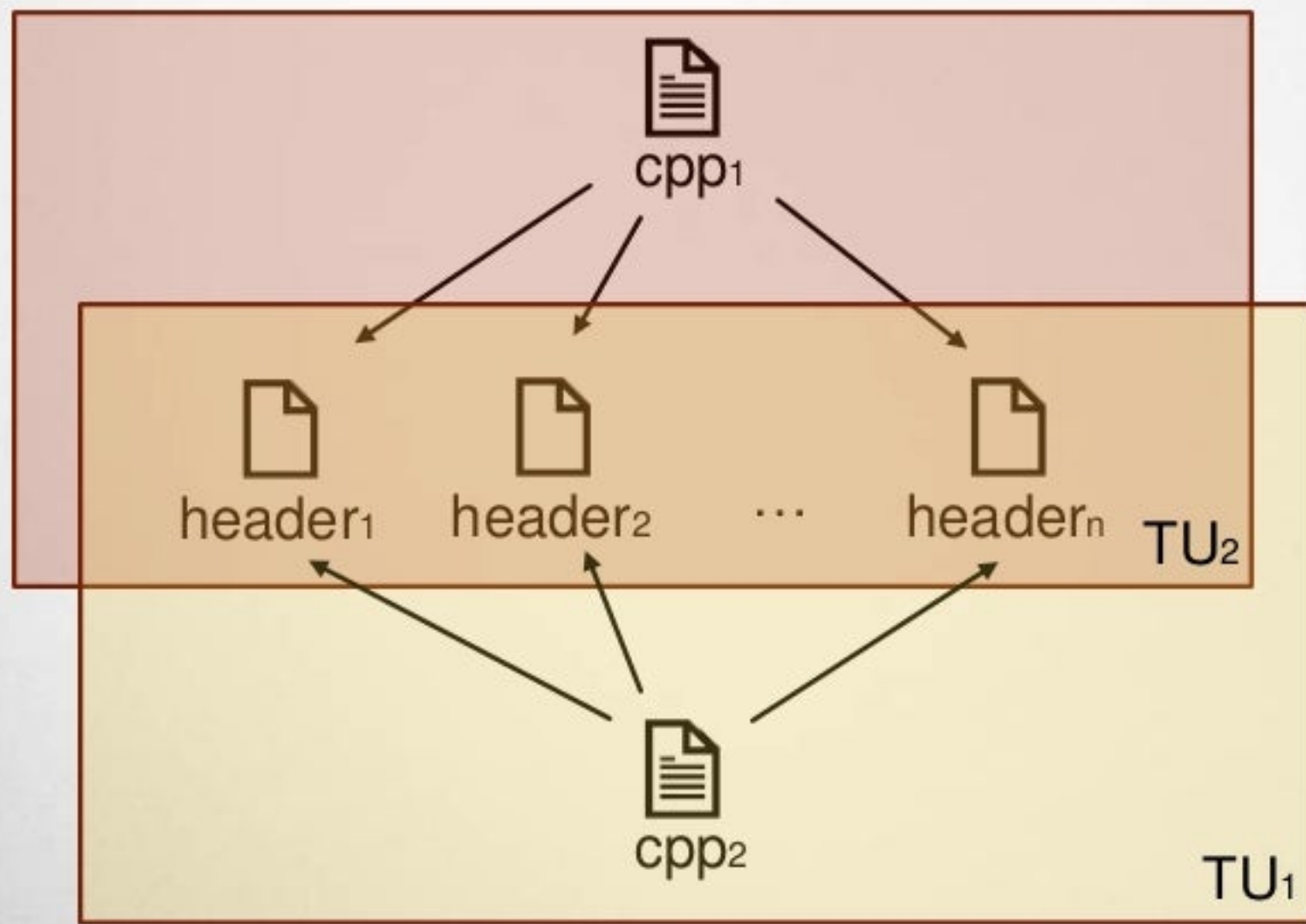
# **СПОСОБЫ УСКОРЕНИЯ СБОРКИ**

## PRECOMPILED HEADERS (PCH)

- › Специальный тип заголовочного файла
  - › Исключена зависимость от контекста включения
  - › Повторное включение заголовочных файлов, включённых в PCH не должно влиять на смысл программы
  - › Требуется одинаковый набор параметров компилятора для всех единиц трансляции, которые используют PCH
- › Их можно компилировать!

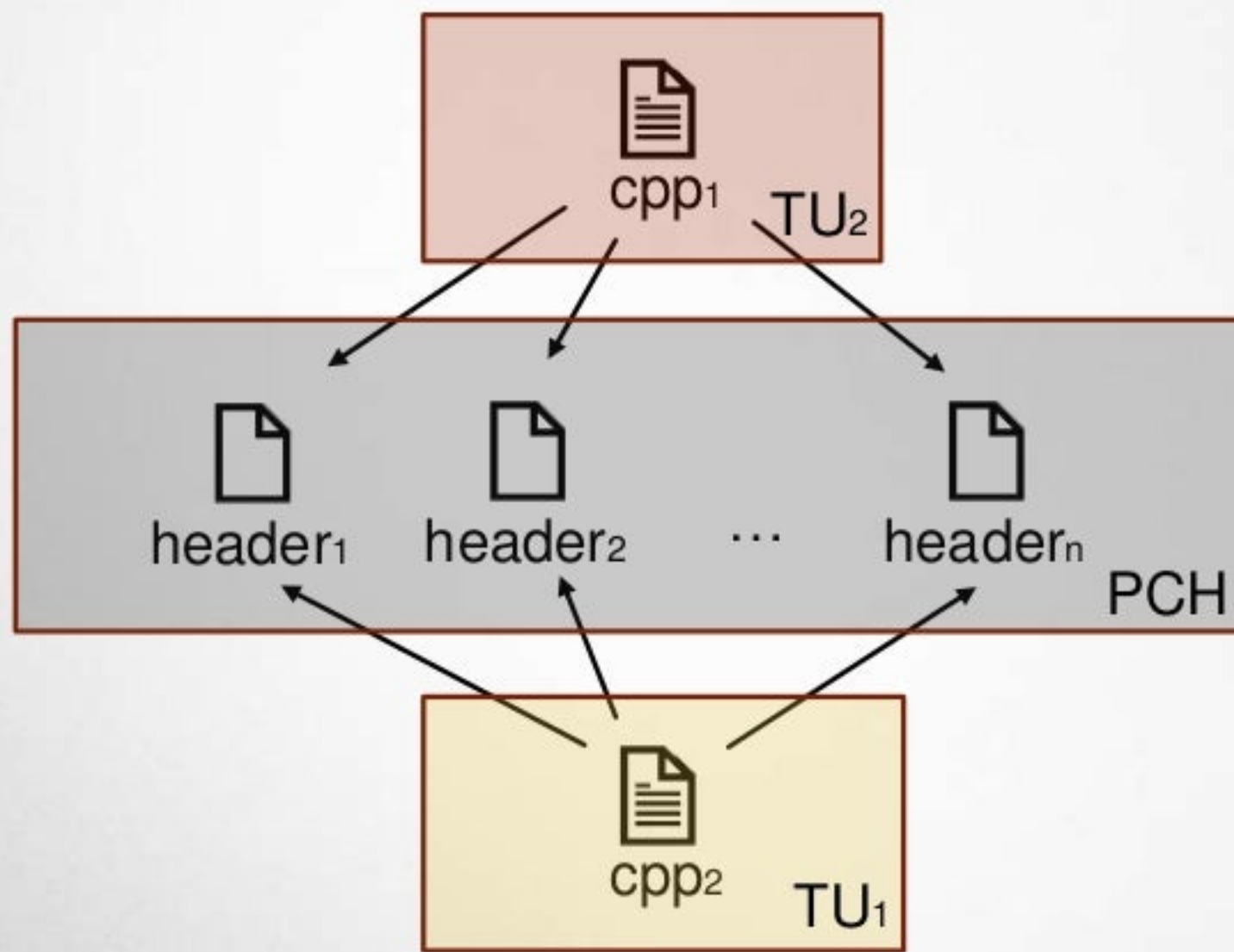


# PRECOMPILED HEADERS (PCH)





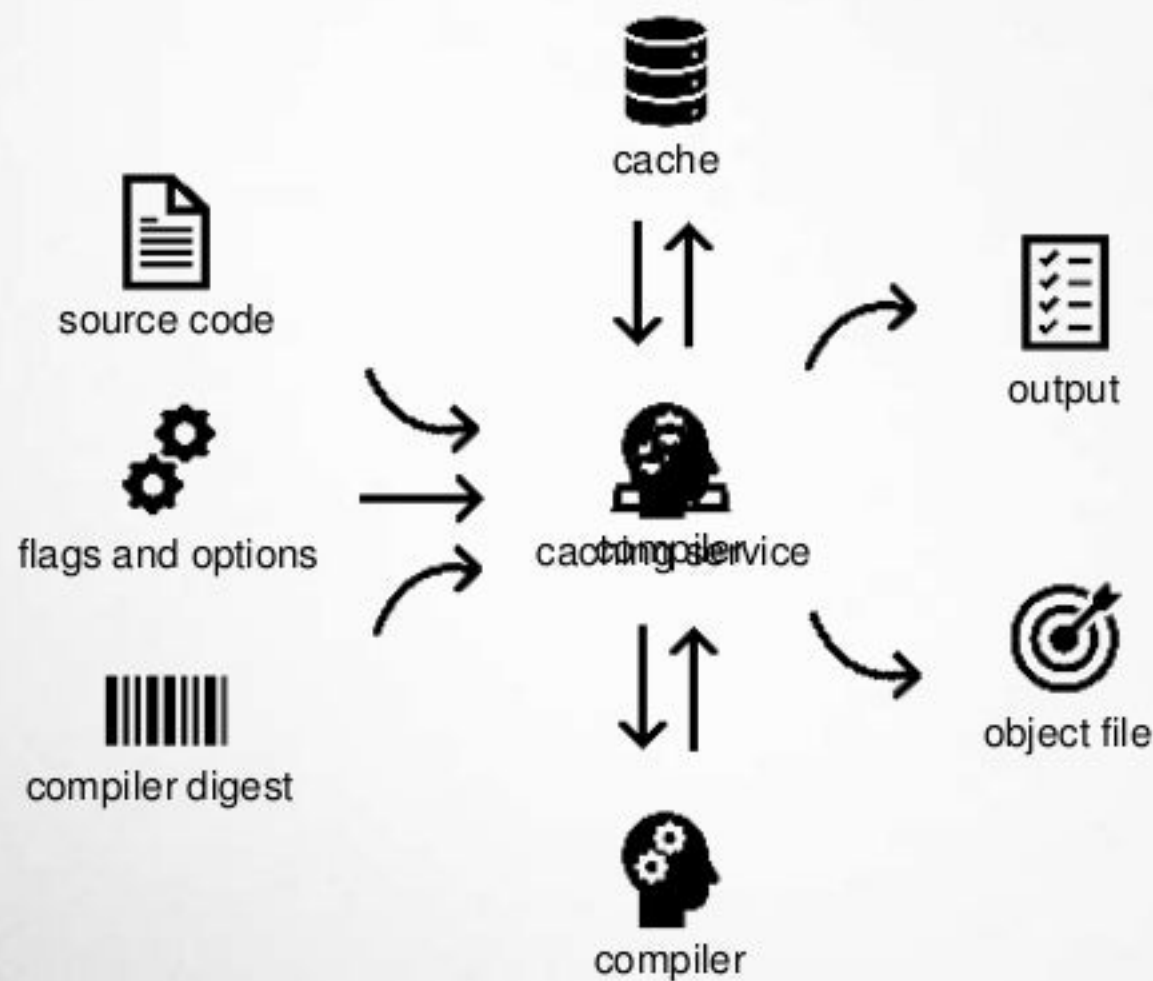
# PRECOMPILED HEADERS (PCH)



# PRECOMPILED HEADERS (PCH)

- Ограничения и особенности
  - Каждый translation unit может использовать ровно один PCH
  - Нужно быть внимательным к различиям директив препроцессора разных translation unit
  - Требуется внесения изменений в структуру и параметры сборки проекта
  - Нюансы реализации в разных компиляторах

# КЭШИРОВАНИЕ ОБЪЕКТНЫХ ФАЙЛОВ





## КЭШИРОВАНИЕ ОБЪЕКТНЫХ ФАЙЛОВ (2)

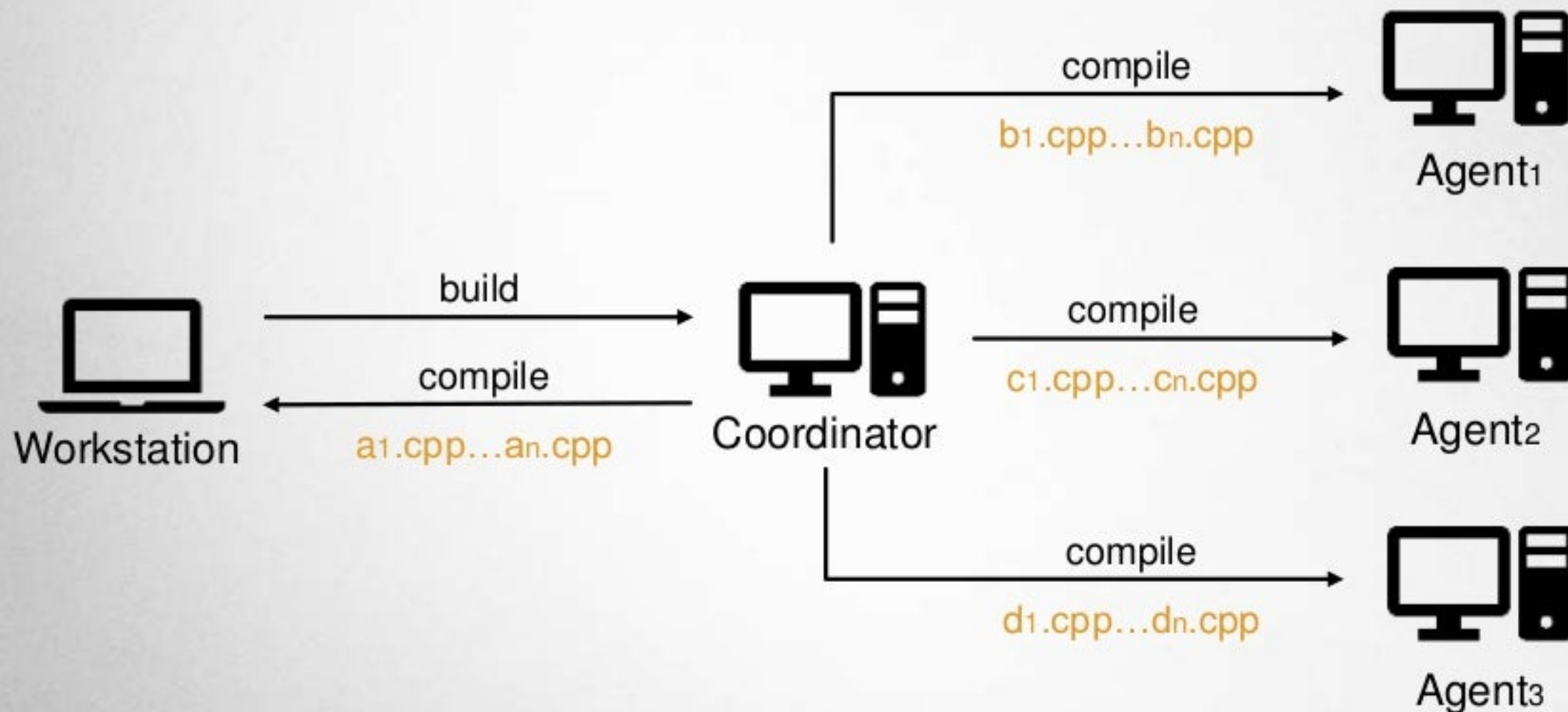
- Для каждого набора входных данных компилятора сохраняется результат его работы
- Схема очень эффективна, когда большая часть транслируемого кода не меняется
- Вторая и последующие сборки до 10-ти раз быстрее первой



## КЭШИРОВАНИЕ ОБЪЕКТНЫХ ФАЙЛОВ (3)

- Особенности и ограничения:
  - Первоначальная сборка всегда дольше сборки без кэширования
  - Необходимость повторной сборки при изменении даже незначительных параметров компиляции
  - Иногда кэшируются «битые» объектные файлы
  - Плохо работает с механизмом RCH

# РАСПРЕДЕЛЁННАЯ СБОРКА





## РАСПРЕДЕЛЁННАЯ СБОРКА (2)

- Особенности и ограничения:
  - Эффективность сильно зависит от конкретной реализации и конфигурации сети
  - Некорректная работа какого-либо из агентов не позволяет успешно собрать проект
  - Отладка проблем сборки становится сложнее





**UNITY BUILD**

## UNITY BUILD

- **Идея:** максимально уменьшить количество компилируемых исходных файлов



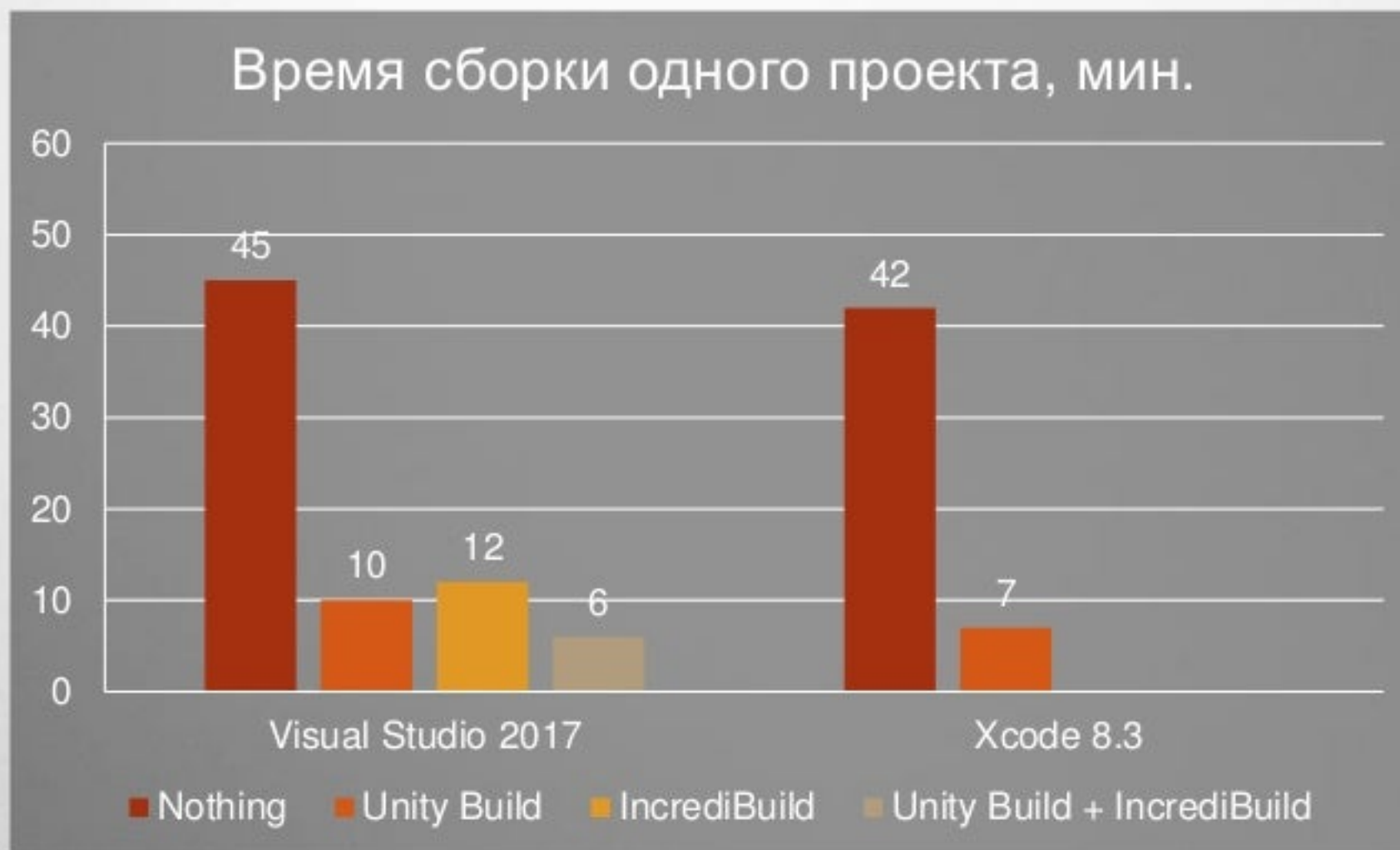


## UNITY BUILD (2)

*The Dalai Lama walks into a pizza shop and says  
"can you make me one with everything?"*

- Объединим все исходные .crr файлы в один: Unity pack
  - Исключён повторный разбор и анализ одних и тех же заголовочных файлов
  - Не инстанцируются по многу раз одни и те же шаблоны
  - Сильно упрощается процесс линковки

# UNITY BUILD (3)



# UNITY BUILD: BEFORE

Your project

## Headers



hdr1.h



hdr2.h

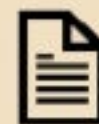


hdr3.h

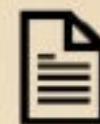
## Sources



src1.cpp



src2.cpp



src3.cpp



# UNITY BUILD: AFTER

Your project unity project

## Headers



hdr1.h



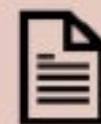
hdr2.h



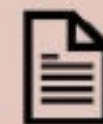
hdr3.h



src1.cpp



src2.cpp



src3.cpp

## Sources



unity\_pack.cpp

## unity\_pack.cpp

```
#include "src1.cpp"  
#include "src2.cpp"  
#include "src3.cpp"
```



## UNITY BUILD: USE EFFECTIVELY

- Использование только одного Unity Pack'а неэффективно
  - Современные CPU имеют более одного ядра и несколько Unity pack'ов могут собираться параллельно
  - Слишком большой Unity pack может вызвать ошибку “out of memory” или привести к неэффективности процесса компиляции
- Множество исходных файлов следует рассредоточить по нескольким Unity pack'ам

# UNITY BUILD: AUTOMATE IT!

- › Генерацию Unity pack'ов стоит автоматизировать
  - › Рабочее решение на CMake  
<https://github.com/dava/dava.engine/blob/development/Sources/CMake/Modules/UnityBuild.cmake>
  - › Reducing Compilation Time: Unity Builds © Christoph Heindl  
<https://cheind.wordpress.com/2009/12/10/reducing-compilation-time-unity-builds/>



## UNITY BUILD: AUTOMATE IT! (2)

### ➤ Пример проекта на CMake

#### CMakeLists.txt

```
...  
set (HEADER_FILES hdr1.h hdr2.h hdr3.h)  
set (SOURCE_FILES src1.cpp src2.cpp src3.cpp)  
  
if (UNITY_BUILD)  
    # generate unity pack files and add them to ${SOURCE_FILES}  
    generate_unity_packs(MyProject SOURCE_FILES)  
endif()  
  
add_executable(MyProject ${HEADER_FILES} ${SOURCE_FILES})
```



# **ПРАКТИКА ИСПОЛЬЗОВАНИЯ UNITY-СБОРОК**

# АДАПТАЦИЯ ПРОЕКТА К UNITY-СБОРКЕ

- › Необходимы модификации исходного кода
  - › Не использовать глобальные static переменные
  - › Не использовать анонимные namespace
  - › Ограничить использование директивы using namespace
  - › Для всех #define необходимы соответствующие #undef





# STATIC И АНОНИМНЫЕ NAMESPACE

src1.cpp

```
...
static int my_local_var = 0;

namespace {
    void my_local_function() {
        std::cout << "src1\n";
    }
}

int src1_process() {
    my_local_var = 1;
    my_local_function();
}
```

src2.cpp

```
...
static int my_local_var = 0;

namespace {
    void my_local_function() {
        std::cout << "src2\n";
    }
}

int src2_process() {
    my_local_var = 2;
    my_local_function();
}
```



unity\_pack.cpp

## STATIC И АНОНИМНЫЕ NAMESPACE (2)

src1.cpp

```
...
namespace Ssrc1 {
    int my_local_var = 0;
    void my_local_function() {
        std::cout << "src1\n";
    }
}

int src1_process() {
    using namespace Ssrc1;

    my_local_var = 1;
    my_local_function();
}
```

src2.cpp

```
...
namespace Ssrc2 {
    int my_local_var = 0;
    void my_local_function() {
        std::cout << "src2\n";
    }
}

int src2_process() {
    using namespace Ssrc2;

    my_local_var = 2;
    my_local_function();
}
```

unity\_pack.cpp

# USING NAMESPACE

src.cpp

```
...  
#include <vector>  
  
using namespace std;  
  
int foo() {  
    vector<int> numbers;  
}  
  
int bar() {  
    vector<string> strings;  
}
```



src.cpp

```
...  
#include <vector>  
  
int foo() {  
    using namespace std;  
    vector<int> numbers;  
}  
  
int bar() {  
    using namespace std;  
    vector<string> strings;  
}
```





## АДАПТАЦИЯ ПРОЕКТА К UNITY-СБОРКЕ (2)

- Некоторые исходные файлы придётся вынести из unity pack'ов
  - если используются специфические системные API
  - если в проекте есть конфликтующие внешние зависимости
- Процесс адаптации можно автоматизировать



## РАЗРАБОТКА В UNITY-ПРОЕКТЕ

- Благодаря небольшому и предсказуемому времени сборки кажется хорошей идеей, но
  - это ведёт к беспорядку с include'ами
  - не все IDE справляются с разбором и анализом unity pack'ов
  - изменение в исходном файле (.cpp) — это пересборка одного unity\_pack'a
  - изменение в заголовочном файле (.h) — это пересборка 1..N unity\_pack'ов

## РАЗРАБОТКА В UNITY-ПРОЕКТЕ. ПОЛУ-UNITY

- Указанные проблемы решаются использованием механизма избирательного исключения из Unity сборки тех частей проекта, в которых будет вестись активная разработка

UnityIgnoreList.cmake

```
...  
set (PERSONAL_IGNORES "Dir1/" "Dir2/" "Dir3/")
```



# SUMMARY

- › За
  - › Огромный прирост скорости компиляции и компоновки
  - › Одинаково высокая скорость как первой, так и последующих сборок
  - › При адаптации кода вырезается copy-paste

## SUMMARY (2)

- Против
  - Требуется модификации исходного кода
  - Накладывает ограничение на использование возможностей C++
  - Требуется большой объём оперативной памяти при сборке

**СПАСИБО ЗА ВНИМАНИЕ!**



**WARGAMING.NET**  
LET'S BATTLE



# ANY QUESTIONS?

## ЛАПИЦКИЙ АРТЁМ

Software Developer



lapitsky.artem@gmail.com



<https://www.facebook.com/WargamingMinsk>



<https://www.linkedin.com/company/wargaming-net>

[wargaming.com](http://wargaming.com)

