# NO MORE COFFEE BREAKS

## REDUCING C++ PROJECT BUILD TIME

Piotr Osiewicz @ code::dive 2019

osiewicz.github.io/cd-builds

# WHO AM I?

- C++ enthusiast (with weak spot for C)
- Software engineer @ security sector
- Cracow
- Taking photos of all kinds of things (mostly buildings)

# C/C++ BUILD PROCESS IN A NUTSHELL

1. Preprocessing
2. Compilation
3. Linking

# GLOSSARY

- Translation unit
- Object file
- Symbol

# SYMBOLS

| | **Weak symbol** | **Strong symbol** |
|---|---|---|
| Weak symbol | Pick any, discard the rest | Pick strong symbol |
| Strong symbol | Pick strong symbol | ODR violation (link time error) |

# IMPLICIT TEMPLATE INSTANTIATION

Takes place at usage site when compiler must know the complete type.

Can occur multiple times in whole program (introduces a **weak** symbol).

```
// Implicitly instantiate std::array<3,int>
array<int, 3> myGlobalArray;
// Implicitly instantiate std::array<5,int>
static_assert(sizeof(array<int, 5>) == sizeof(int[5]));
// Does not instantiate the std::array<int, 8> - why?
static_assert(sizeof(array<int, 8>*) == sizeof(double*));
```

# EXPLICIT TEMPLATE INSTANTIATION

Forces instantiation of union, class or function it refers to.

Explicit template instantiation can appear only once in whole program (introduces a **strong** symbol).

```
template void foo<int>(int);
```

# TEMPLATES

# REDUNDANCY IN TEMPLATE USAGE

Function templates can be instantiated **multiple times** in various TU`s when one instantiation would be enough

# VANILLA WAY TO CREATE A PLAIN FUNCTION

1. Put function **declaration** in header file (included by external function callers)
2. Put function **definition** in .cpp file
3. Any calls to the function are **resolved at link time**

# VANILLA WAY TO CREATE A FUNCTION TEMPLATE

1. Put function template **definition** in header file
2. Caller TU **#include** and instantiate function template with some argument set - there can be **many** implicit instantiations in whole program
3. Since implicit instantiations of function templates are **weak** symbols, **excessive** definitions are removed at link time

# CAN WE DO ANY BETTER?

1. Put function template **declaration** in header file
2. Put function template **definition** in .cpp file
3. **Explicitly** instantiate function templates with any argument sets used in your codebase

**LIBRARY MAINTAINERS RIGHT NOW:**

(┛ಠ_ಠ)┛彡┻━┻

This is a **partial solution** to the stated problem

## IT IS NOT CONVENIENT

- All argument sets must be known & explicitly stated in "instantiating" TU beforehand (**can't rely** on implicit instantiation & introduces **additional dependencies**)
- **Constrains** usage of template by external user (probably not a good way to ship a template as a part of library)

# IT IS NOT SO BAD EITHER

- Good for internal development when argument sets are finite & instantiations wouldn't introduce dependency spaghetti

I call it "optimization of **implementation**" - it is done by the code maintainer

# OPTIMIZATION OF USAGE

C++11 introduced **extern templates** (explicit instatiation declaration)

```
// Compiler will skip implicit instantiation
// of foo<int> in this TU (after this line)
// leaving it up for linker resolution.
// Explicit instantiation of foo<int> must be
// done in other TU.
extern template int foo<int>(int);
```

## UPDATED ALGORITHM

1. Put function template **definition** in header file
2. **Prevent** implicit instantiations in N - 1 out of N TUs that use given template with arbitrary argument set
3. **Explicitly** instantiate template in **arbitrarily** chosen TU

IMO the only issue (🍰) is that it's not clear which TU should explicitly instantiate the template

## THE CAKE IS A LIE! 🍰

## IT'S NOT THE ONLY ISSUE!

Both approaches lead to loss of **inline expansion** potential - this optimization technique must have function body in same TU as caller. In worst case all N calls now have **worse performance**

## IS CAKE A LIE? 🍰

**LTO** enables **cross-TU inlining**, hence nothing is really lost.

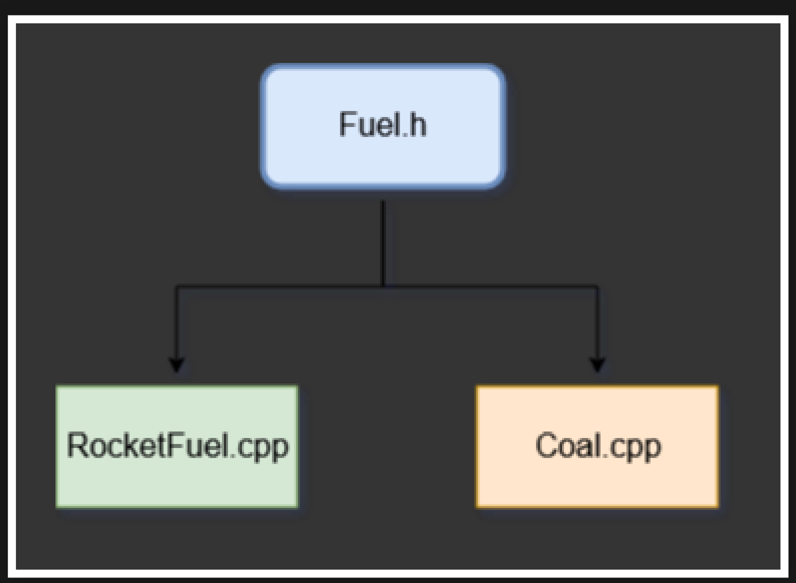
## TO RECAP - EXTERN TEMPLATE IS NICE

- Opt-in user-facing mechanism
- Not all N - 1 TUs have to use it - it's **totally** fine if you forget

# HEADER FILES

## REDUNDANCY CAUSED BY HEADER FILES

- Dependency of my dependency is my dependency
- Parse it one more time (context dependency)


Fuel.h
RocketFuel.cpp
Coal.cpp

## PRECOMPILED HEADERS

- Store preprocessed header file for use by several translation units

# SINGLE TRANSLATION UNIT*

*Also known as Jumbo build or Unity build

```
// stu.cpp - the only file passed to the toolchain
#include "tu1.cpp"
#include "tu2.cpp"
...
#include "tuOver9000.cpp"
```

## HEADER FILES SHARED BY #INCLUDED TUS IN STU ARE PARSED ONLY ONCE*

* assuming presence of include guards

## THERE ARE ALSO MORE INLINING OPPORTUNITIES

# HOWEVER...

```cpp
// tu19.cpp
// The following might cause linker errors under STU while
// being perfectly valid on it's own
static int quickly_find_meaning_of_life() {
    return 42;
}
...
```

# THINK ABOUT THE TRANSITIVITY OF YOUR HEADERS

```cpp
// a.h
#include "SomeSuperClass.h"
int do_stuff(const SomeSuperClass& value);
...
// a.cpp
#include "a.h"
int do_stuff(const SomeSuperClass& value){
        return value.member;
}
```

```cpp
// a.h
class SomeSuperClass;
int do_stuff(const SomeSuperClass& value);
...
// a.cpp
#include "SomeSuperClass.h"
int do_stuff(const SomeSuperClass& value){
        return value.member;
}
```

include-what-you-use

# KNOW YOUR TOOLS

# PARALLELIZE YOUR BUILD (LOCALLY)

# DISTRIBUTE THE BUILD

# CACHE ALL THE ARTIFACTS!

# INCREMENTAL LINKING IS A WIN!

lol nope

## INCREMENTAL LINKING MIGHT ADD PADDING TO THE BINARY TO ALLOW PATCHING

Worse performance & bigger binary

# MODULES

lol nope

I HAVE NO EXPERIENCE WITH THEM

BUT THEN AGAIN - WHO DOES?

# THE END

## THANKS TO:

- Mateusz Dudek 🐦
- Mateusz Gacek (@mgacek8) 🦇
- Bartosz Bryk 🍔