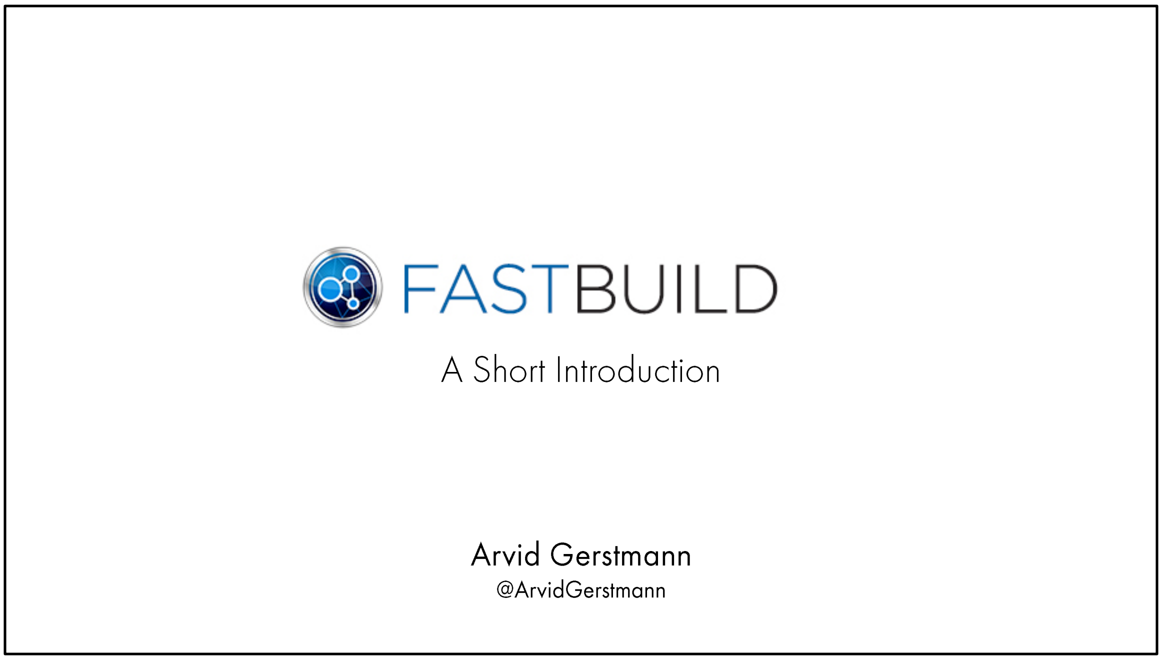


# FASTBUILD

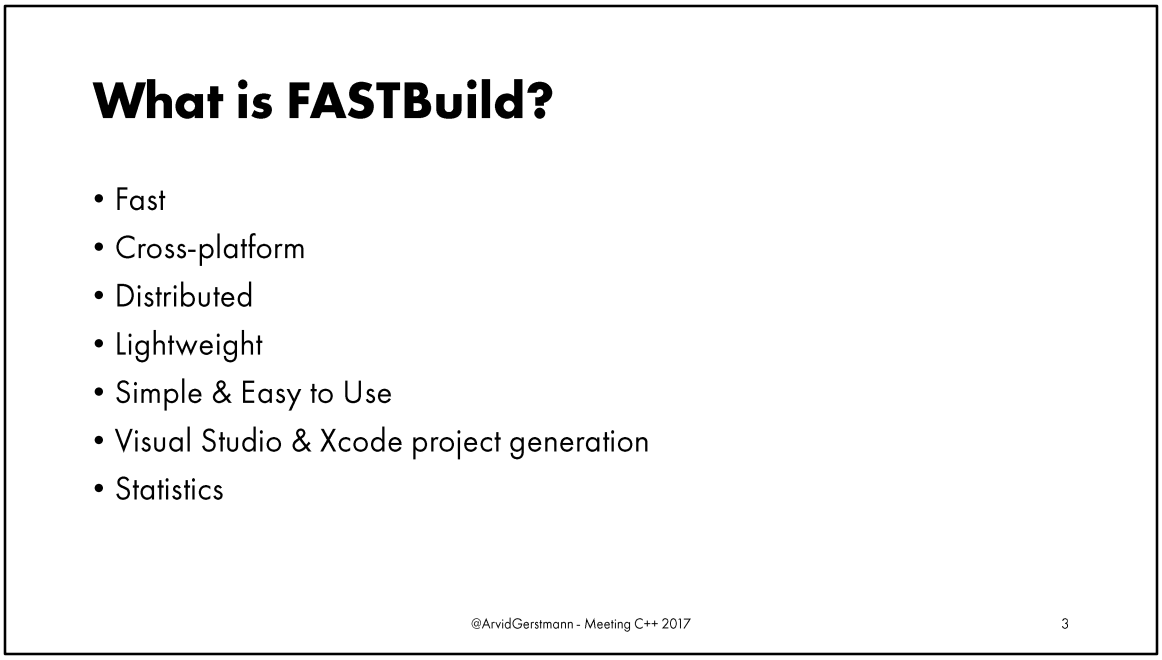
A Short Introduction

Arvid Gerstmann
@ArvidGerstmann

Hi, my name is Arvid Gerstmann. And this is FASTBuild: A Short Introduction.
Let‘s get started!

## What is FASTBuild?

What is FASTBuild, well, like the name probably already gave away, it‘s a build system.



## What is FASTBuild?

- Fast
- Cross-platform
- Distributed
- Lightweight
- Simple & Easy to Use
- Visual Studio & Xcode project generation
- Statistics

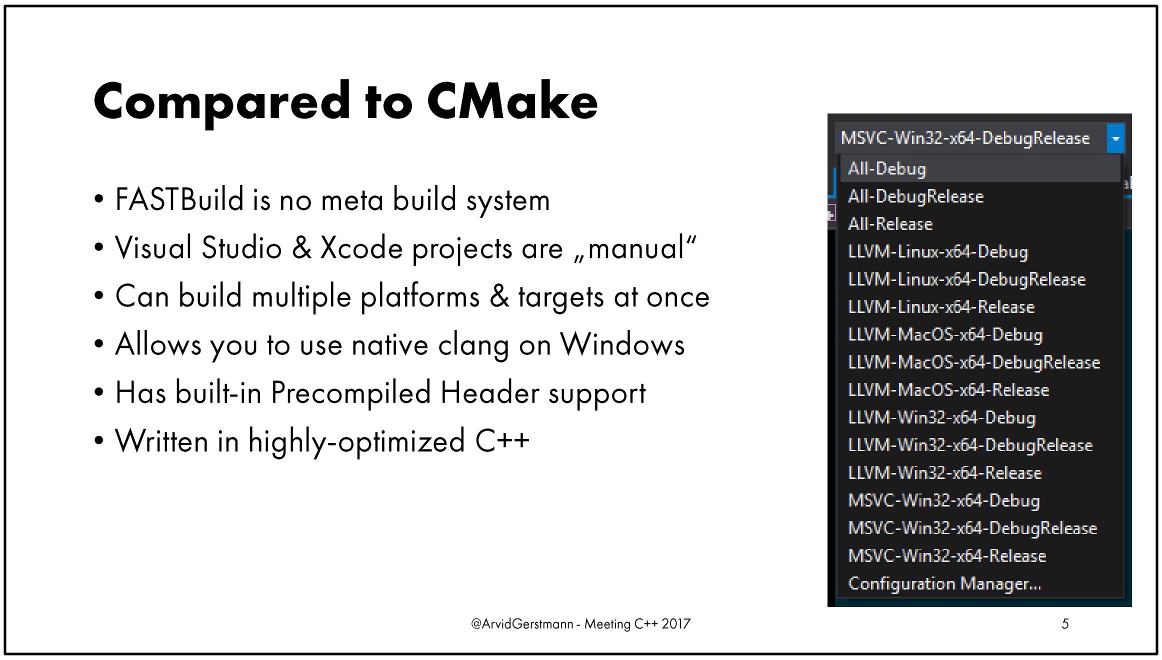
@ArvidGerstmann - Meeting C++ 2017 3

**Fast.** FASTBuild, uses parallel compilation and an, optional, object cache to speed up compilation times. While Unity/blob are providing another built-in mechanism to speeding up your build.
**Cross-platform.** It has great support for all major operating systems. Both as a target and host. It can target all major operating systems and consoles, using any major compiler (gcc, clang, msvc, snc, GreenHills, CodeWarrior & NVCC).
**Distributed**. FASTBuild has a separate worker component, which allows to distribute compilation steps over the network.
**Lightweight**. FASTBuild is a single, 600kb, self-contained, executable, with no external dependencies. Installation consist of simply putting it into your path. It's perfect to be checked-in along with all your sources, for more on that, come to my lightning talk in session 3.
**Simple & Easy to use.** FASTBuild employs a simple, yet powerful, DSL to configure builds. It's easy to pick-up and understand.
**Visual Studio and Xcode project generation**. Built-in support to create projects for popular IDEs, more on that later. Build output is fully Visual Studio compatible, giving you the same error integration as if you were compiling using Visual Studio itself.
**Statistics.** Ever wondered what is slowing down your build? Wonder no more, FASTBuild can generate detailed HTML build reports during compilation.

## Compared to CMake

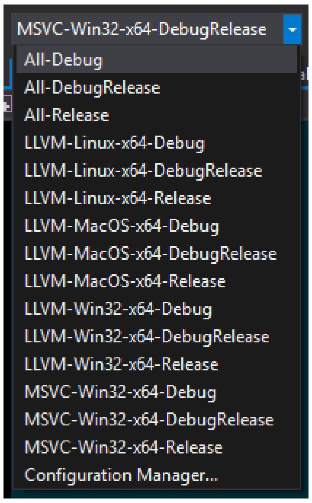
@ArvidGerstmann - Meeting C++ 2017 4

Alright, enough buzzwords. To give you a little better overview, I'll quickly compare FASTBuild to another famous build systems: Cmake



## Compared to CMake

- FASTBuild is no meta build system
- Visual Studio & Xcode projects are „manual“
- Can build multiple platforms & targets at once
- Allows you to use native clang on Windows
- Has built-in Precompiled Header support
- Written in highly-optimized C++



@ArvidGerstmann - Meeting C++ 2017 5

**FASTBuild is no meta-build system.** It does not depend on other build-tools, like cmake does. FASTBuild is entirely self-contained, and only requires a compiler to be used.

**The Visual Studio & Xcode project generation is „manual“.** Meaning, that FASTBuild cannot create projects on it‘s own, it needs your assistance. You need to specify the commands to run, where to look for includes and the included projects. However, this allows for a very powerful project creation.

FASTBuild allows you to build for multiple platform or targets at the same time. No restrictions. And if you like, all while never leaving visual studio.

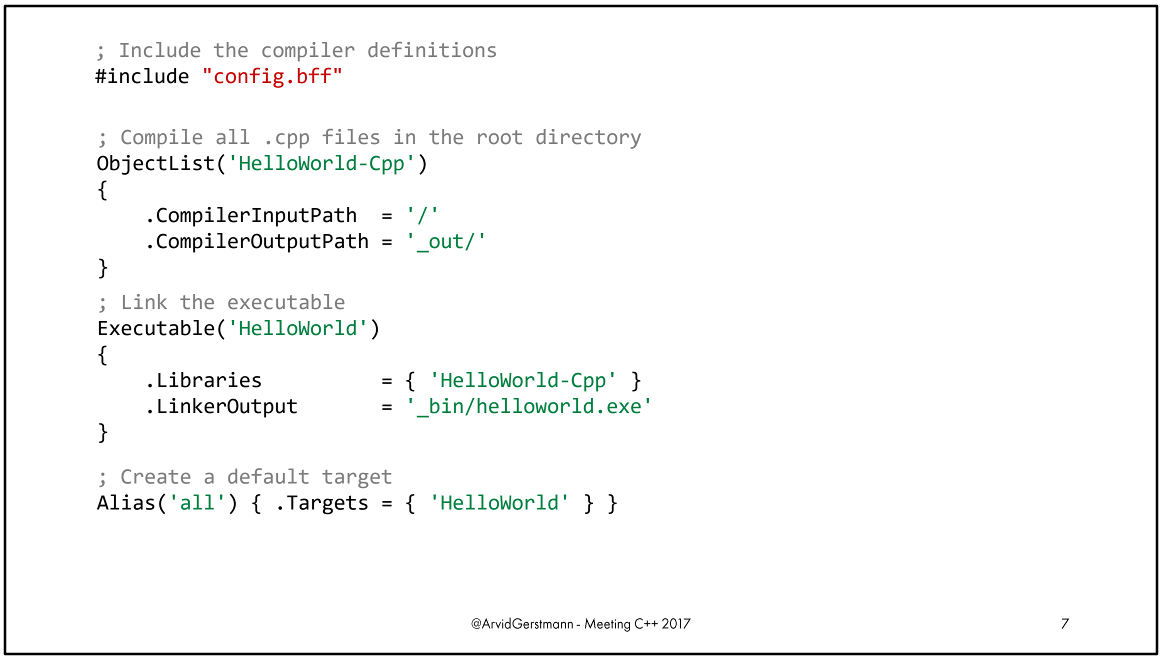
**Allows usage of native clang on windows**, without having to use clang-cl, writing custom cmake toolchains or having to resort to other hacks.

Has **Precompiled header** support built-in, which allows painless use of precompiled headers with any of the supported compilers.

Written in highly optimized C++, in a small and easily hackable code base.

## Getting Started

FASTBuild requires a little bit of boilerplate setup, containing paths to your compilers tools, like CL.EXE and LINK.EXE for the MSVC toolchain, or clang++ and ld for the clang / gcc toolchain.
I'll assume you already have this setup already done, you can get the example project from github at the end of my talk.
Let's create our first target.



```
; Include the compiler definitions
#include "config.bff"

; Compile all .cpp files in the root directory
ObjectList('HelloWorld-Cpp')
{
    .CompilerInputPath  = '/'
    .CompilerOutputPath = '_out/'
}
; Link the executable
Executable('HelloWorld')
{
    .Libraries          = { 'HelloWorld-Cpp' }
    .LinkerOutput       = '_bin/helloworld.exe'
}

; Create a default target
Alias('all') { .Targets = { 'HelloWorld' } }
```

@ArvidGerstmann - Meeting C++ 2017 7

First, we need to include the required compiler definitions. This is done using the „include“ directive. Config.bff contains all compiler flags required to compile an executable or library.

Next, we need to compile our source files into object files. To do that, we use the ObjectList „function“.
While they‘re called „functions“ in FASTBuild terminology, don‘t think of them as actual function calls, but rather as a definitions of targets. FASTBuild internally will build a dependency graph of their usage.

It takes a bunch of arguments, but we can ignore most of them and rely on the defaults. As we can see, it takes an input and output path.
It‘s globbing, recursively, over the input path, looking for files ending in .cpp and compiling them to object files, which are put into _out.

Next, we need to link our objects into an executable, for that, we use the „Executable“ function.
It takes the named ObjectList from the previous step as input, and will output the .exe into _bin.

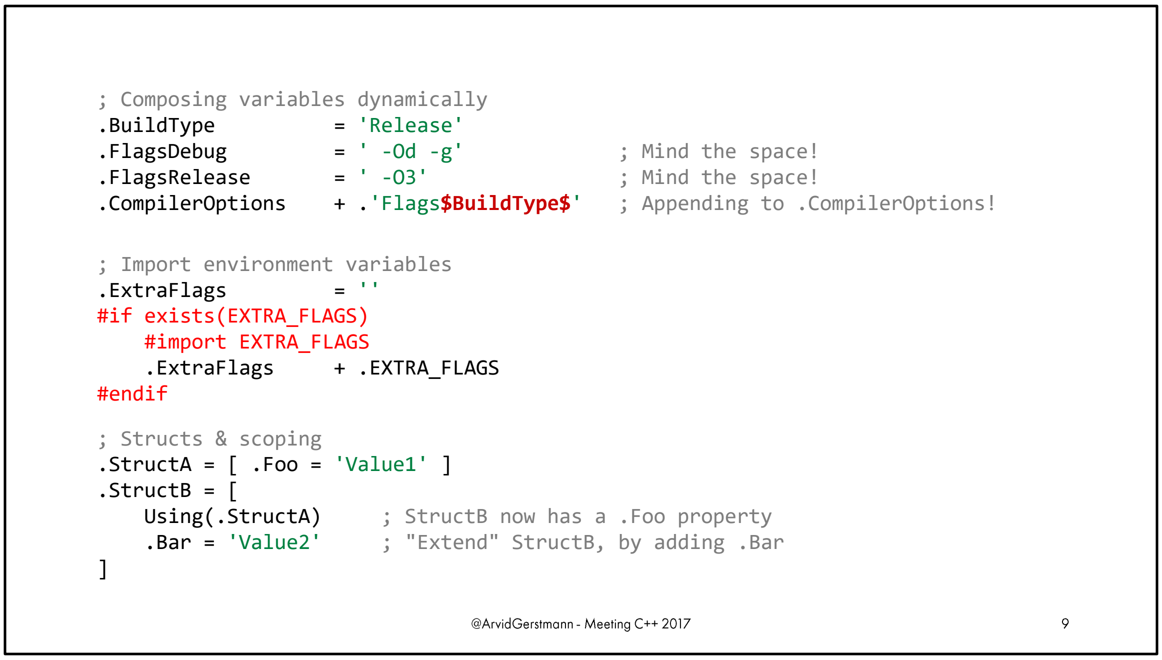
Last but not least, we define the target „all“, which is called by default if we invoke fastbuild on the commandline without any arguments.

**Demo time!**

## Tips & Tricks

@ArvidGerstmann - Meeting C++ 2017 8

We have a little bit of time left, which means I can show you a few tips & tricks, which aren‘t entirely obvious from the get-go, but help tremendously.



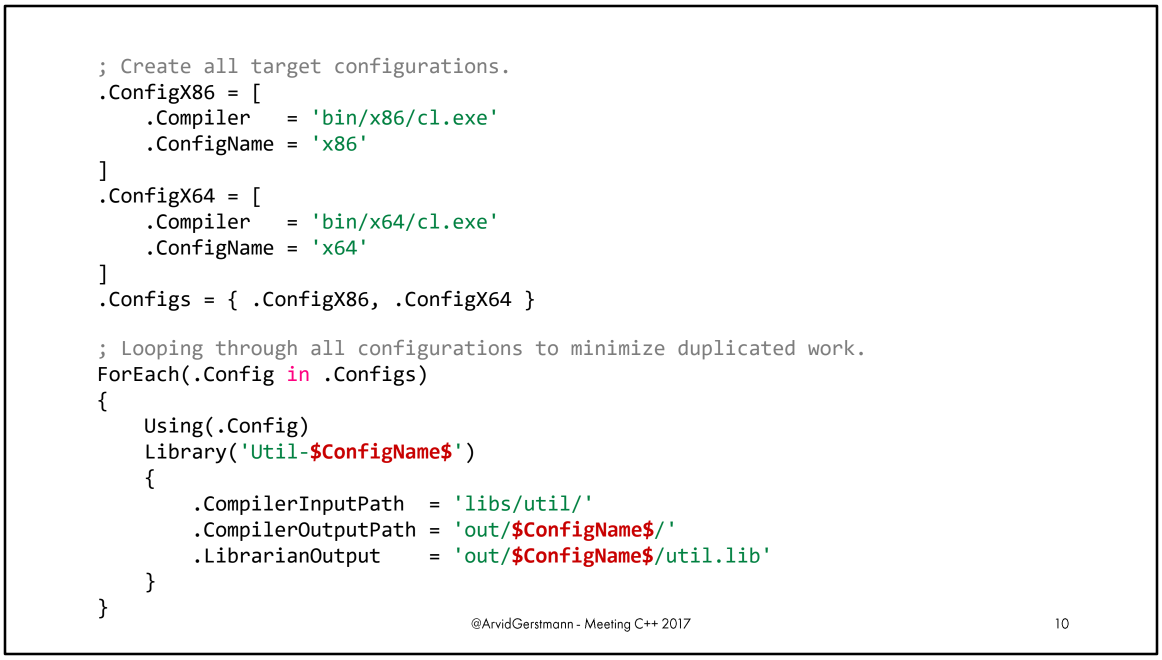
```
; Composing variables dynamically
.BuildType          = 'Release'
.FlagsDebug         = ' -Od -g'            ; Mind the space!
.FlagsRelease       = ' -O3'               ; Mind the space!
.CompilerOptions    + .'Flags$BuildType$'   ; Appending to .CompilerOptions!

; Import environment variables
.ExtraFlags         = ''
#if exists(EXTRA_FLAGS)
    #import EXTRA_FLAGS
    .ExtraFlags     + .EXTRA_FLAGS
#endif

; Structs & scoping
.StructA = [ .Foo = 'Value1' ]
.StructB = [
    Using(.StructA)     ; StructB now has a .Foo property
    .Bar = 'Value2'     ; "Extend" StructB, by adding .Bar
]
```

@ArvidGerstmann - Meeting C++ 2017 9

**Composing variables dynamically.** Since FASTBuilds DSL is lacking an „if“ statement, you need a different way of branching. The following example shows how to dynamically build the compiler flags, based on which buildtype is selected. Don‘t worry, you‘ll get an error when the variable does not exist.

**Import environment variables.** It can be useful to inspect and import environment variables. This is, for example, useful to detect the presence of Visual Studio on Windows, which is done in the first example of this talk and can be seen in the github repository.

**Structs and scoping.** In FASTBuild, you don‘t access the property of a struct directly, but rather bring all it‘s properties into scope by using „Using“. The following example demonstrates that. This is a very powerful technique, which can be used to decrease configuration complexity.



```
; Create all target configurations.
.ConfigX86 = [
    .Compiler   = 'bin/x86/cl.exe'
    .ConfigName = 'x86'
]
.ConfigX64 = [
    .Compiler   = 'bin/x64/cl.exe'
    .ConfigName = 'x64'
]
.Configs = { .ConfigX86, .ConfigX64 }

; Looping through all configurations to minimize duplicated work.
ForEach(.Config in .Configs)
{
    Using(.Config)
    Library('Util-$ConfigName$')
    {
        .CompilerInputPath  = 'libs/util/'
        .CompilerOutputPath = 'out/$ConfigName$/'
        .LibrarianOutput    = 'out/$ConfigName$/util.lib'
    }
}
```

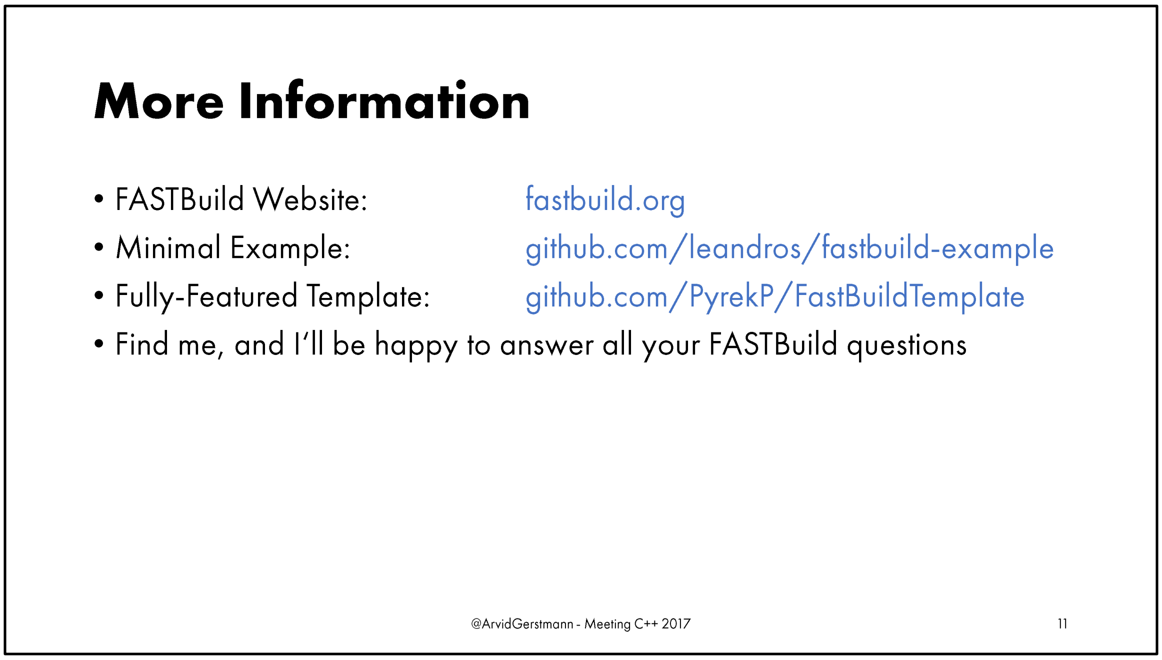
@ArvidGerstmann - Meeting C++ 2017 10

Let‘s see a little more complete example, demonstrating the use of the previously shown techniques.

We‘re creating our target configurations, for the x86 and x86_64 architecture. Keep in mind that .Compiler is a variable taken as an argument by „Library“, we‘ll make use of that later. Next, we create an array containing all configurations ...

... And loop over said array. Inside the „ForEach“ we bring the current config into scope, and create a named library. This „Library“ function will now make use of the previously defined .Compiler variable, which we brought into scope.

Each of the created libraries can now either be put into an „Alias“, as we‘ve seen before, built directly from the command-line using the name or used as a dependency to other targets.



## More Information

- FASTBuild Website: fastbuild.org
- Minimal Example: github.com/leandros/fastbuild-example
- Fully-Featured Template: github.com/PyrekP/FastBuildTemplate
- Find me, and I'll be happy to answer all your FASTBuild questions

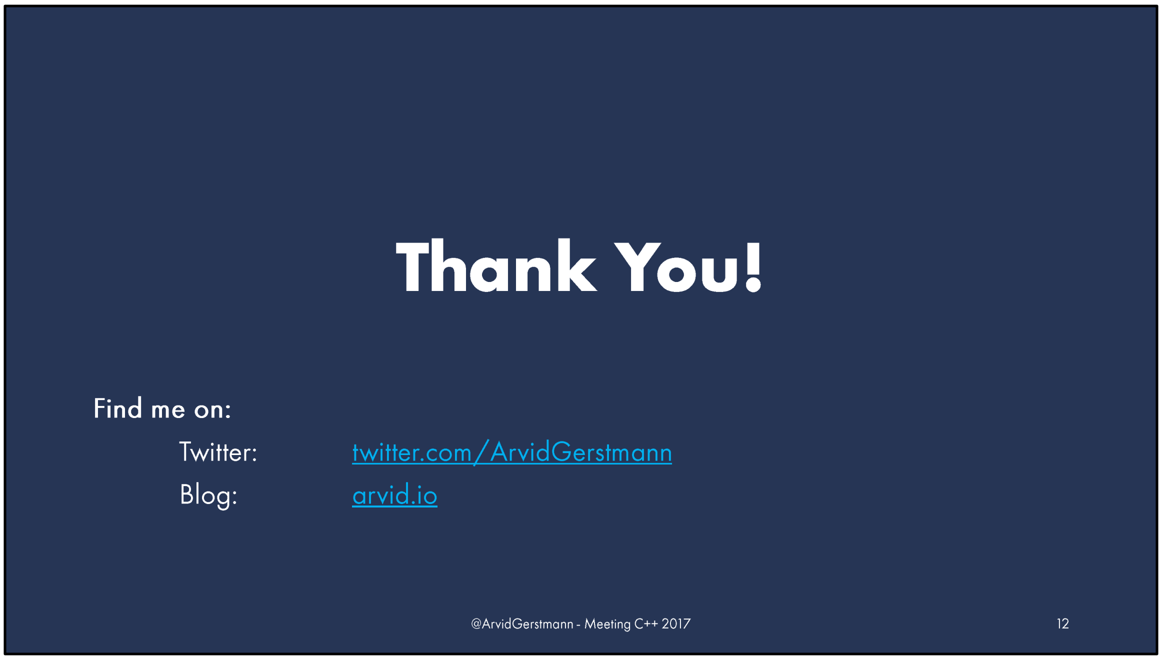
@ArvidGerstmann - Meeting C++ 2017 11

For more information, please visit the official FASTBuild website: fastbuild.org.

To get the example I've shown, visit github.com/leandros/fastbuild-example, the folder „example1/" contains the basic example I've shown. The folder „example2/" contains a more complete example, which can be used as a template to start using FASTBuild.

Another honourable mention is Pyrek Pawels „FastBuildTemplate", which can also be used to jump-start your projects using FASTBuild.

I'll be happy to answer all your FASTBuild related questions.



## Thank You!

Find me on:

Twitter: twitter.com/ArvidGerstmann

Blog: arvid.io

Thank you!