CMake

Guillaume Pagnoux

November 5, 2019

Outline

Introduction

Introducing `CMake`

Usage

Conclusion

Introduction

42sh

- You won't be able to use `make` for the project
- Instead you will use either `CMake` or `Autotools`

42sh

- You won't be able to use `make` for the project
- Instead you will use either `CMake` or `Autotools`

Why not use `make` (1/2)?

- *Real* applications are packaged and installed
- How do you package your app?
- How do you install it?
- What dependencies do you need?
- How do you build it on other OS/distributions?

Why not use `make` (1/2)?

- *Real* applications are packaged and installed
- How do you package your app?
- How do you install it?
- What dependencies do you need?
- How do you build it on other OS/distributions?

Why not use `make` (1/2)?

- *Real* applications are packaged and installed
- How do you package your app?
- How do you install it?
- What dependencies do you need?
- How do you build it on other OS/distributions?

Why not use `make` (1/2)?

- *Real* applications are packaged and installed
- How do you package your app?
- How do you install it?
- What dependencies do you need?
- How do you build it on other OS/distributions?

Why not use `make` (1/2)?

- *Real* applications are packaged and installed
- How do you package your app?
- How do you install it?
- What dependencies do you need?
- How do you build it on other OS/distributions?

Why not use `make` (2/2)?

This is not a trivial problem.

Introducing `CMake`

CMake (1/2)

- Developed by KitWare.
- Mostly used to build C++, but not only.
- A *slightly* better syntax than Autotools.
- Still not that awesome.
    - But is there a build system that is perfect?

CMake (1/2)

- Developed by KitWare.
- Mostly used to build C++, but not only.
- A *slightly* better syntax than Autotools.
- Still not that awesome.
    - But is there a build system that is perfect?

CMake (1/2)

- Developed by KitWare.
- Mostly used to build C++, but not only.
- A *slightly* better syntax than Autotools.
- Still not that awesome.
    - But is there a build system that is perfect?

CMake (1/2)

- Developed by KitWare.
- Mostly used to build C++, but not only.
- A *slightly* better syntax than Autotools.
- Still not that awesome.
    - But is there a build system that is perfect?

CMake (1/2)

- Developed by KitWare.
- Mostly used to build C++, but not only.
- A *slightly* better syntax than Autotools.
- Still not that awesome.
    - But is there a build system that is perfect?

CMake (2/2)

- New language standards support is often late.
  - Because CMake tries to be smart about languages features.
- Lacks proper coding guidelines
  - Yes, you should treat your build system as code
    - It *should* be clean!

CMake (2/2)

- New language standards support is often late.
  - Because CMake tries to be smart about languages features.
- Lacks proper coding guidelines
  - Yes, you should treat your build system as code
  - It *should* be clean!

CMake (2/2)

- New language standards support is often late.
  - Because CMake tries to be smart about languages features.
- Lacks proper coding guidelines
  - Yes, you should treat your build system as code
    - It *should* be clean!

CMake (2/2)

- New language standards support is often late.
  - Because CMake tries to be smart about languages features.
- Lacks proper coding guidelines
  - Yes, you should treat your build system as code
    - It *should* be clean!

`CMake` (2/2)

- New language standards support is often late.
  - Because `CMake` tries to be smart about languages features.
- Lacks proper coding guidelines
  - Yes, you should treat your build system as code
    - It *should* be clean!

Different `CMake` versions

- The state of `CMake` code is even worst than you think.
- The *paradigm* used in `CMake` code changed between `CMake 2` and `CMake 3`.
  - What we call *modern `CMake`* is every version `>=3.X`

Different `CMake` versions

- The state of `CMake` code is even worst than you think.
- The *paradigm* used in `CMake` code changed between `CMake` 2 and `CMake` 3.
    - What we call *modern `CMake`* is every version `>=3.X`

Different `CMake` versions

- The state of `CMake` code is even worst than you think.
- The *paradigm* used in `CMake` code changed between `CMake` 2 and `CMake` 3.
  - What we call *modern `CMake`* is every version `>=3.X`

Why does it matter?

This is a big problem:

- A solution found on the Internet might not be a good one
  - It's usually not a good one!
- You have to rely heavily on `CMake` man pages and be extremely critic about what you find online.

Why does it matter?

This is a big problem:

- A solution found on the Internet might not be a good one
  - It's usually not a good one!
- You have to rely heavily on `CMake` man pages and be extremely critic about what you find online.

Why does it matter?

This is a big problem:

- A solution found on the Internet might not be a good one
  - It's usually not a good one!
- You have to rely heavily on `CMake` man pages and be extremely critic about what you find online.

What you must keep in mind

Modern `CMake` is *target-based*.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Some explanation

- `CMake` 2 relied on variables to configure the build.
- This is a big issue:
  - You loose track of what options you defined.
  - You can end up with unwanted dependencies on your targets
  - `CMake` scoping is global. No local variables.
    - So we are in hell.
- Almost all the code you will find outside the `CMake` documentation is probably using variables heavily.

Please!

Again, *I beg you*, be critic about the code online!

Modern `CMake` master race

- *Almost* everything is now associated to a target in modern `CMake`
- Only some global settings must still be set using variables
  - And personally, I prefer to avoid those and be more verbose when writing targets

Modern `CMake` master race

- *Almost* everything is now associated to a target in modern `CMake`
- Only some global settings must still be set using variables
  - And personally, I prefer to avoid those and be more verbose when writing targets

Modern `CMake` master race

- *Almost* everything is now associated to a target in modern `CMake`
- Only some global settings must still be set using variables
  - And personally, I prefer to avoid those and be more verbose when writing targets

Usage

The necessary ~~evil~~ boilerplate

CMake builds are described in file named `CMakeLists.txt`.

```
1 cmake_minimum_required(VERSION 3.0)
2 project(42sh VERSION 1.0.0 LANGUAGES C)
```

Variables (1/2)

- We need to talk about them
- They are used in most CMake projects
- Here is how you set one:

```
1 set(VARIABLE_NAME value)
```

- You can also define lists the same way

```
1 set(LIST_NAME elt1 elt2 [elt3 ...])
```

Variables (1/2)

- We need to talk about them
- They are used in most `CMake` projects
- Here is how you set one:

```
1 set(VARIABLE_NAME value)
```

- You can also define lists the same way

```
1 set(LIST_NAME elt1 elt2 [elt3 ...])
```

Variables (1/2)

- We need to talk about them
- They are used in most `CMake` projects
- Here is how you set one:

```
1  set(VARIABLE_NAME value)
```

- You can also define lists the same way

```
1  set(LIST_NAME elt1 elt2 [elt3 ...])
```

Variables (1/2)

- We need to talk about them
- They are used in most `CMake` projects
- Here is how you set one:

```
1  set(VARIABLE_NAME value)
```

- You can also define lists the same way

```
1  set(LIST_NAME elt1 elt2 [elt3 ...])
```

Variables (2/2)

- You can access a variable value using the following syntax:

```
1 ${VARIABLE_NAME}
```

- I almost never use them.

Variables (2/2)

- You can access a variable value using the following syntax:

```
1 ${VARIABLE_NAME}
```

- I almost never use them.

Building an executable

```
1  add_executable(<executable_name> source1 [source2 ...])
```

And that's it !

Not so fast!

Like with *Autotools*, you must first configure your build.

An alternative for two possibilities

```
chesh$ mkdir build
chesh$ cd build
chesh$ cmake ..
```

Or

```
chesh$ cmake . -Bbuild
```

Now I can do a demo.

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Pimp my build

You can add a few useful flags to your build:

- `-DCMAKE_BUILD_TYPE=<Type>`
  - `Debug`: No optimizations, debug symbols
  - `Release`: Optimizations, no debug symbols
  - `RelWithDebInfo`: Some optimizations (`-Og`), debug symbols
- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
  - Creates a `compile_commands.json`. Useful for some editor features.
- `-DCMAKE_VERBOSE_MAKEFILE=ON`
  - Make the generated `Makefile` prints every executed commands (not really needed and can be overridden at build time)

Checking symbols and includes (1/2)

You can ask `CMake` to check if a header or a symbol exists. First include the correct modules:

```
include(CheckIncludeFile)
include(CheckSymbolExists)
```

Checking symbols and includes (1/2)

```
1  check_include_file("stdlib.h" HAVE_STDLIB)
2  if(NOT HAVE_STDLIB)
3    message(FATAL_ERROR "Could not find stdlib.")
4  endif()
5
6  check_symbol_exists(malloc "stdlib.h" HAVE_MALLOC)
7  if(NOT HAVE_MALLOC)
8    message(FATAL_ERROR "failed to find malloc")
9  endif()
```

Subdirectories (1/3)

- You don't want to write all your build configuration in one file when the project gets bigger.
- So you can ask `CMake` to recursively go into a subdirectory.

Subdirectories (1/3)

- You don't want to write all your build configuration in one file when the project gets bigger.
- So you can ask `CMake` to recursively go into a subdirectory.

Subdirectories (2/3)

```
1  add_subdirectory(<dir_name>)
```

Subdirectories (3/3)

Beware of the global scope!

Where is my binary?

- `CMake` do not generate all targets in the build folder itself
- A target is placed in a folder that has the same relative path in the source folder than in the build folder.
  - Meaning: `42sh` in `<root>/src` will end up being built in `<build>/src`
- This is not a problem in most traditional projects
- But it is with `42sh` (because the binary is expected to be in the build folder)

Where is my binary?

- `CMake` do not generate all targets in the build folder itself
- A target is placed in a folder that has the same relative path in the source folder than in the build folder.
  - Meaning: `42sh` in `<root>/src` will end up being built in `<build>/src`
- This is not a problem in most traditional projects
- But it is with `42sh` (because the binary is expected to be in the build folder)

Where is my binary?

- `CMake` do not generate all targets in the build folder itself
- A target is placed in a folder that has the same relative path in the source folder than in the build folder.
  - Meaning: `42sh` in `<root>/src` will end up being built in `<build>/src`
- This is not a problem in most traditional projects
- But it is with `42sh` (because the binary is expected to be in the build folder)

Where is my binary?

- `CMake` do not generate all targets in the build folder itself
- A target is placed in a folder that has the same relative path in the source folder than in the build folder.
  - Meaning: `42sh` in `<root>/src` will end up being built in `<build>/src`
- This is not a problem in most traditional projects
- But it is with `42sh` (because the binary is expected to be in the build folder)

Where is my binary?

- `CMake` do not generate all targets in the build folder itself
- A target is placed in a folder that has the same relative path in the source folder than in the build folder.
  - Meaning: `42sh` in `<root>/src` will end up being built in `<build>/src`
- This is not a problem in most traditional projects
- But it is with `42sh` (because the binary is expected to be in the build folder)

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

Introducing target properties

- Targets have properties
- These are flags, install paths, macro definitions, etc. . .
- Most of them are implicitly set by `CMake` (often by using the value of a global variable)
- And most of them are manipulated with specific functions
  - `target_include_directories`, `target_compile_definitions`, etc. . .
- Those who lacks a specific function can be set with `set_target_properties`.
  - For example, the *output directory* or the *C standard* to use.

set_target_properties

`set_target_properties(<target_name> PROPERTIES [property1 va`

And some useful properties/values:

- `C_STANDARD 99`: sets the target C standard to use
- `C_EXTENSIONS OFF`: toggle the use of *GNU extensions*
- `RUNTIME_OUTPUT_DIRECTORY`: sets where the target should end up

set_target_properties

`set_target_properties(<target_name> PROPERTIES [property1 va`

And some useful properties/values:

- `C_STANDARD 99`: sets the target C standard to use
- `C_EXTENSIONS OFF`: toggle the use of *GNU extensions*
- `RUNTIME_OUTPUT_DIRECTORY`: sets where the target should end up

set_target_properties

`set_target_properties(<target_name> PROPERTIES [property1 va`

And some useful properties/values:

- `C_STANDARD 99`: sets the target C standard to use
- `C_EXTENSIONS OFF`: toggle the use of *GNU extensions*
- `RUNTIME_OUTPUT_DIRECTORY`: sets where the target should end up

Implicit variables (1/2)

There are some variables defined implicitly by `CMake`, some on a per-file basis:

- `CMAKE_BINARY_DIR`: The build directory
- `CMAKE_CURRENT_BINARY_DIR`: The current folder in the build directory
- `CMAKE_SOURCE_DIR`: The project root (where the top-level `CMakeLists.txt` is)
- `CMAKE_CURRENT_SOURCE_DIR`: The current source directory (changes when going in a subdirectory)

Implicit variables (1/2)

There are some variables defined implicitly by `CMake`, some on a per-file basis:

- `CMAKE_BINARY_DIR`: The build directory
- `CMAKE_CURRENT_BINARY_DIR`: The current folder in the build directory
- `CMAKE_SOURCE_DIR`: The project root (where the top-level `CMakeLists.txt` is)
- `CMAKE_CURRENT_SOURCE_DIR`: The current source directory (changes when going in a subdirectory)

Implicit variables (1/2)

There are some variables defined implicitly by `CMake`, some on a per-file basis:

- `CMAKE_BINARY_DIR`: The build directory
- `CMAKE_CURRENT_BINARY_DIR`: The current folder in the build directory
- `CMAKE_SOURCE_DIR`: The project root (where the top-level `CMakeLists.txt` is)
- `CMAKE_CURRENT_SOURCE_DIR`: The current source directory (changes when going in a subdirectory)

Implicit variables (1/2)

There are some variables defined implicitly by `CMake`, some on a per-file basis:

- `CMAKE_BINARY_DIR`: The build directory
- `CMAKE_CURRENT_BINARY_DIR`: The current folder in the build directory
- `CMAKE_SOURCE_DIR`: The project root (where the top-level `CMakeLists.txt` is)
- `CMAKE_CURRENT_SOURCE_DIR`: The current source directory (changes when going in a subdirectory)

Implicit variables (2/2)

So to build `42sh` in the build directory:

```
1 set_target_properties(42sh PROPERTIES
2   RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}
3 )
```

Creating libraries - Part one (1/2)

- Building a library is, as we said, a non-trivial thing.
- Luckily for us, `CMake` already knows how to build them!
- The not that nice thing: there are three slightly different syntax for what `CMake` calls libraries. Let's see the simple one.

Creating libraries - Part one (1/2)

- Building a library is, as we said, a non-trivial thing.
- Luckily for us, `CMake` already knows how to build them!
- The not that nice thing: there are three slightly different syntax for what `CMake` calls libraries. Let's see the simple one.

Creating libraries - Part one (1/2)

- Building a library is, as we said, a non-trivial thing.
- Luckily for us, `CMake` already knows how to build them!
- The not that nice thing: there are three slightly different syntax for what `CMake` calls libraries. Let's see the simple one.

Creating libraries - Part one (2/2)

```
add_library(<library_name> [STATIC|SHARED|MODULE]
  [source1] [source2 ...]
)
```

Let's add that to our fake project!

Linking libraries - Part one (1/2)

We are now building libraries! How do we use them ?

Linking libraries - Part one (2/2)

- Using a library is more than adding a linker flag
- You may need to add macro definitions, flags, include directories, etc. . .

Linking libraries - Part one (2/2)

- Using a library is more than adding a linker flag
- You may need to add macro definitions, flags, include directories, etc. . .

Dependency types (1/3)

- Targets may have dependencies
- Dependencies have, like any other targets, properties: flags, install paths, etc. . .
- And they have a *type*
  - In other words: a scope that will define their behavior when added as a dependency
- Note: `CMake` has one way to add a dependency to a target. It is what they call *linking* the dependency to the target (even if the dependency is not an *actual* object).

Dependency types (1/3)

- Targets may have dependencies
- Dependencies have, like any other targets, properties: flags, install paths, etc...
- And they have a *type*
  - In other words: a scope that will define their behavior when added as a dependency
- Note: `CMake` has one way to add a dependency to a target. It is what they call *linking* the dependency to the target (even if the dependency is not an *actual* object).

Dependency types (1/3)

- Targets may have dependencies
- Dependencies have, like any other targets, properties: flags, install paths, etc...
- And they have a *type*
  - In other words: a scope that will define their behavior when added as a dependency
- Note: `CMake` has one way to add a dependency to a target. It is what they call *linking* the dependency to the target (even if the dependency is not an *actual* object).

Dependency types (1/3)

- Targets may have dependencies
- Dependencies have, like any other targets, properties: flags, install paths, etc. . .
- And they have a *type*
  - In other words: a scope that will define their behavior when added as a dependency
- Note: `CMake` has one way to add a dependency to a target. It is what they call *linking* the dependency to the target (even if the dependency is not an *actual* object).

Dependency types (1/3)

- Targets may have dependencies
- Dependencies have, like any other targets, properties: flags, install paths, etc. . .
- And they have a *type*
  - In other words: a scope that will define their behavior when added as a dependency
- Note: `CMake` has one way to add a dependency to a target. It is what they call *linking* the dependency to the target (even if the dependency is not an *actual* object).

Dependency types (2/3)

- `PRIVATE`: the property is used by the target only
- `INTERFACE`: the property is not used by the target, but passed to targets that depends on it.
- `PUBLIC`: `PRIVATE` + `INTERFACE`
  - In words: the property is used by the target, and targets that depends on it.

Dependency types (2/3)

- `PRIVATE`: the property is used by the target only
- `INTERFACE`: the property is not used by the target, but passed to targets that depends on it.
- `PUBLIC`: `PRIVATE` + `INTERFACE`
  - In words: the property is used by the target, and targets that depends on it.

Dependency types (2/3)

- `PRIVATE`: the property is used by the target only
- `INTERFACE`: the property is not used by the target, but passed to targets that depends on it.
- `PUBLIC`: `PRIVATE` + `INTERFACE`
    - In words: the property is used by the target, and targets that depends on it.

Dependency types (2/3)

- `PRIVATE`: the property is used by the target only
- `INTERFACE`: the property is not used by the target, but passed to targets that depends on it.
- `PUBLIC`: `PRIVATE` + `INTERFACE`
    - In words: the property is used by the target, and targets that depends on it.

Dependency types (3/3)

Why?

- Because it avoids the need to write (and thus forget) libraries, definitions, flags or include directories needed to use a specific dependency!
- Also, becomes very elite with *interfaces* (more on that later).
- That means you can have `CMake` *modules*!

Dependency types (3/3)

Why?

- Because it avoids the need to write (and thus forget) libraries, definitions, flags or include directories needed to use a specific dependency!
- Also, becomes very elite with *interfaces* (more on that later).
- That means you can have `CMake` *modules*!

Dependency types (3/3)

Why?

- Because it avoids the need to write (and thus forget) libraries, definitions, flags or include directories needed to use a specific dependency!
- Also, becomes very elite with *interfaces* (more on that later).
- That means you can have `CMake` *modules*!

Linking libraries - Part deux

```
target_link_libraries(<target_name> [PUBLIC|PRIVATE|INTERFACE]
  lib1
  [lib2 ...]
)
```

A word on language standards (1/5)

- I lied to you.
- In theory, you should not specify the language standard to use, and especially not by setting the target property manually.
  - It is still better than setting the global variable for the default C standard.
- Since `CMake` tries to be compiler generic, it is expected that you specify the compiler features you need and let `CMake` add the correct flags for your compiler.
- But `CMake` is made to build `C++` in the first place...

A word on language standards (1/5)

- I lied to you.
- In theory, you should not specify the language standard to use, and especially not by setting the target property manually.
  - It is still better than setting the global variable for the default C standard.
- Since `CMake` tries to be compiler generic, it is expected that you specify the compiler features you need and let `CMake` add the correct flags for your compiler.
- But `CMake` is made to build `C++` in the first place...

A word on language standards (1/5)

- I lied to you.
- In theory, you should not specify the language standard to use, and especially not by setting the target property manually.
  - It is still better than setting the global variable for the default C standard.
- Since `CMake` tries to be compiler generic, it is expected that you specify the compiler features you need and let `CMake` add the correct flags for your compiler.
- But `CMake` is made to build `C++` in the first place...

A word on language standards (1/5)

- I lied to you.
- In theory, you should not specify the language standard to use, and especially not by setting the target property manually.
  - It is still better than setting the global variable for the default C standard.
- Since `CMake` tries to be compiler generic, it is expected that you specify the compiler features you need and let `CMake` add the correct flags for your compiler.
- But `CMake` is made to build `C++` in the first place...

A word on language standards (1/5)

- I lied to you.
- In theory, you should not specify the language standard to use, and especially not by setting the target property manually.
  - It is still better than setting the global variable for the default C standard.
- Since `CMake` tries to be compiler generic, it is expected that you specify the compiler features you need and let `CMake` add the correct flags for your compiler.
- But `CMake` is made to build `C++` in the first place...

A word on language standards (2/5)

In C++, here is how you should do it:

```
target_compile_features(<target_name>
  [PUBLIC|PRIVATE|INTERFACE]
  cxx_constexpr
  cxx_variadic_templates
  ...
)
```

A word on language standards (3/5)

Or, if you use a lot of features from a specific standard (here C++ 17):

```
target_compile_features(<target_name
  [PUBLIC|PRIVATE|INTERFACE]
  cxx_std_17
)
```

A word on language standards (4/5)

The same thing exists in C:

```
1 target_compile_features(<target_name
2   [PUBLIC|PRIVATE|INTERFACE]
3   c_std_99
4 )
```

This is only valid for CMake versions >=3.8.

A word on language standards (5/5)

- This works, but `CMake` may not add the `-std=c99` flag if it is not necessary.
- And, for `ACU` projects, you are required to have this flag.
- Hence the syntax I showed you earlier.
- But keep in mind, for real projects, you should use *compile features*.

A word on language standards (5/5)

- This works, but `CMake` may not add the `-std=c99` flag if it is not necessary.
- And, for `ACU` projects, you are required to have this flag.
- Hence the syntax I showed you earlier.
- But keep in mind, for real projects, you should use *compile features*.

A word on language standards (5/5)

- This works, but `CMake` may not add the `-std=c99` flag if it is not necessary.
- And, for `ACU` projects, you are required to have this flag.
- Hence the syntax I showed you earlier.
- But keep in mind, for real projects, you should use *compile features*.

A word on language standards (5/5)

- This works, but `CMake` may not add the `-std=c99` flag if it is not necessary.
- And, for `ACU` projects, you are required to have this flag.
- Hence the syntax I showed you earlier.
- But keep in mind, for real projects, you should use *compile features*.

Compile flags

Here is how to add compile flags to a target:

```
target_compile_options(<target_name>
  [PUBLIC|PRIVATE|INTERFACE]
  "-Wall"
  "-Wextra"
  ...
)
```

Macro definitions

Here is how to add compile flags to a target:

```
1  target_compile_definitions(<target_name>
2    [PUBLIC|PRIVATE|INTERFACE]
3    "_GNU_SOURCE"
4    "_SOURCE"
5    "GP=13"
6    ...
7  )
```

Include directories

Here you can add include directories to a target:

```
1 target_include_directories(<target_name>
2   [PUBLIC|PRIVATE|INTERFACE]
3   directory1
4   [directory2 ...]
5   )
```

Interfaces (1/3)

- Sometimes, you might want to define a set of options, features, include directories, etc. . .
- You could set them globally with a variable, like with `CMake 2`, but again it will backfire at you *someday*.
- Other times, you may want to use a library that is header-only. . .
- . . . and still *link* it like any other libraries
- There is a tool in modern `CMake` we can use for that: *interfaces*.

Interfaces (1/3)

- Sometimes, you might want to define a set of options, features, include directories, etc. . .
- You could set them globally with a variable, like with `CMake 2`, but again it will backfire at you *someday*.
- Other times, you may want to use a library that is header-only. . .
- . . . and still *link* it like any other libraries
- There is a tool in modern `CMake` we can use for that: *interfaces*.

Interfaces (1/3)

- Sometimes, you might want to define a set of options, features, include directories, etc...
- You could set them globally with a variable, like with `CMake 2`, but again it will backfire at you *someday*.
- Other times, you may want to use a library that is header-only...
- ... and still *link* it like any other libraries
- There is a tool in modern `CMake` we can use for that: *interfaces*.

Interfaces (1/3)

- Sometimes, you might want to define a set of options, features, include directories, etc...
- You could set them globally with a variable, like with `CMake 2`, but again it will backfire at you *someday*.
- Other times, you may want to use a library that is header-only...
- ... and still *link* it like any other libraries
- There is a tool in modern `CMake` we can use for that: *interfaces*.

Interfaces (1/3)

- Sometimes, you might want to define a set of options, features, include directories, etc...
- You could set them globally with a variable, like with `CMake 2`, but again it will backfire at you *someday*.
- Other times, you may want to use a library that is header-only...
- ... and still *link* it like any other libraries
- There is a tool in modern `CMake` we can use for that: *interfaces*.

Interfaces (2/3)

```
1 add_library(<interface_name> INTERFACE)
```

Yep, it is a special kind of library! From there all you have to do is add whatever options with the `INTERFACE` type (that's important!).

Interfaces (3/3)

You can now link it with your target like any other libraries:

```
1 target_link_libraries(<target_name>
2   [PUBLIC|PRIVATE|INTERFACE]
3   [other_libraries]
4   interface_name
5   )
```

Generator expressions (1/3)

- Sometimes you may want to add some options or define a variable value conditionally.
- For example, you may want to have *address sanitizer* added automatically for *Debug* builds.
- Or, `-Werror` only on *Release* builds
- We could do that with a variable (old-school style), but we can use an other powerful `CMake`: *generator expressions*

Generator expressions (1/3)

- Sometimes you may want to add some options or define a variable value conditionally.
- For example, you may want to have *address sanitizer* added automatically for *Debug* builds.
- Or, `-Werror` only on *Release* builds
- We could do that with a variable (old-school style), but we can use an other powerful `CMake`: *generator expressions*

Generator expressions (1/3)

- Sometimes you may want to add some options or define a variable value conditionally.
- For example, you may want to have *address sanitizer* added automatically for *Debug* builds.
- Or, `-Werror` only on *Release* builds
- We could do that with a variable (old-school style), but we can use an other powerful `CMake`: *generator expressions*

Generator expressions (1/3)

- Sometimes you may want to add some options or define a variable value conditionally.
- For example, you may want to have *address sanitizer* added automatically for *Debug* builds.
- Or, `-Werror` only on *Release* builds
- We could do that with a variable (old-school style), but we can use an other powerful `CMake`: *generator expressions*

Generator expressions (2/3)

Generator expressions look like this:

- `$<KEYWORD:Value>`
- `KEYWORD` can be a generator expression, a variable, or a *keyword* that defines the behavior of the generator expression.

Generator expressions (2/3)

Generator expressions look like this:

- `$<KEYWORD:Value>`
- `KEYWORD` can be a generator expression, a variable, or a *keyword* that defines the behavior of the generator expression.

Generator expressions (3/3)

Here are some examples:

```
1  $<COMMA> # It is a comma; by the way, this is a comment
2  $<CONFIG:Debug> # Evaluates to 1 (true) if debug build
3  $<$<CONFIG:Release>:-O3> # -O3 if release build
4  # NDEBUG if release build, else NDEBUG
5  $<IF:$<CONFIG:Release>,NDEBUG,DEBUG>
6  # -fsanitize=address if not release build
7  $<IF:$<NOT:$<CONFIG:Release>>,-fsanitize=address>
```

Finding libraries (1/3)

In some projects (like `42sh`), you will want to use external libraries.

- If you, or someone else (for example the library maintainer), has written a `Find*` module, you can use the `find_package` function.
- Else, you can use the `find_library` function.

Finding libraries (1/3)

In some projects (like `42sh`), you will want to use external libraries.

- If you, or someone else (for example the library maintainer), has written a `Find*` module, you can use the `find_package` function.
- Else, you can use the `find_library` function.

Finding libraries (2/3)

Let's say you want readline:

```
1 find_library(READLINE_LIB readline)
```

This will set the variable `READLINE_LIB` to store the result of the function. It will be `NOTFOUND` if the library could not be found.

Finding libraries (3/3)

You can then link it like this:

```
target_link_libraries(<target_name>
  [PUBLIC|PRIVATE|INTERFACE]
  ${READLINE_LIB}
)
```

Testing your project (1/3)

- `CMake` has a builtin way to launch tests: `ctest`
- You can enable it with `enable_testing()`
- And add tests with:

```
1   add_test(NAME test_name COMMAND <test_command>)
```

Testing your project (1/3)

- `CMake` has a builtin way to launch tests: `ctest`
- You can enable it with `enable_testing()`
- And add tests with:

```
1   add_test(NAME test_name COMMAND <test_command>)
```

Testing your project (1/3)

- `CMake` has a builtin way to launch tests: `ctest`
- You can enable it with `enable_testing()`
- And add tests with:

```
1   add_test(NAME test_name COMMAND <test_command>)
```

Testing your project (2/3)

- The test command can be an other target
- So all you have to do is create targets for unit testing that links with parts of your project (use libraries!) and a unit testing framework...
- ... and a python test suite that you also run with `ctest`.

Testing your project (2/3)

- The test command can be an other target
- So all you have to do is create targets for unit testing that links with parts of your project (use libraries!) and a unit testing framework...
- ... and a python test suite that you also run with `ctest`.

Testing your project (2/3)

- The test command can be an other target
- So all you have to do is create targets for unit testing that links with parts of your project (use libraries!) and a unit testing framework...
- ... and a python test suite that you also run with `ctest`.

Testing your project (3/3)

- To run your test, use the `ctest` command in your build directory
- `CMake` also generates a `test` rule in the `Makefile`
- Beware, for `42sh` you need a `check` rule in the generated `Makefile`
  - This can be solved easily like this:

```
1 add_custom_target(check COMMAND ${CMAKE_CTEST_COMMAND})
```

Testing your project (3/3)

- To run your test, use the `ctest` command in your build directory
- `CMake` also generates a `test` rule in the `Makefile`
- Beware, for `42sh` you need a `check` rule in the generated `Makefile`
  - This can be solved easily like this:

```
1 add_custom_target(check COMMAND ${CMAKE_CTEST_COMMAND})
```

Testing your project (3/3)

- To run your test, use the `ctest` command in your build directory
- `CMake` also generates a `test` rule in the `Makefile`
- Beware, for `42sh` you need a `check` rule in the generated `Makefile`
    - This can be solved easily like this:

```
1  add_custom_target(check COMMAND ${CMAKE_CTEST_COMMAND})
```

Testing your project (3/3)

- To run your test, use the `ctest` command in your build directory
- `CMake` also generates a `test` rule in the `Makefile`
- Beware, for `42sh` you need a `check` rule in the generated `Makefile`
    - This can be solved easily like this:

```
1  add_custom_target(check COMMAND ${CMAKE_CTEST_COMMAND})
```

Conclusion

Conclusion

- You should now know how to do the most common things with `CMake`.
- There are many more features available that I probably don't know about.
- Again, don't forget, be critical about the code you find online.
- Avoid variables if you can.

Conclusion

- You should now know how to do the most common things with `CMake`.
- There are many more features available that I probably don't know about.
- Again, don't forget, be critical about the code you find online.
- Avoid variables if you can.

Conclusion

- You should now know how to do the most common things with `CMake`.
- There are many more features available that I probably don't know about.
- Again, don't forget, be critical about the code you find online.
- Avoid variables if you can.

Conclusion

- You should now know how to do the most common things with `CMake`.
- There are many more features available that I probably don't know about.
- Again, don't forget, be critical about the code you find online.
- Avoid variables if you can.

Documentation

- CMake Documentation
- An introduction to modern CMake

Questions

Questions ?