CMake tutorial

and its friends CPack, CTest and CDash

Eric NOULARD - eric.noulard@gmail.com

https://cmake.org

Compiled on September 21, 2017

Initially given by Eric Noulard for Toulibre on February, 8^{th} 2012.

Thanks to...

- Kitware for making a really nice set of tools and making them open-source
- the CMake mailing list for its friendliness and its more than valuable source of information
- CMake developers for their tolerance when I break the dashboard or mess-up with the Git workflow,
- CPack users for their patience when things don't work as they ~~should~~ expect
- Alan, Alex, Bill, Brad, Clint, David, Eike, Julien, Mathieu, Michael & Michael, Stephen, Domen, and many more...
- My son Louis for the nice CPack 3D logo done with Blender.
- and...Toulibre for initially hosting this presention in Toulouse, France.

Outline

And thanks to contributors as well...

History

This presentation was initially made by Eric Noulard for a Toulibre (http://www.toulibre.fr) given in Toulouse (France) on February, 8^{th} 2012. After that, the source of the presentation has been release under CC-BY-SA, http://creativecommons.org/licenses/by-sa/3.0/us/ and put on https://github.com/TheErk/CMake-tutorial then contributors stepped-in.

Many thanks to all contributors (alphabetical order):

Contributors

Sébastien Dinot, Andreas Mohr.

Outline

CMake tool sets

CMake

CMake is a cross-platform build systems generator which makes it easier to build software in a unified manner on a broad set of platforms:

 ,  ,  ,  , AIX, IRIX,  , iOS...

CMake has friends softwares that may be used on their own or together:

- CMake: build system generator
- CPack: package generator
- CTest: systematic test driver
- CDash: a dashboard collector

Outline of Part I: CMake

Build what?

Software build system

A software build system is the usage of a [set of] tool[s] for building software applications.

Why do we need that?

Build what?

Software build system

A software build system is the usage of a [set of] tool[s] for building software applications.

Why do we need that?

- because most softwares consist of several parts that need some building to put them together,

Build what?

Software build system

A software build system is the usage of a [set of] tool[s] for building software applications.

Why do we need that?

- because most softwares consist of several parts that need some building to put them together,
- because softwares are written in various languages that may share the same building process,

Build what?

Software build system

A software build system is the usage of a [set of] tool[s] for building software applications.

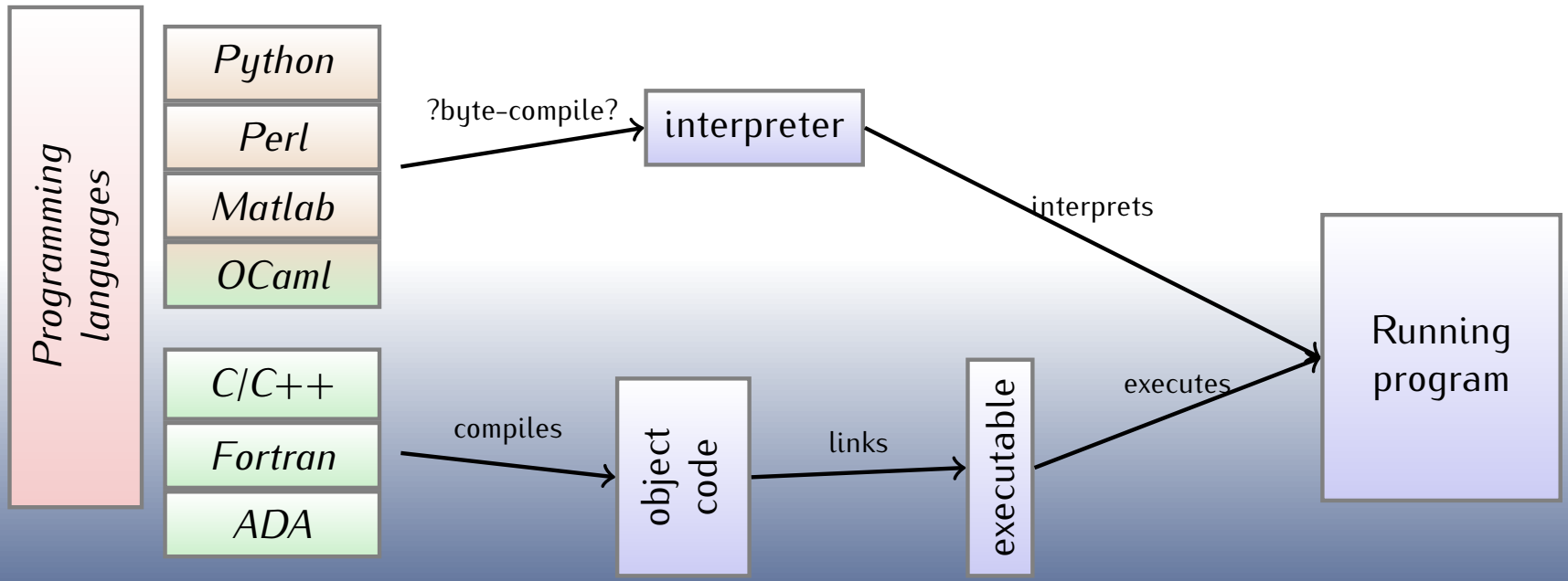
Why do we need that?

- because most softwares consist of several parts that need some building to put them together,
- because softwares are written in various languages that may share the same building process,
- because we want to build the same software for various computers (PC, Macintosh, Workstation, mobile phones and other PDA, embedded computers) and systems (Windows, Linux, *BSD, other Unices (many), Android, etc...)

Programming languages

Compiled vs interpreted or what?

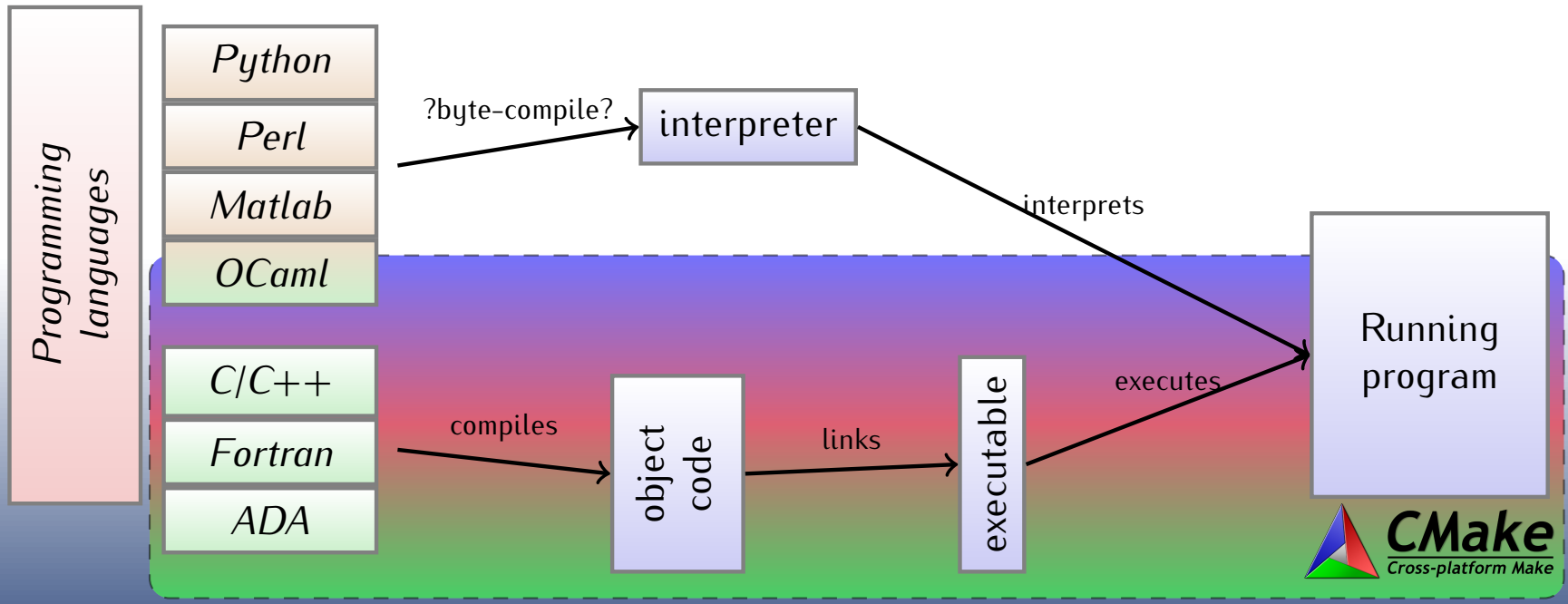
Building an application requires the use of some programming language: Python, Java, C++, Fortran, C, Go, Tcl/Tk, Ruby, Perl, OCaml,...



Programming languages

Compiled vs interpreted or what?

Building an application requires the use of some programming language: Python, Java, C++, Fortran, C, Go, Tcl/Tk, Ruby, Perl, OCaml,...



Build systems: several choices

Alternatives

CMake is not the only build system [generator]:

- (portable) hand-written Makefiles, depends on `make` tool (may be GNU Make).
- Apache ant (or Maven or Gradle), dedicated to Java (almost).
- Portable IDE: Eclipse, Code::Blocks, Geany, NetBeans, ...
- GNU Autotools: Autoconf, Automake, Libtool. Produce makefiles. Bourne shell needed (and M4 macro processor).
- http://www.scons.org only depends on Python.
- ...

Build systems or build systems generator

Build systems

A tool which builds, a.k.a. compiles, a set of source files in order to produce binary executables and libraries. Those kind of tools usually takes as input a file (e.g. a Makefile) and while reading it issues compile commands. The main goal of a build tool is to (re)build the minimal subset of files when something changes. A non exhaustive list: [GNU] make, ninja, MSBuild, SCons, ant, ...

A **Build systems generator** is a tool which generates files for a particular build system. e.g. CMake or Autotools.

What build systems do?

Targets and sources

The main feature of a build system is to offer a way to describe how a target (executable, PDF, shared library...) is built from its sources (set of object files and/or libraries, a latex or rst file, set of C/C++/Fortran files...). Basically a target **depends** on one or several sources and one can run a set of **commands** in order to built the concerned target from its sources.

The main goals/features may be summarized as:

- describe dependency graph between sources and targets
- associate one or several commands to rebuilt target from source(s)
- issue the minimal set of commands in order to rebuild a target

A sample Makefile for make

```
1  CC=gcc
2  CFLAGS=-Wall -Werror -pedantic -std=c99
3  LDFLAGS=
4  EXECUTABLES=Acrodictlibre Acrolibre
5
6  # default rule (the first one)
7  all : $(EXECUTABLES)
8  # explicit link target
9  Acrolibre: acrolibre.o
10     $(CC) $(CFLAGS) -o $@ $^
11 # explicit link and compile target
12 Acrodictlibre: acrolibre.c acrodict.o
13     $(CC) $(CFLAGS) -DUSE_ACRODICT -o $@ $^
14
15 # Implicit rule using file extension
16 # Every .o file depends on corresponding .c (and may be .h) file
17 %.o : %.c %.h
18     $(CC) $(CFLAGS) -c $<
19 %.o : %.c
20     $(CC) $(CFLAGS) -c $<
21 clean:
22     @\rm -f *.o $(EXECUTABLES)
```

Comparisons and [success] stories

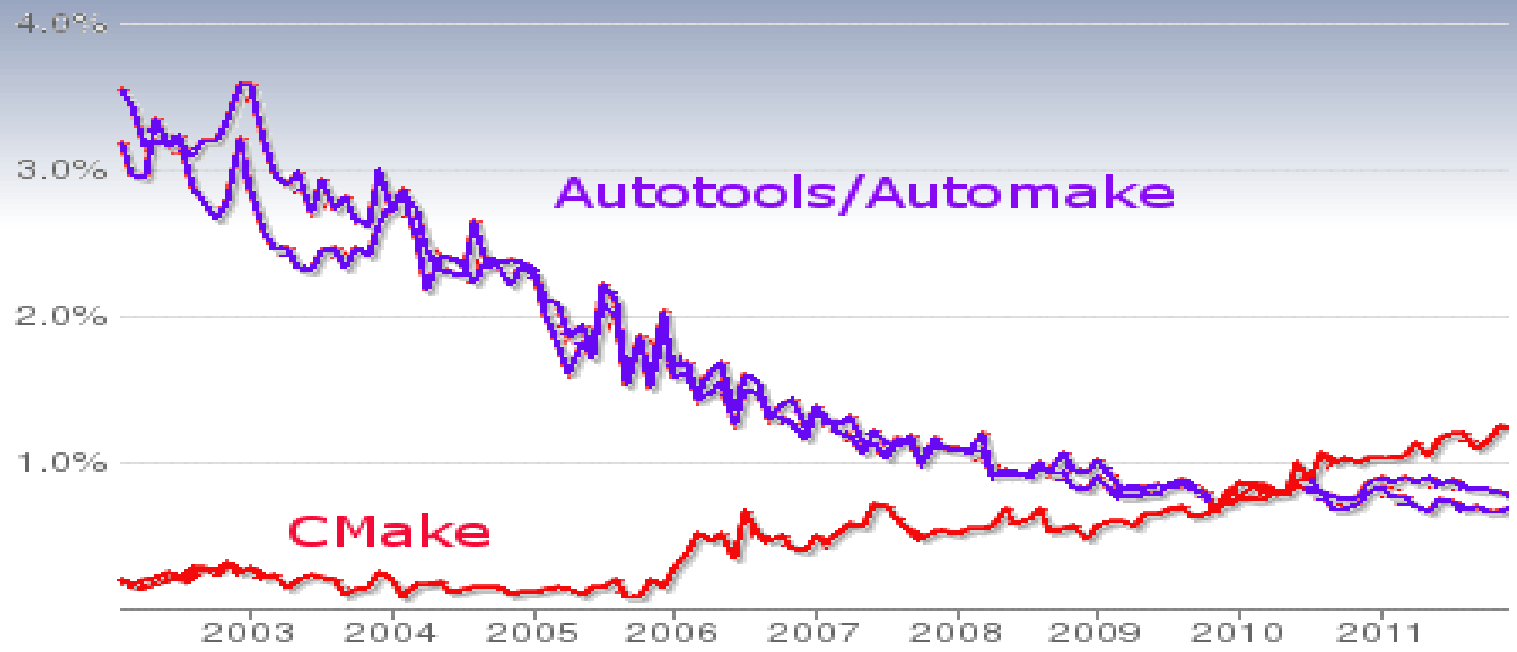
Disclaimer

This presentation is biased. I mean totally.
I am a big CMake fan, I did contribute to CMake, thus I'm not impartial at all. But I will be ready to discuss why CMake is the greatest build system out there :-)

Go and forge your own opinion:

- Bare list: http://en.wikipedia.org/wiki/List_of_build_automation_software
- A comparison: http://www.scons.org/wiki/SconsVsOtherBuildTools
- KDE success story (2006): "Why the KDE project switched to CMake – and how" http://lwn.net/Articles/188693/

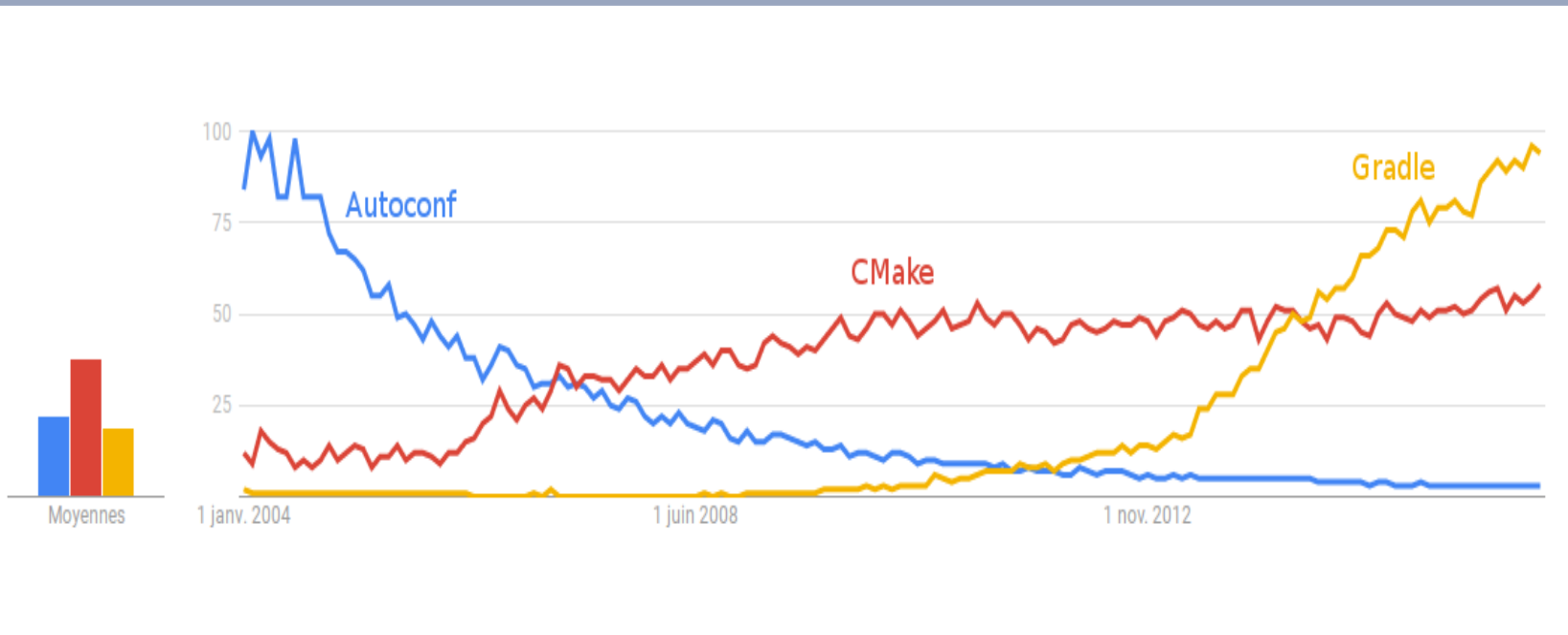
CMake/Auto[conf|make] on OpenHub



https://www.openhub.net/languages/compare

Language comparison of CMake to automake and autoconf showing the percentage of developers commits that modify a source file of the respective language (data from 2012).

CMake/Autoconf/Gradle on Google Trend



https://www.google.com/trends

Scale is based on the average worldwide request traffic searching for CMake, Autoconf and Gradle in all years (2004–now).

Outline

A build system generator

- CMake is a generator: it generates native build systems files (Makefile, Ninja, IDE project files [XCode, CodeBlocks, Eclipse CDT, Codelite, Visual Studio, Sublime Text...], ...),
- CMake scripting language (declarative) is used to describe the build,
- The developer edits `CMakeLists.txt`, invokes CMake but should never edit the generated files,
- CMake may be (automatically) re-invoked by the build system,
- CMake has friends who may be very handy (CPack, CTest, CDash)

The CMake workflow

When do things take place?

CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow

1. CMake time: CMake is running & processing `CMakeLists.txt`

When do things take place?

CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow

1. CMake time: CMake is running & processing `CMakeLists.txt`
2. Build time: the build tool runs and invokes (at least) the compiler

When do things take place?

CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow

1. CMake time: CMake is running & processing `CMakeLists.txt`
2. Build time: the build tool runs and invokes (at least) the compiler
3. Install time: the compiled binaries are installed
   i.e. from build area to an install location.

When do things take place?

CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow

1. CMake time: CMake is running & processing `CMakeLists.txt`
2. Build time: the build tool runs and invokes (at least) the compiler
3. Install time: the compiled binaries are installed
   i.e. from build area to an install location.
4. CPack time: CPack is running for building package

When do things take place?

CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow

1. CMake time: CMake is running & processing `CMakeLists.txt`
2. Build time: the build tool runs and invokes (at least) the compiler
3. Install time: the compiled binaries are installed
   i.e. from build area to an install location.
4. CPack time: CPack is running for building package
5. Package Install time: the package (from previous step) is installed

When do things take place?

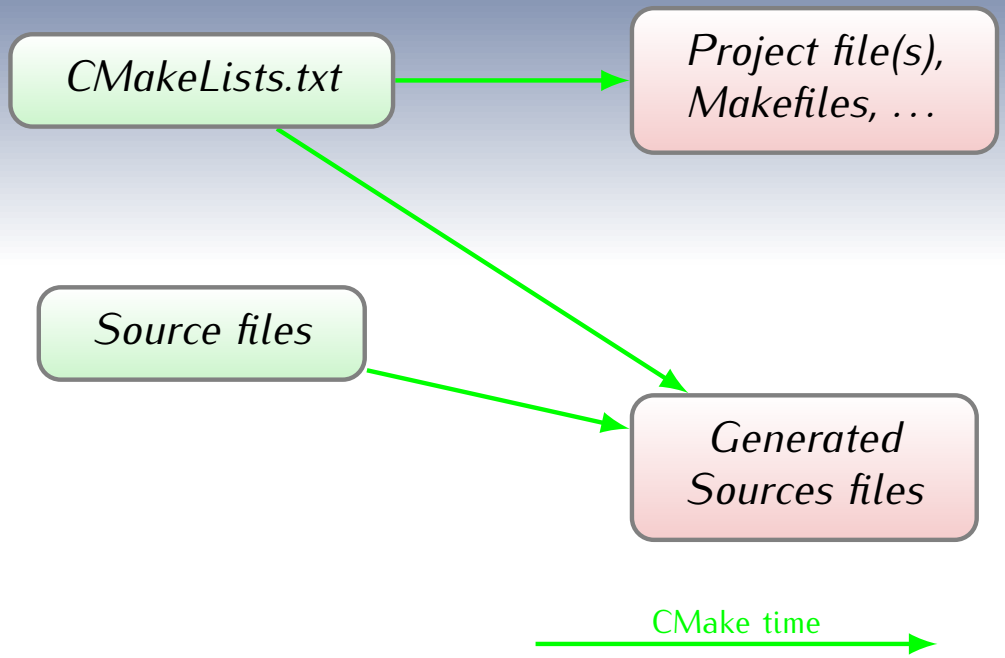
CMake is a generator which means it does not compile (i.e. build) the sources, the underlying build tool (make, Ninja, XCode, Visual Studio...) does.

The CMake workflow (pictured)

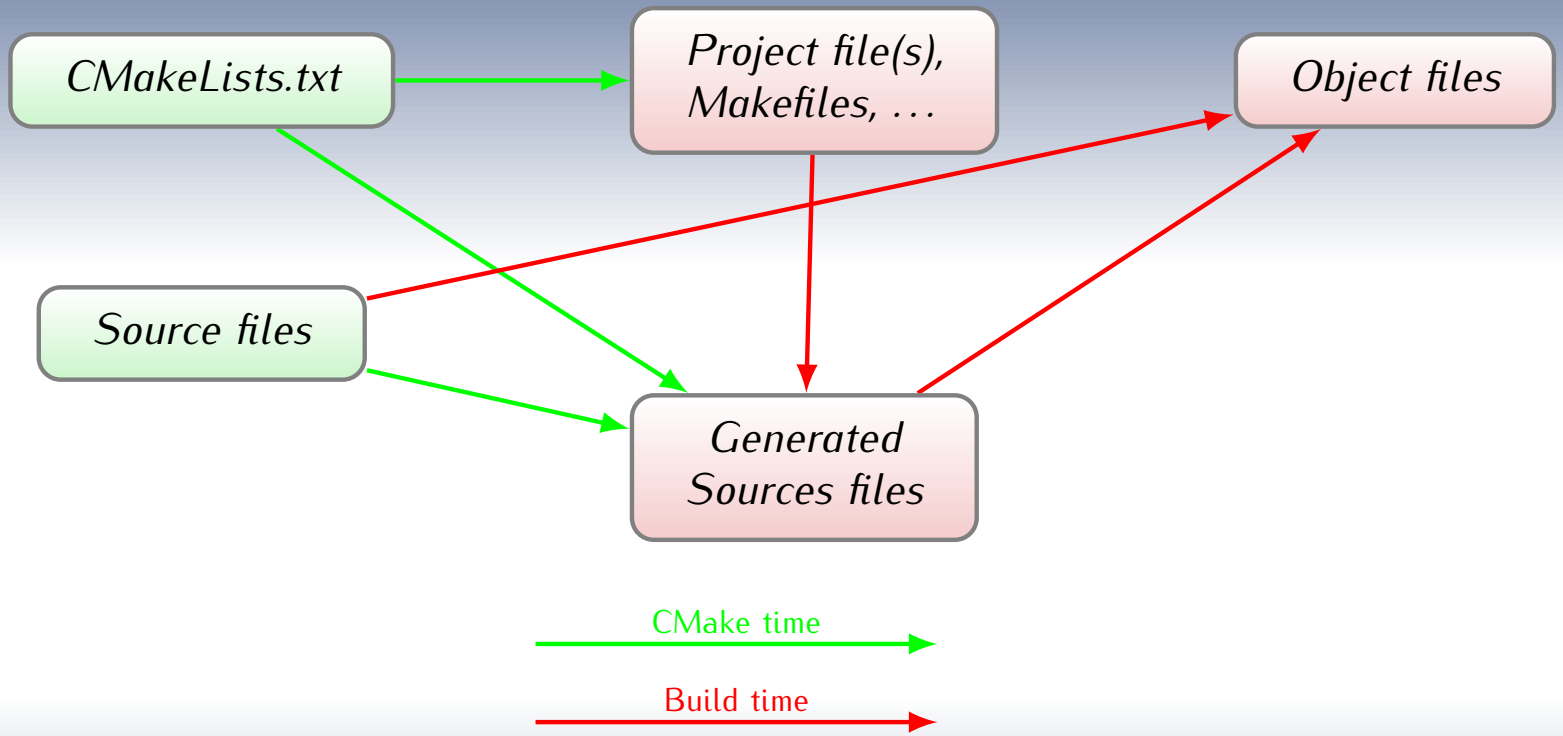
CMakeLists.txt

Source files

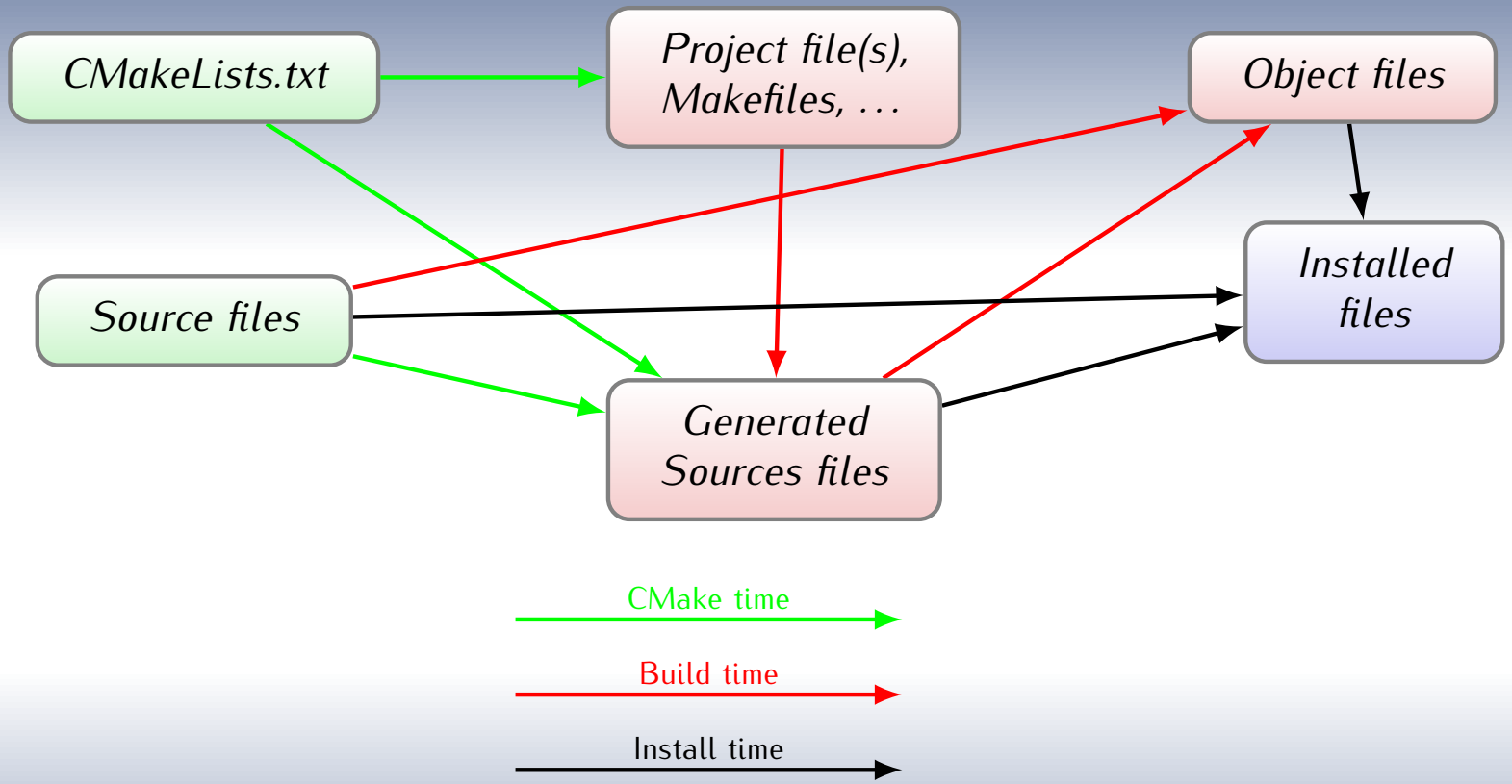
The CMake workflow (pictured)



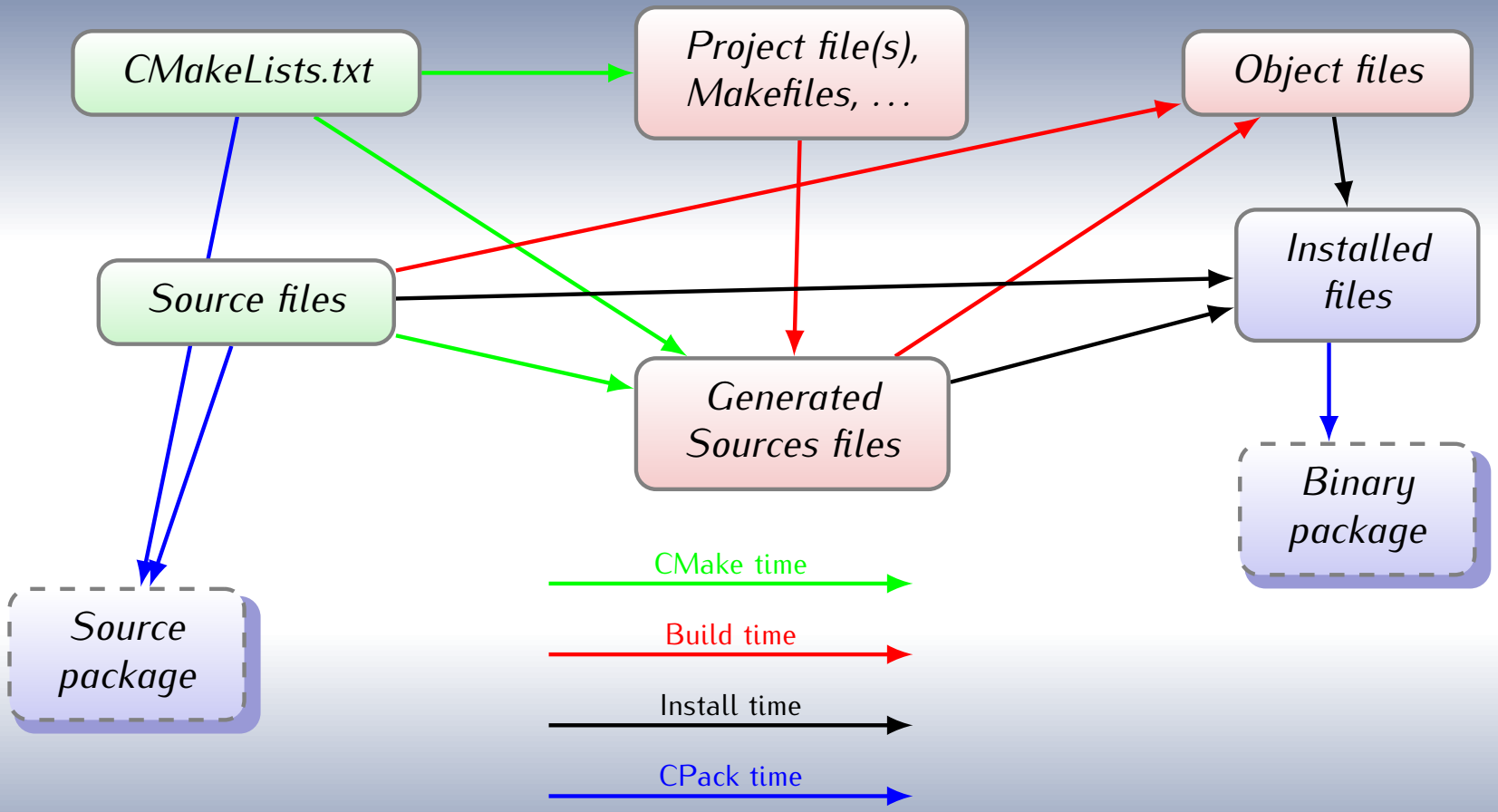
The CMake workflow (pictured)



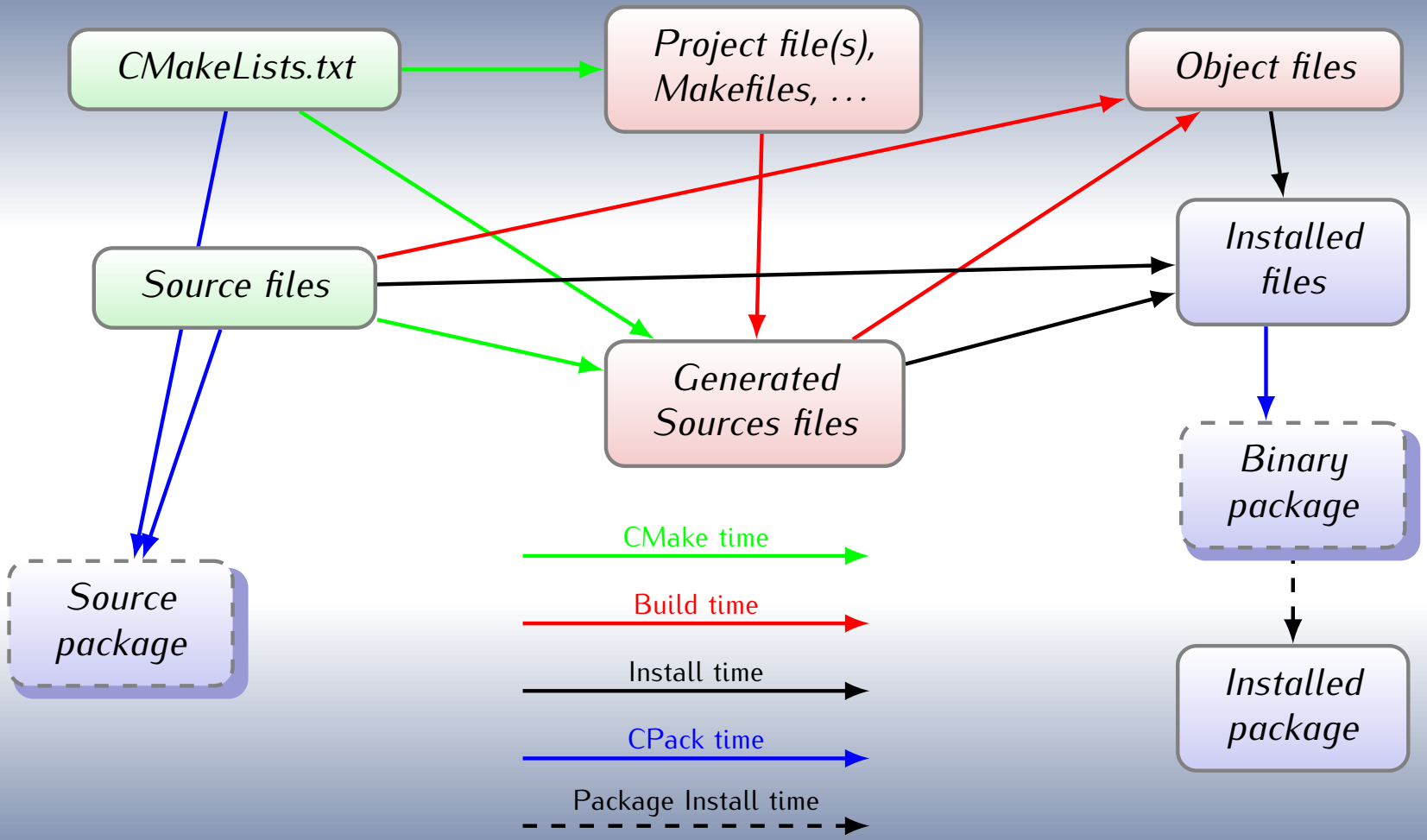
The CMake workflow (pictured)



The CMake workflow (pictured)



The CMake workflow (pictured)



Building an executable

Listing 1: Building a simple program

```
1 cmake_minimum_required (VERSION 3.0)
2 # This project use C source code
3 project (TotallyFree C)
4 set(CMAKE_C_STANDARD 99)
5 set(CMAKE_C_EXTENSIONS False)
6 # build executable using specified list of source files
7 add_executable(Acrolibre acrolibre.c)
```

CMake scripting language is [mostly] declarative. It has commands which are documented from within CMake:

```
$ cmake --help-command-list | wc -l
117
$ cmake --help-command add_executable
...
  add_executable
       Add an executable to the project using the specified source files.
```

Builtin documentation I

CMake builtin doc for 'project' command

```
  $ cmake --help-command project
project
-------

Set a name, version, and enable languages for the entire project.

 project(<PROJECT-NAME> [LANGUAGES] [<language-name>...])
 project(<PROJECT-NAME>
         [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
         [LANGUAGES <language-name>...])

Sets the name of the project and stores the name in the
``PROJECT_NAME`` variable.
[...]
       Optionally you can specify which languages your project supports.
       Example languages are CXX (i.e.  C++), C, Fortran, etc.  By default C and CXX are enabled.
       E.g.  if you do  not have a C++ compiler, you can disable the check  for it by  explicitly
       listing the languages  you want  to support, e.g.  C. By using the special language "NONE"
       all checks for any language can be disabled.
```

Builtin documentation II

Online doc : https://cmake.org/documentation/
Unix Manual: `cmake-variables(7)`, `cmake-commands(7)`, `cmake-xxx(7)`, ...
All doc generated using Sphinx, QtHelp file as well:

1. get QtHelp file from CMake:
   https://cmake.org/cmake/help/v3.6/CMake.qch
   and copy it to `CMake-tutorial/examples/`
2. use CMake.qhcp you may find in the source of this tutorial:
   `CMake-tutorial/examples/CMake.qhcp`
3. compile QtHelp collection file:
   `qcollectiongenerator CMake.qhcp -o CMake.qhc`
4. display it using Qt Assistant:
   `assistant -collectionFile CMake.qhc`

Generating & building

Building with CMake and make is easy:

Building with make

```
1  $ ls totally-free
2  acrolibre.c  CMakeLists.txt
3  $ mkdir build
4  $ cd build
5  $ cmake ../totally-free
6  -- The C compiler identification is GNU 4.6.2
7  -- Check for working C compiler: /usr/bin/gcc
8  -- Check for working C compiler: /usr/bin/gcc -- works
9  ...
10 $ make
11 ...
12  [100%] Built target Acrolibre
13 $ ./Acrolibre toulibre
```

Source tree vs Build tree

Even the most simple project should never mix-up sources with generated files. CMake supports out-of-source build.

Generating & building

Building with CMake and ninja is easy:

Building with ninja

```
1  $ ls totally-free
2  acrolibre.c  CMakeLists.txt
3  $ mkdir build-ninja
4  $ cd build-ninja
5  $ cmake -GNinja ../totally-free
6  -- The C compiler identification is GNU 4.6.2
7  -- Check for working C compiler: /usr/bin/gcc
8  -- Check for working C compiler: /usr/bin/gcc -- works
9  ...
10 $ ninja
11 ...
12 [6/6] Linking C executable Acrodictlibre
13 $ ./Acrolibre toulibre
```

Source tree vs Build tree

Even the most simple project should never mix-up sources with generated files. CMake supports out-of-source build.

Generating & building

Cross-Building with CMake and make is easy:

Building with cross-compiler

```
1  $ ls totally-free
2  acrolibre.c  CMakeLists.txt
3  $ mkdir build-win32
4  $ cd build-win32
5  $ cmake -DCMAKE_TOOLCHAIN_FILE=../totally-free/Toolchain-cross-linux.cmake ../totally-free
6  -- The C compiler identification is GNU 6.1.1
7  -- Check for working C compiler: /usr/bin/i686-w64-mingw32-gcc
8  ...
9  $ make
10 ...
11 [100%] Linking C executable Acrolibre.exe
12 [100%] Built target Acrolibre
13 $ ./Acrolibre toulibre
```

Source tree vs Build tree

Even the most simple project should never mix-up sources with generated files. CMake supports out-of-source build.

Always build out-of-source

Out-of-source is better

People are lazy (me too) and they think that because building in source is possible and authorizes less typing they can get away with it. In-source build is a BAD choice.

Out-of-source build is always better because:

Always build out-of-source

Out-of-source is better

People are lazy (me too) and they think that because building in source is possible and authorizes less typing they can get away with it. In-source build is a BAD choice.

Out-of-source build is always better because:

1. Generated files are separated from manually edited ones (thus you don't have to clutter your favorite VCS ignore files).

Always build out-of-source

Out-of-source is better

People are lazy (me too) and they think that because building in source is possible and authorizes less typing they can get away with it. In-source build is a BAD choice.

Out-of-source build is always better because:

1. Generated files are separated from manually edited ones (thus you don't have to clutter your favorite VCS ignore files).
2. You can have several build trees for the same source tree

Always build out-of-source

Out-of-source is better

People are lazy (me too) and they think that because building in source is possible and authorizes less typing they can get away with it. In-source build is a BAD choice.

Out-of-source build is always better because:

1. Generated files are separated from manually edited ones (thus you don't have to clutter your favorite VCS ignore files).
2. You can have several build trees for the same source tree
3. This way it's always safe to completely delete the build tree in order to do a clean build

Too much keyboard, time to click? I

CMake comes with severals tools

A matter of choice / taste:

- a command line: `cmake`
- a curses-based TUI: `ccmake`
- a Qt-based GUI: `cmake-gui`

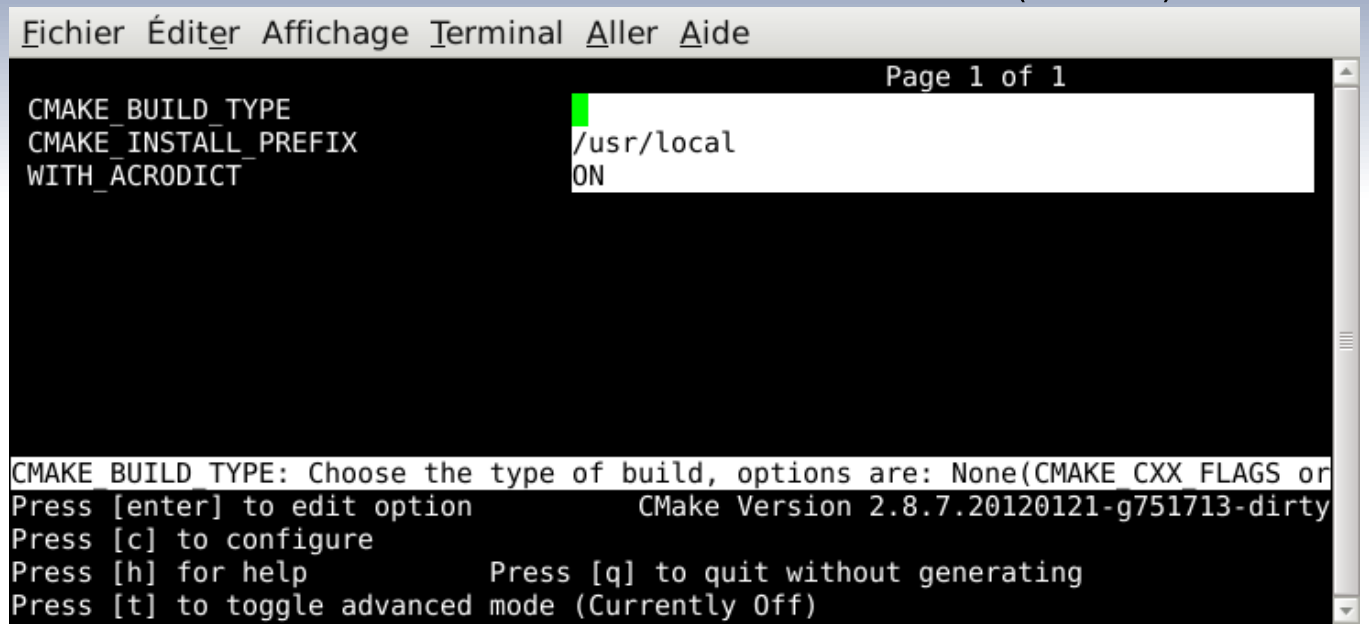
Calling convention

All tools expect to be called with a single argument which may be interpreted in 2 different ways.

- path to the source tree, e.g.: `cmake /path/to/source`
- path to an existing build tree, e.g.: `cmake-gui .`

Too much keyboard, time to click? II

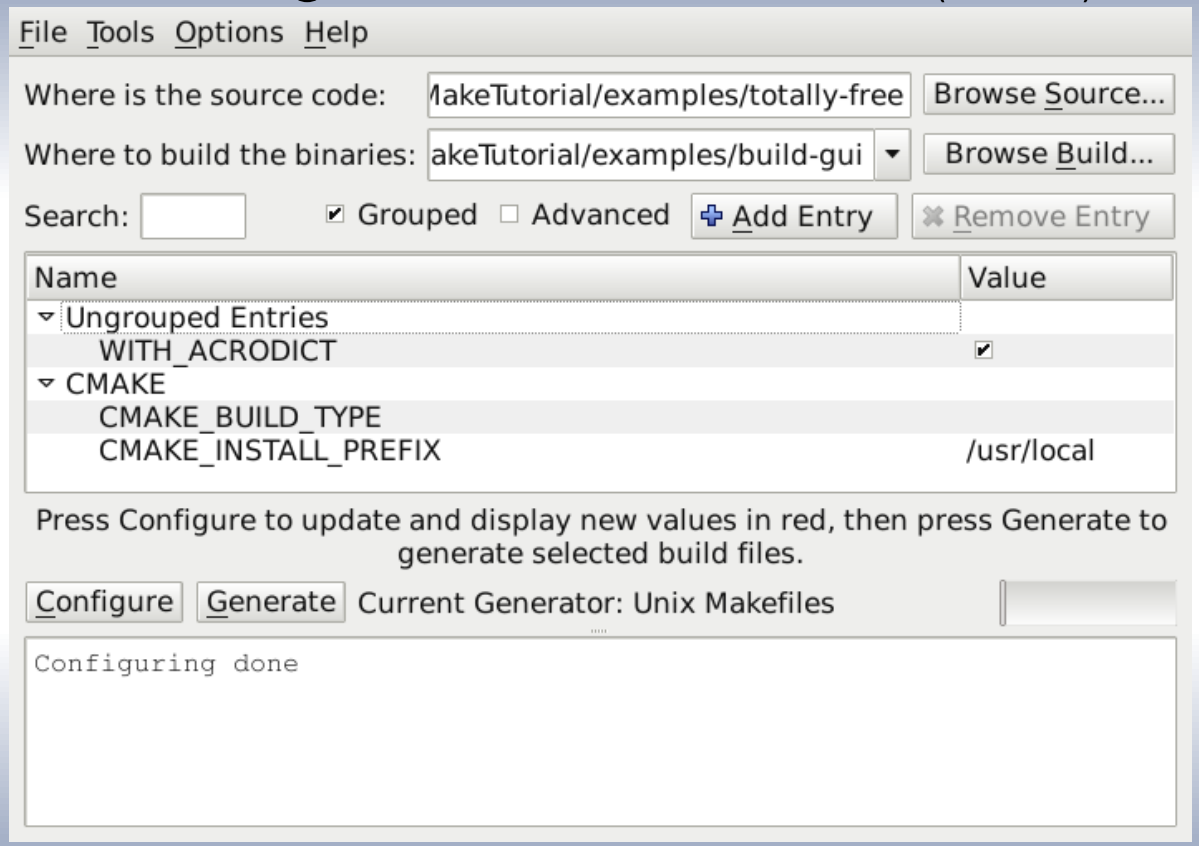
`ccmake` : the curses-based TUI (demo)



Here we can choose to toggle the WITH_ACRODICT **OPTION**.

Too much keyboard, time to click? III

`cmake-gui` : the Qt-based GUI (demo)



Again, we can choose to toggle the WITH_ACRODICT **OPTION**.

Remember CMake is a build **generator**?

The number of active generators depends on the platform we are running on Unix, Apple, Windows:

```
1  Borland Makefiles
2  MSYS Makefiles
3  MinGW Makefiles
4  NMake Makefiles
5  NMake Makefiles JOM
6  Unix Makefiles
7  Visual Studio 10
8  Visual Studio 10 IA64
9  Visual Studio 10 Win64
10 Visual Studio 11
11 Visual Studio 11 Win64
12 Visual Studio 6
13 Visual Studio 7
14 Visual Studio 7 .NET 2003
15 Visual Studio 8 2005
16 Visual Studio 8 2005 Win64
17 Visual Studio 9 2008
18 Visual Studio 9 2008 IA64
19 Visual Studio 9 2008 Win64
20 Watcom WMake
21 CodeBlocks - MinGW Makefiles
22 CodeBlocks - NMake Makefiles
23 CodeBlocks - Unix Makefiles
24 Eclipse CDT4 - MinGW Makefiles
25 Eclipse CDT4 - NMake Makefiles
26 Eclipse CDT4 - Unix Makefiles
27 KDevelop3
28 KDevelop3 - Unix Makefiles
29 XCode
30 Ninja
```

Equally simple on other platforms

It is as easy for a Windows build, however names for executables and libraries are computed in a platform specific way.

CMake + MinGW Makefile

```
1  $ ls totally-free
2  acrodict.h acrodict.c acrolibre.c  CMakeLists.txt
3  $ mkdir build-win32
4  $ cd build-win32
5  $ cmake -DCMAKE_TOOLCHAIN_FILE=../totally-free/Toolchain-cross-linux.cmake ../totally-free
6  ...
7  $ make
8  Scanning dependencies of target acrodict
9  [ 33%] Building C object CMakeFiles/acrodict.dir/acrodict.c.obj
10 Linking C shared library libacrodict.dll
11 Creating library file: libacrodict.dll.a
12 [ 33%] Built target acrodict
13 Scanning dependencies of target Acrodictlibre
14 [ 66%] Building C object CMakeFiles/Acrodictlibre.dir/acrolibre.c.obj
15 Linking C executable Acrodictlibre.exe
16 [ 66%] Built target Acrodictlibre
17 Scanning dependencies of target Acrolibre
18 [100%] Building C object CMakeFiles/Acrolibre.dir/acrolibre.c.obj
19 Linking C executable Acrolibre.exe
20 [100%] Built target Acrolibre
```

Installing things

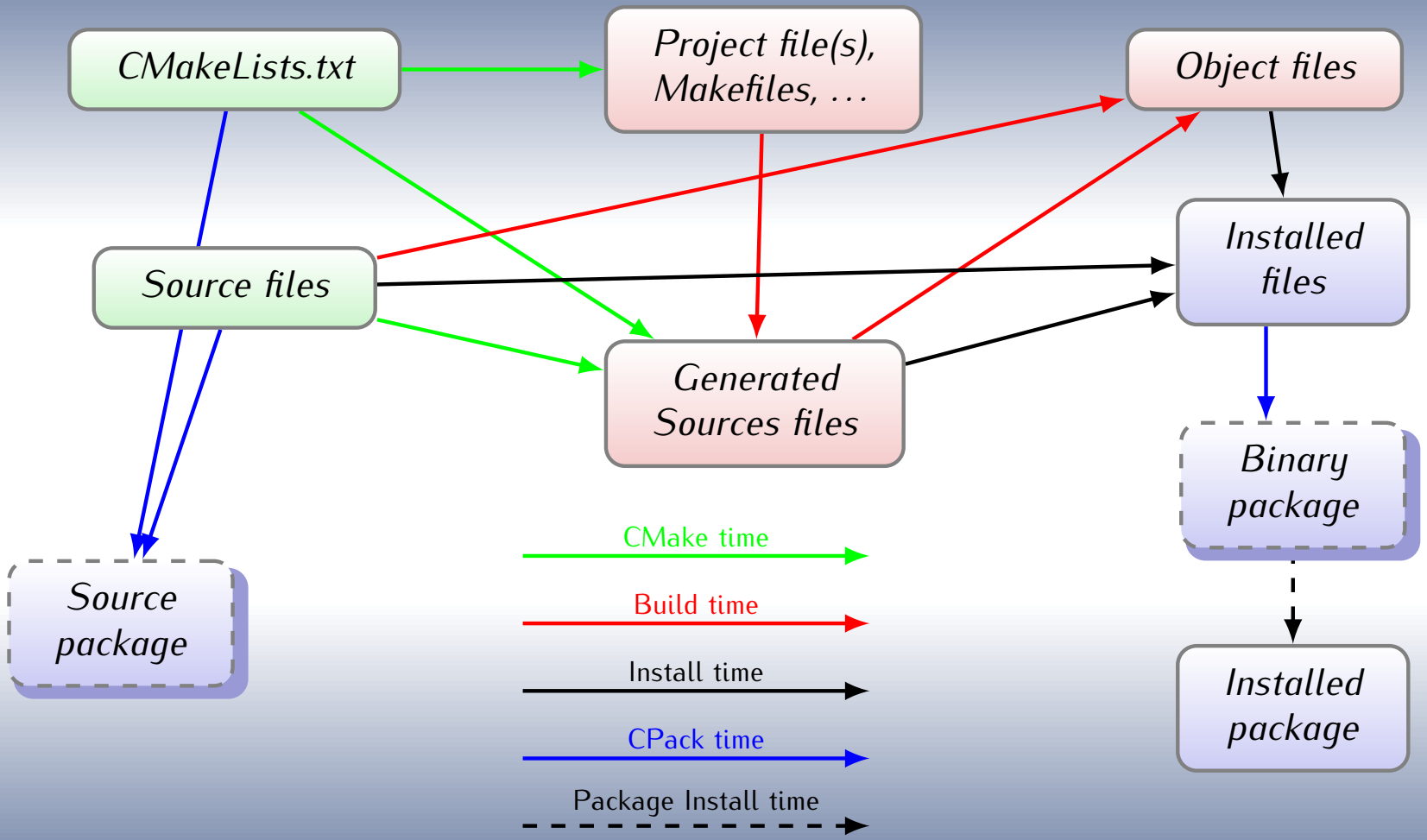
Install

Several parts or the software may need to be installed: this is controlled by the CMake **install** command.
Remember `cmake --help-command install`!!

Listing 2: install command examples

```
...
add_executable(Acrolibre acrolibre.c)
install(TARGETS Acrolibre DESTINATION bin)
if (WITH_ACRODICT)
  ...
  install(TARGETS Acrodictlibre acrodict
          RUNTIME DESTINATION bin
          LIBRARY DESTINATION lib
          ARCHIVE DESTINATION lib/static)
  install(FILES acrodict.h DESTINATION include)
endif(WITH_ACRODICT)
```

The CMake workflow (pictured)



The install target

Install target

The `install` target of the underlying build tool (in our case `make`) appears in the generated build system as soon as some **install** commands are used in the `CMakeLists.txt`.

```
1  $ make DESTDIR=/tmp/testinstall install
2  [ 33%] Built target acrodict
3  [ 66%] Built target Acrodictlibre
4  [100%] Built target Acrolibre
5  Install the project...
6  -- Install configuration: ""
7  -- Installing: /tmp/testinstall/bin/Acrolibre
8  -- Installing: /tmp/testinstall/bin/Acrodictlibre
9  -- Removed runtime path from "/tmp/testinstall/bin/Acrodictlibre"
10 -- Installing: /tmp/testinstall/lib/libacrodict.so
11 -- Installing: /tmp/testinstall/include/acrodict.h
12 $
```

Summary

CMake basics

Using CMake basics we can already do a lot of things with minimal writing.

- Write simple build specification file: `CMakeLists.txt`
- Discover compilers (C, C++, Fortran)
- Build executable and library (shared or static) in a cross-platform manner
- Package the resulting binaries with CPack
- Run systematic tests with CTest and publish them with CDash

Seeking more information or help

There are several places you can go by yourself:

1. (re-)Read the documentation: https://cmake.org/documentation
2. Read the FAQ: https://cmake.org/Wiki/CMake_FAQ
3. Read the Wiki: https://cmake.org/Wiki/CMake
4. Ask on the Mailing List: https://cmake.org/mailing-lists
5. Browse the built-in help:

```
man cmake-xxxx
cmake --help-xxxxx
assistant -collectionFile examples/CMake.qhc
```