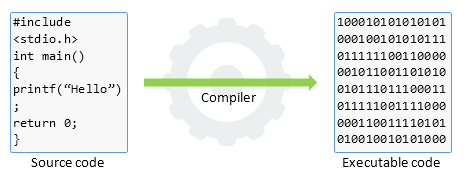
# Compilation Process

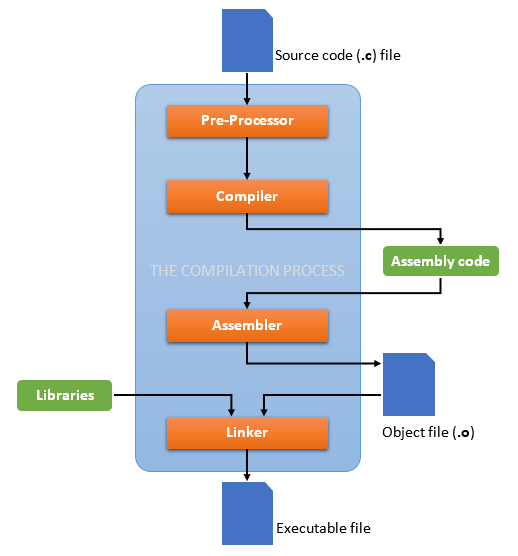
The process of translating source code written in high-level language to low-level machine code is called *compilation*. The compilation is done by a special software known as [compiler](https://codeforwin.org/2017/05/compiler-and-its-need.html). The compiler checks source code for any syntactical or structural error, and generates object code with extension .obj (in Windows) or .o (in Linux) if source code is error-free.



The entire C/C++ compilation process is broken to four stages.

* Pre-processing
* Compilation
* Assembling
* Linking

The below image describes the entire C/C++ compilation process:



To take a deep dive inside the C/C++ compilation process, let’s compile a C program. Create a text file named helloworld.c without following content:

#include <stdio.h>

int main()

{

printf("Hello, World!");

return 0;

}

To compile the above program, open command prompt and use below command:

gcc -save-temps helloworld.c -o helloworld

The -save-temps option will preserve and save all temporary files created during the C/C++ compilation. It will generate four files in the same directory namely:

// On Linux:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.o (generated by assembler)

helloworld (generated by linker)

// On Windows:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.obj (generated by assembler)

helloworld.exe (generated by linker)

Now let's look into these files and learn about different stages of compilation.

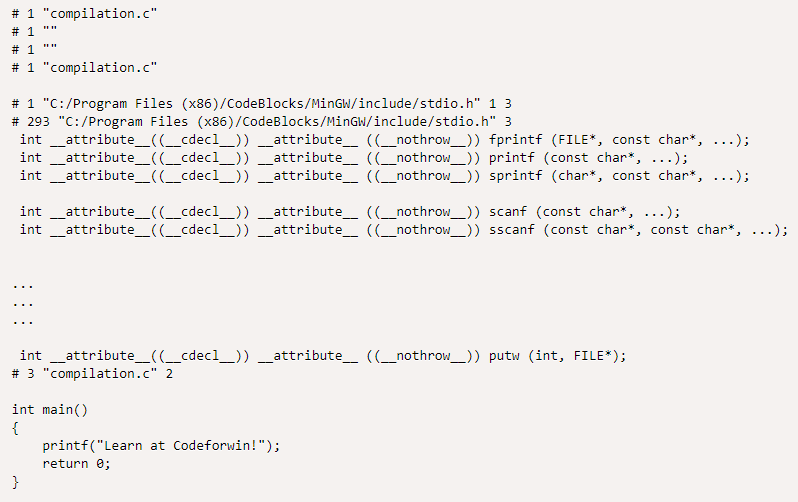
## Pre-Processing

Pre-processor is a small software that performs below tasks:

* Remove comments from the source code.
* Expansion of included header files.
* Macro expansion.

After pre-processing, a temporary with .i extension is generated. Since, it inserts contents of header files to the source code file, this generated file has a larger size than the original source code file.

Here is an extract of compilation.i file:



You can notice that the statement #include<stdio.h> is replaced by its contents. Comment before the #include line is also trimmed.

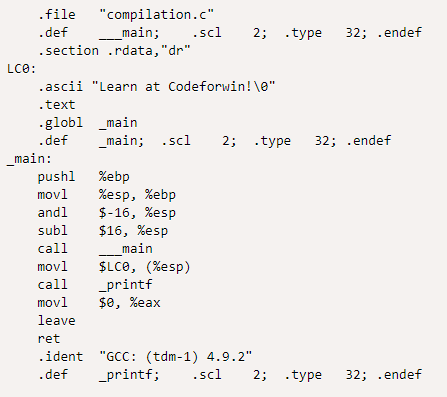
## Compilation

In next phase, the compiler performs following tasks:

* Check the program for syntax errors.
* Translate the file into assembly language (intermediate code).
* Optionally optimize the translated code for better performance.

After compiling, an ***intermediate code file*** (in assembly language) with .s extension is generated.

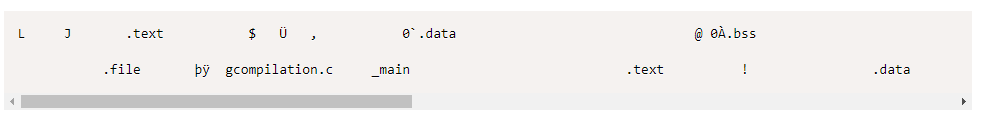
Let us look into compilation.s file:



## Assembling

Assembler accepts the intermediate code file and translates to machine code. After successful assembling, it generates .o file (on Linux) or .obj file (on Windows) known as ***object file***.

In our case, a compilation.o file is generated. It is encoded in machine language and cannot be viewed using text editors. However, if you still open it in a text editor, it looks like:



## Linking

Finally, the linker comes in action and performs the final task of compilation process. It accepts the object file. It links all the function calls with their original definition; that means the function printf() gets linked to its original definition.

The linker can generate one of following files based on your configuration:

* ***Executable file*** (no extension on Linux, or .exe on Windows).
* ***Static library*** (.a on Linux – also called *archive library*, or .lib *static library* on Windows).
* ***Dynamic library*** (.so on Linux – also called *shared object library*, or .dll *dynamic library* on Windows).

**Static Linking and Dynamic Linking**

|  |  |
| --- | --- |
| **Static linking** | **Dynamic linking** |
| Done by the linker in the final step of the compilation. | Done at run time by the OS. |
| Statically linked files consume more disk and memory as all the modules are already linked. | Only one copy of the reference module is stored which is used by many programs, thereby saving memory and disk space. |
| All the library modules are copied to the final executable image. When the program is loaded, the OS keeps only a single file in the memory which contains both the source code and the referencing libraries. | Only the names of external or shared libraries are kept in the memory. Dynamic linking lets many programs use single copy of executable module. |
| If external source program is changed then they have to be re-compiled and re-linked. | Only a single module needs to be updated and re-compiled. This is one of the greatest advantages that dynamic linking offers. |
| Statically linked programs are faster than their dynamic counterpart. | Dynamically linked programs are slower than their static counterpart. |
| Since the statically linked file contains every package and module, no compatibility issues occur. | Since the library files are separately stored there may be compatibility issues (say one library file is compiled by new version of compiler). |
| Statically linked programs always take constant load time. | The time is variable in dynamically linked programs. |

# GCC/G++ Compiler

Simple GCC command generator: <https://www.rapidtables.com/code/linux/gcc.html>

List of command options/flags: <https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html>

## Notes

### -pthread vs -lpthread

The pthread is POSIX threads library

We **should use -pthread**, and it’s **not recommended to use -lpthread** for two reasons:

* With modern Linux, using -lpthread alone might cause "Undefined reference to symbol 'pthread\_...".
* The linker takes libraries in order, and only takes as much as it needs to resolve references which are undefined at that point. So, you need to put -pthread after other static libs.

On the other hand, -pthread is considered as a standard flag for the preprocessor, so it should be used consistently for both compilation and linking.

## GDB Debugger

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjHGWPN_S9NS_Ky1-tC8ZrUI>

About **GUI for GDB**, there are many options. But I would suggest Eclipse (see *Tutorials/IDEs/Eclipse/Eclipse Tutorial.docx*) which runs based on Makefile and gcc/g++ (we can think that "Eclipse is like a front-end to GDB"), and offers both local and remote debugging.

# Clang/Clang++ Compiler

# Makefile

**Note**

This guide is for Linux. For Window, follow the same principle.

## What Is Makefile?

A Makefile is basically **a script which defines and controls the whole building process automatically**. It works based on a strict set of rules defined and implemented by the [***GNU make***](http://www.gnu.org/software/make/manual/make.html) program.

For example, let’s assume we have the following source files in the same directory:

// In main.c

#include <stdio.h>

#include "functions.h"

int main() {

   print\_hello();

   printf("The factorial of 5 is %d.\n"), factorial(5);

   return 0;

}

// In hello.c:

#include <stdio.h>

#include "functions.h"

void print\_hello() {

   prinf("Hello, World!\n");

}

// In factorial.c:

#include "functions.h"

int factorial(int n) {

   if (n != 1) {

      return (n \* factorial(n-1));

   } else {

      return 1;

   }

}

// In functions.h:

void print\_hello();

int factorial(int n);

The trivial way to compile these files into executable is:

$ gcc main.c hello.c factorial.c –I. -o hello

Or:

$ cc main.c hello.c factorial.c –I. -o hello

Or you can reverse the order of output and input files as follows:

$ cc -o hello main.c hello.c factorial.c –I.

For a large project where we might have hundreds of source files, it becomes extremely difficult and time-wasting to maintain the building process. Moreover, you might notice that you usually only work on a small section of the program (e.g., several functions in some files) and much of the remaining program is unchanged; re-compiling all source files is definitively unnecessary.

That's when a Makefile comes to play! The following sections describe how to write Makefile steps by steps.

## Rules

### File Name

A Makefile should be named as Makefile or Makefile.

### Target - Dependency

A target entry form looks like this:

target: dependency-list

    command

    command

    command

Where:

* The target is the output file. When you run make, the target serves as an argument ($ make <target>).
* The dependency-list is the list of input files. Multiple dependencies are separated by spaces.
* The commands are a series of steps used to make the target. Each command has to start with a tab character, NOT spaces.

### Flow

**The first target is referred as the default target which is called first when we run make**. The make program then looks at the default target's list of dependencies; if any of them are older, it will run the targets for those dependencies before running itself.

**Example 1**:

some\_file: other\_file

echo "This will run second, because it depends on 'other\_file'."

other\_file:

echo "This will run first."

Output:

$ make

This will run first.

This will run second, because it depends on 'other\_file'.

**Example 2**:

blah: blah.o

cc blah.o -o blah

blah.o: blah.c

cc -c blah.c -o blah.o

blah.c:

echo "int main() { return 0; }" > blah.c

clean:

rm -f blah.o blah.c blah

Output:

$ make

echo "int main() { return 0; }" > blah.c

cc -c blah.c -o blah.o

cc blah.o -o blah

$ make clean

rm -f blah.o blah.c blah

### Re-Compiling

The *GNU make* program is intelligent and works based on the changes you do in your source files. From the very first example, we know that all main.c, hello.c and factorial.c are dependent on functions.h, and main.c is dependent on both hello.c and factorial.c. If you make any changes in functions.h, then the *make* program will recompile all the source files to generate new object files. However, if you make a change in main.c, as this is not dependent of any other file, then only main.c file will be recompiled.

**While compiling a file, the *make* program checks its object file and compares the time stamps. If the source file has a newer time stamp than the object file, it will generate a new object file** assuming that the source file has been changed.

## Variables

<https://www.gnu.org/software/make/manual/html_node/Using-Variables.html>

## Macros

The *make* program allows to use macros, which are similar to variables and can only be strings.

### User-Defined Macros

files = file1 file2

some\_file: $(files)

echo "Look at this variable: " $(files)

touch some\_file

file1:

touch file1

file2:

touch file2

### Pre-Defined Macros

**Conventional Macros**

There are various pre-defined macros used in implicit rules. You can see them by typing make -p to print out the defaults. They fall into two classes:

1. Macros that are names of programs:

* CC: Program to compiling C programs. Default is cc.
* CXX: Program to compiling C++ programs. Default is g++.
* CPP: Program to running the C preprocessor. Default is $(CC) -E.
* LINT: Program to use to run lint on source code. Default is lint.
* RM: Command to remove a file. Default is rm -f.

2. Macros that contain arguments of the programs:

* ASFLAGS: Flags to give to the assembler when explicitly invoked on a '.s' or '.S' file.
* CFLAGS: Flags to give to the C compiler.
* CXXFLAGS: Flags to give to the C++ compiler.
* CPPFLAGS: Flags to give to the C preprocessor and programs, which use it (such as C and Fortran compilers).
* LDFLAGS: Flags to give to compilers when they are supposed to invoke the linker, 'ld'.
* LINTFLAGS: Flags to give to lint.

**Notes**:

* You can cancel all variables used by implicit rules with '-R' or '--no-builtin-variables' option.
* You can define macros at the command line as: $ make CPP = /home/courses/spring02

**Special Macros**

There are some special macros predefined, such as:

* $@ is the name of the output file being generated (left hand side of :).
* $? is the list of names of changed dependencies (
* $^ is the dependencies list (right hand side of :).
* $< is the first item in the dependencies list (first item at the right-hand side of :).
* $\* is the prefix shared by target and dependencies.

Example 1:

# $@ represents hello and $? picks up all the changed source files.

SRC = main.c factorial.c hello.c

CFLAG = -Wall -g

CC = gcc

hello: ${SRC}

${CC} ${CFLAGS} $? $(LDFLAGS) -o $@

Example 2:

# Make of .o files out of .c files

# Way 1:

%.o: %.c:

$(CC) $(CFLAGS) -c $^ -o $@

# Way2:

# Note: This way is old (not recommended). Should use the first way instead.

.c.o:

$(CC) $(CFLAGS) -c $^ -o $@

Alternatively:

.c.o:

$(CC) $(CFLAGS) -c $\*.c

## Examples

The following Makefile example is generic. You can apply it (and customize it) to many projects:

# Run 'make' or 'make all' to build executable file

# Run 'make clean' to remove all object files and executable files

# Run 'make depend' to use makedepend to automatically generate dependencies  (which are added to end of Makefile)

# Define the compiler to use ('gcc' if C, or 'g++' if C++)

CC = gcc

# Define compiler flags

#   For example, '-Wall' enables all warnings, '-g' adds debug info

CFLAGS = -Wall -g

# Define directories containing header files other than /usr/include

#   The rule is appending '-I' before the directory name

HEADERS = -I/home/triho/include  -I../include

# Define directories containing libraries other than /usr/lib

#   The rule is appending '-L' before the directory name

LFLAGS = -L/home/triho/lib  -L../lib

# Define names of libraries to link into executable:

#   The rule is appending '-l' before the lib name

#   For example, to link in libraries mylib.so or mylib.a, use -lmylib (no extension)

LIBS = -lmylib1 -lmylib2

# Define the C source files

SRCS =  emitter.c \

Better way: Search all .c files in dir: https://stackoverflow.com/a/3774731

        error.c \

        init.c \

        main.c \

        parser.c

# Define the C object files

#   This uses Suffix Replacement within a macro: $(name:oldstr=newstr)

#   For each word in 'name', replace 'oldstr' with 'newstr'

# Below we're replacing the suffix .c of all words in SRCS with the .o suffix

OBJS = $(SRCS:.c=.o)

# Define the executable file

EXE = mycc

# Running 'make' will invoke the first target entry in the file

# You can name this target entry anything, but "default" or "all" are the convention

all: $(EXE)

echo "All source files have been compiled"

Without it, still works

# Compiling: Create object files from source files

# Below we're using pre-defined macros:

#    '$^' is the list of names of the dependencies (.c files)

#    '$@' is the name of the target (.o file)

.c.o:

Another way:

%.o: %.c:

$(CC) $(CFLAGS) $(HEADERS) -c $^ -o $@

# Linking: Create the executable file from object files

$(EXE): $(OBJS)

$(CC) $(CFLAGS) $(HEADERS) $(OBJS) $(LFLAGS) $(LIBS) -o $(EXE)

# Running 'make clean' removes the executable file, all .o files and \*~ backup files

clean:

$(RM) \*.o \*~ $(EXE)

# Running 'make depend' generates dependencies of C source files automatically

# Install makedepend on Ubuntu: sudo apt-get install xutils-dev

depend: $(SRCS)

makedepend $(HEADERS) $^

# DO NOT DELETE THIS LINE -- make depend needs it

## Tips

### .PHONY

Adding .PHONY to a target prevents *make* from confusing the phony target with a file name.

In the below example, if there is a source code file named clean.c, then make clean will still run correctly.

.PHONY: clean

clean:

$(RM) \*.o \*~ $(EXECUTABLE)

### Makedepend (Auto Dependency Creation)

Install:

* Ubuntu: sudo apt-get install xutils-dev

<https://linux.die.net/man/1/makedepend>

[https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/make.html#Makefile\_depends](https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/make.html#makefile_depends)

### Others

* More features: [https://www.tutorialspoint.com/Makefile/Makefile\_features.htm](https://www.tutorialspoint.com/makefile/makefile_features.htm)
* Compile all C files at once: [here](https://stackoverflow.com/questions/170467/makefiles-compile-all-c-files-at-once)
* Using if … else in Makefile: [here](https://www.avrfreaks.net/forum/how-use-if-else-endif-makefile)
* Include other makefiles: by using the include directive.
* Debug mode vs release mode: [here](https://randu.org/tutorials/c/make.php)
* [CVS](https://www.gnu.org/software/automake/manual/html_node/CVS.html) is another tool that may be useful for very large projects. CVS stands for Concurrent Versions System and it allows you to record the history of your source files. CVS stores the base source and then stores the differences for each version. CVS also allows for protecting code pieces of a multi-developer effort from accidental overwriting.

# CMake

Tutorial: <https://cmake.org/cmake/help/latest/index.html>

Examples: <https://github.com/ttroy50/cmake-examples>

## What Is CMake?

While the make utility (or Makefile) is a buildsystem which drives the compiler and other build tools to build code, CMake is a **generator of buildsystems**. It can produce:

* Makefile (Unix, MinGW, NMake, etc.)
* Buildsystem files (Ninja, etc.)
* IDE project files (Visual Studio, XCode, Eclipse CDT, CodeBlock, KDevelop, etc.)

The best thing about CMake is that the **same CMakeLists.txt file is used across platforms**. So, if you're working on a platform-independent project (e.g. using MSVC compiler on Windows and g++ compiler on Linux), CMake is extremely convenient.

CMake is widely used for the C and C++ languages, but it may be used to build source code of other languages too.

## Installation

<https://cmake.org/install/>

## CLI

### CMake Help

D:\MYGIT\Personal\Tutorials\C-C++\Code\_C++\GTest\build>cmake --help

Usage

**cmake [options] <path-to-source>**

**cmake [options] <path-to-existing-build>**

**cmake [options] -S <path-to-source> -B <path-to-build>**

Specify a source directory to (re-)generate a build system for it in the

current working directory. Specify an existing build directory to

re-generate its build system.

Options

-S <path-to-source> = Explicitly specify a source directory.

-B <path-to-build> = Explicitly specify a build directory.

-C <initial-cache> = Pre-load a script to populate the cache.

-D <var>[:<type>]=<value> = Create or update a cmake cache entry.

-U <globbing\_expr> = Remove matching entries from CMake cache.

-G <generator-name> = Specify a build system generator.

-T <toolset-name> = Specify toolset name if supported by

generator.

-A <platform-name> = Specify platform name if supported by

generator.

--toolchain <file> = Specify toolchain file

[CMAKE\_TOOLCHAIN\_FILE].

--install-prefix <directory> = Specify install directory

[CMAKE\_INSTALL\_PREFIX].

-Wdev = Enable developer warnings.

-Wno-dev = Suppress developer warnings.

-Werror=dev = Make developer warnings errors.

-Wno-error=dev = Make developer warnings not errors.

-Wdeprecated = Enable deprecation warnings.

-Wno-deprecated = Suppress deprecation warnings.

-Werror=deprecated = Make deprecated macro and function warnings

errors.

-Wno-error=deprecated = Make deprecated macro and function warnings

not errors.

--preset <preset>,--preset=<preset>

= Specify a configure preset.

--list-presets[=<type>] = List available presets.

-E = CMake command mode.

-L[A][H] = List non-advanced cached variables.

--fresh = Configure a fresh build tree, removing any

existing cache file.

--build <dir> = Build a CMake-generated project binary tree.

--install <dir> = Install a CMake-generated project binary

tree.

--open <dir> = Open generated project in the associated

application.

-N = View mode only.

-P <file> = Process script mode.

--find-package = Legacy pkg-config like mode. Do not use.

--graphviz=<file> = Generate graphviz of dependencies, see

CMakeGraphVizOptions.cmake for more.

--system-information [file] = Dump information about this system.

--log-level=<ERROR|WARNING|NOTICE|STATUS|VERBOSE|DEBUG|TRACE>

= Set the verbosity of messages from CMake

files. --loglevel is also accepted for

backward compatibility reasons.

--log-context = Prepend log messages with context, if given

--debug-trycompile = Do not delete the try\_compile build tree.

Only useful on one try\_compile at a time.

--debug-output = Put cmake in a debug mode.

--debug-find = Put cmake find in a debug mode.

--debug-find-pkg=<pkg-name>[,...]

= Limit cmake debug-find to the

comma-separated list of packages

--debug-find-var=<var-name>[,...]

= Limit cmake debug-find to the

comma-separated list of result variables

--trace = Put cmake in trace mode.

--trace-expand = Put cmake in trace mode with variable

expansion.

--trace-format=<human|json-v1>

= Set the output format of the trace.

--trace-source=<file> = Trace only this CMake file/module. Multiple

options allowed.

--trace-redirect=<file> = Redirect trace output to a file instead of

stderr.

--warn-uninitialized = Warn about uninitialized values.

--no-warn-unused-cli = Don't warn about command line options.

--check-system-vars = Find problems with variable usage in system

files.

--compile-no-warning-as-error= Ignore COMPILE\_WARNING\_AS\_ERROR property and

CMAKE\_COMPILE\_WARNING\_AS\_ERROR variable.

--profiling-format=<fmt> = Output data for profiling CMake scripts.

Supported formats: google-trace

--profiling-output=<file> = Select an output path for the profiling data

enabled through --profiling-format.

-h,-H,--help,-help,-usage,/? = Print usage information and exit.

--version,-version,/V [<file>]

= Print version number and exit.

--help <keyword> [<file>] = Print help for one keyword and exit.

--help-full [<file>] = Print all help manuals and exit.

--help-manual <man> [<file>] = Print one help manual and exit.

--help-manual-list [<file>] = List help manuals available and exit.

--help-command <cmd> [<file>]= Print help for one command and exit.

--help-command-list [<file>] = List commands with help available and exit.

--help-commands [<file>] = Print cmake-commands manual and exit.

--help-module <mod> [<file>] = Print help for one module and exit.

--help-module-list [<file>] = List modules with help available and exit.

--help-modules [<file>] = Print cmake-modules manual and exit.

--help-policy <cmp> [<file>] = Print help for one policy and exit.

--help-policy-list [<file>] = List policies with help available and exit.

--help-policies [<file>] = Print cmake-policies manual and exit.

--help-property <prop> [<file>]

= Print help for one property and exit.

--help-property-list [<file>]= List properties with help available and

exit.

--help-properties [<file>] = Print cmake-properties manual and exit.

--help-variable var [<file>] = Print help for one variable and exit.

--help-variable-list [<file>]= List variables with help available and exit.

--help-variables [<file>] = Print cmake-variables manual and exit.

Generators

The following generators are available on this platform (\* marks default):

\* Visual Studio 17 2022 = Generates Visual Studio 2022 project files.

Use -A option to specify architecture.

Visual Studio 16 2019 = Generates Visual Studio 2019 project files.

Use -A option to specify architecture.

Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.

Optional [arch] can be "Win64" or "ARM".

Visual Studio 12 2013 [arch] = Deprecated. Generates Visual Studio 2013

project files. Optional [arch] can be

"Win64" or "ARM".

Borland Makefiles = Generates Borland makefiles.

NMake Makefiles = Generates NMake makefiles.

NMake Makefiles JOM = Generates JOM makefiles.

MSYS Makefiles = Generates MSYS makefiles.

MinGW Makefiles = Generates a make file for use with

mingw32-make.

Green Hills MULTI = Generates Green Hills MULTI files

(experimental, work-in-progress).

Unix Makefiles = Generates standard UNIX makefiles.

Ninja = Generates build.ninja files.

Ninja Multi-Config = Generates build-<Config>.ninja files.

Watcom WMake = Generates Watcom WMake makefiles.

CodeBlocks - MinGW Makefiles = Generates CodeBlocks project files

(deprecated).

CodeBlocks - NMake Makefiles = Generates CodeBlocks project files

(deprecated).

CodeBlocks - NMake Makefiles JOM

= Generates CodeBlocks project files

(deprecated).

CodeBlocks - Ninja = Generates CodeBlocks project files

(deprecated).

CodeBlocks - Unix Makefiles = Generates CodeBlocks project files

(deprecated).

CodeLite - MinGW Makefiles = Generates CodeLite project files

(deprecated).

CodeLite - NMake Makefiles = Generates CodeLite project files

(deprecated).

CodeLite - Ninja = Generates CodeLite project files

(deprecated).

CodeLite - Unix Makefiles = Generates CodeLite project files

(deprecated).

Eclipse CDT4 - NMake Makefiles

= Generates Eclipse CDT 4.0 project files

(deprecated).

Eclipse CDT4 - MinGW Makefiles

= Generates Eclipse CDT 4.0 project files

(deprecated).

Eclipse CDT4 - Ninja = Generates Eclipse CDT 4.0 project files

(deprecated).

Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files

(deprecated).

Kate - MinGW Makefiles = Generates Kate project files (deprecated).

Kate - NMake Makefiles = Generates Kate project files (deprecated).

Kate - Ninja = Generates Kate project files (deprecated).

Kate - Ninja Multi-Config = Generates Kate project files (deprecated).

Kate - Unix Makefiles = Generates Kate project files (deprecated).

Sublime Text 2 - MinGW Makefiles

= Generates Sublime Text 2 project files

(deprecated).

Sublime Text 2 - NMake Makefiles

= Generates Sublime Text 2 project files

(deprecated).

Sublime Text 2 - Ninja = Generates Sublime Text 2 project files

(deprecated).

Sublime Text 2 - Unix Makefiles

= Generates Sublime Text 2 project files

(deprecated).

### Quick Start Example

Assumpt that a CMakeLists.txt is available.

#### Step 1: Configuration

|  |  |
| --- | --- |
| Way 1:  # From the directory of source code, create a build directory  $ cd myproject  $ mkdir build  # In the build directory, run cmake to configure the project  # and generate a native build system  $ cd build  $ cmake .. | Way 2:  # From the directory of source code,  # create the build folder named "build" (if not exist)  # Then configure the project and generate a native build system  $ cd myproject  $ cmake -B build -S .  # -B build: Specifies the build directory  # where the build files will be generated.  # -S .: Specifies the source directory, which in this case is the current directory. |

Explanation:

* The **generated files** will be placed in the build directory. Note that using a separate build directory (also known as "out-of-source" build) keeps the source directory clean and allows for multiple build configurations (like debug and release) without conflicts.
* CMake will look for the CMakeLists.txt file in the parent directory. So make sure you have it before hand. We will learn about this file in the later sections.
* In the configuration step, CMake will:
  + Parses the CMakeLists.txt file. If any error is found, it stops and displays the error message.
  + Configures project:
    - Specifies CMake minimum version required to build the project.
    - Sets project name and version.
    - Define targets, like executables and libraries.
    - Etc.
  + Generates a native build system:
    - The native build system can be Makefiles, Visual Studio project file, etc. based on the platform you are using.
    - These files will be used later when you run cmake --build . to compile the project.
  + Cache variables:
    - CMake **creates a cache file** (usually named CMakeCache.txt) in the build directory to store configuration options and paths to dependencies. It allows CMake to remember certain settings between runs. Also you can read this file to understand all the settings CMake generated.

Notes:

* + This first step is **one-time setting**. It means that once you've successfully configured the project, you **generally do not need to run it again for every source code modification**. The key point behind is that the CMakeCache.txt retains configuration information. If your changes don’t invalidate this cache, CMake might not require a full reconfiguration for the build to succeed.

Generally speaking:

* + - If you do not change key settings (like the project name or target definitions), don't need to re-configure.
    - If you do change minor settings, like adding new libraries or changing compiler flags, don't need to re-configure.

However, **although it might work fine, the best practice is to always rerun the configuration step after making any changes to the CMakeLists.txt file**.

#### Step 2: Compilation

|  |  |  |
| --- | --- | --- |
| Way 1:  # In the build directory, call the build system to  # actually compile and link the project  $ cmake --build . | Way 2:  # In the source directory  $ cmake --build build | Way 3:  # In the build directory  $ make # if Make build system  $ ... # if other build system |

Tips:

* + If you execute step 2 but forgot the step 1, you'll get error: *Error: could not load cache*.

#### Step 3: Installation

|  |  |  |
| --- | --- | --- |
| Way 1:  # In the build directory, copy the targets (executable, libs, etc.) into the installation directory defined by install().  $ cmake --install . | Way 2:  # In the source directory  $ cmake --install build | Way 3:  # In the build directory  $ make install # if Make build system  $ ... # if other build system |

Notes:

* + The default installation directory is:
    - In Linux:
    - Binaries: /usr/local/bin
    - Libraries: /usr/local/lib
    - Include Files: /usr/local/include
    - In Windows:
    - Binaries: C:\Program Files\YourProjectName\bin
    - Libraries: C:\Program Files\YourProjectName\lib
    - Include Files: C:\Program Files\YourProjectName\include

#### Step 4: Running

|  |  |  |
| --- | --- | --- |
| Way 1:  # In the build directory  $ cmake --build . --target <your-exe-name> | Way 2:  # In the source directory  $ cmake --build build --target <your-exe-name> | Way 3:  # Locate the exe file and run it  $ ./<your-exe-name> |

### Other Examples

#### Setting CMake Generator

The cmake -G option is used to specify the generator that CMake will use to create the build system files.

Syntax:

$ cmake -G "GeneratorName" path/to/source

Examples:

* + Makefiles generator for Unix systems: $ cmake -G "Unix Makefiles" ..
  + Makefiles generator for MinGW on Windows: $ cmake -G "MinGW Makefiles" ..
  + Makefiles generator for Visual Studio projects: $ cmake -G "Visual Studio 16 2019" ..
  + Makefiles generator for Ninja build files: $ cmake -G "Ninja" ..

## CMakeList File

### Built-In Functions

|  |  |  |
| --- | --- | --- |
| **Command** | **Usage** | **Note** |
| **General** | | |
| set(  <var\_name>  <value>  [options]  ) | Create and set value for variables.   * Type: Variables can be string or list. * Override or append:   + If the variable already exists, the function will overwrite its value.   + You can choose to append values to an existing variable by concatenating. * Cache: You can store variables in the CMake cache, making them persistent across CMake runs. Note that to clear or reset the cache, you have to either delete the CMakeCache.txt or the entire build folder.   To use variables, refer them with ${var\_name} syntax in case of string variable, or with ${var\_name\_index} syntax in case of separate item in list variable. | # Set a string variable  set(  HEADER\_PATH  project/src/include)  # Set a list variable  set(  MY\_LIST  "Item1" "Item2")  # Append value to the list  set(  MY\_LIST  "${MY\_LIST};Item3")  # Store variables in CMake cache  set(  MY\_OPTION  "Default"  CACHE STRING "Example option")  # Using a string variable  message(  "MY\_VAR is: ${MY\_VAR}")  # Using a list variable  message(  "First item: ${MY\_LIST}")  "Second item: ${MY\_LIST\_1}") |
| message(  [<mode>]  "message to display"  ) | Display messages to the user during the configuration process.  Modes:   * STATUS: Info. Prefixed with --. * WARNING: Warn. Prefixed with CMake Warning, together with code line. * AUTHOR\_WARNING: Warnings for authors. Prefixed with CMake Author Warning, together with code line. * SEND\_ERROR: Error but does not stop the configuration process. Prefixed with CMake Error. * FATAL\_ERROR: Error and stops the configuration process. Prefixed with CMake Error. * DEPRECATION: Indicating that a feature or function is outdated. Prefixed with CMake Deprecation Warning, together with code line. |  |
| include(  <file\_path>  [OPTIONAL]  [RESULT <var>]  [NO\_POLICY\_SCOPE]  ) | Include and execute other CMake scripts with .cmake extension within the current CMake context.   * If OPTIONAL is present, then no error is raised if the file does not exist. | # Include another.cmake  include(another)  # Include subpath/another.cmake  include(subpath/another) |
| **Build System** | | |
| cmake\_minimum\_required(  VERSION <version>  ) | Specify a minimum CMake version  *Compulsary. Every CMakeLists.txt must start with it.* | cmake\_minimum\_required(  VERSION 3.10  ) |
| project(  <project\_name>  VERSION <version>  LANGUAGES <language>  ) | Set the project name and version (if needed)  *Compulsary. Should be called after cmake\_minimum\_required()* | project(Tutorial)  project(Tutorial VERSION 1.0)  project(  Tutorial  VERSION 1.0  LANGUAGES CXX  ) |
| configure\_file(  <input\_file>  <output\_file>  [COPYONLY]  ) | Copy a given input file to an output file and substitute some variable values in the input file content | configure\_file(  TutorialConfig.h.in  TutorialConfig.h  ) |
|  |  |  |
| **Build** | | |
| add\_executable(  <target\_name>  <src1> <src2> ...  ) | Create an executable using the specified source code files.  *Used when you want to build executable.* | add\_executable(Tutorial  tutorial.cxx  ) |
| add\_library(  <target\_name>  [type]  <src1> <src2> ...  ) | Create a library target called <target\_name> to be built from the source files listed in the command.   * <type> can be: * STATIC: A static library or an archive of object files for use when linking other targets. * SHARED: A dynamic dynamic library that may be linked by other targets and loaded at runtime. * MODULE: A plugin that may not be linked by other targets, but may be dynamically loaded at runtime using dlopen-like functionality.   If no <type> is given, the default is STATIC or SHARED based on the BUILD\_SHARED\_LIBS variable. | add\_library(MathFunctions STATIC  MathFuncs.cxx mysqrt.cxx  ) |
| add\_library(  <target\_name>  INTERFACE  ) | Add an *interface library* target that may specify usage requirements for dependents but does not compile sources and does not produce a library artifact on disk. |  |
| add\_subdirectory(  <directory>  [binary\_dir]  ) | Add the subdirectory to the build.  *Used when source files are located in multiple sub-directories. Executing the top-level CMakeLists file will trigger executing CMakeLists files from subdirectories.* | add\_subdirectory(  MathFuncs) |
| target\_include\_directories(  <target\_name>  PRIVATE|PUBLIC|INTERFACE  <dir1> <dir2> ...  ) | Specify where the given executable target or library target should look for header files.  *Note: CMake does not recommend using GLOB to collect a list of files from your source tree. If no CMakeLists.txt file changes when a source is added or removed, then CMake cannot know when to regenerate the build. So, we have to add each file explicitly in the CMakeLists.txt.* | target\_include\_directories(  Tutorial PUBLIC  "${CMAKE\_CURRENT\_SOURCE\_DIR}}"  ) |
| include\_directories(  dir1> <dir2> ...  ) | Specify where all targets in directory level should look for header files | include\_directories(  "${CMAKE\_CURRENT\_SOURCE\_DIR}}"  ) |
| target\_link\_libraries(  <target\_name>  PRIVATE|PUBLIC|INTERFACE  <lib1> <lib2> ...  ) | Link the executable target to the library | target\_link\_libraries(  Tutorial PUBLIC  MathFunctions  ) |
| target\_link\_directories(  <target\_name>  [BEFORE]  <INTERFACE|PUBLIC|PRIVATE>  <dir1> <dir2> ...  ) | Specify additional directories to search for libraries when linking a target.  Parameters:   * <target\_name>: The name of the target to link against. * BEFORE: If specified, the provided directories are added to the front of the search path, giving them higher priority than previously specified directories. * <INTERFACE|PUBLIC|PRIVATE>: * INTERFACE: The directory is used by targets that link against this target. * PUBLIC: The directory is used by both this target and any targets that link against it. * PRIVATE: The directory is only used by this target. * <dir1> <dir2> ...: One or more directories to look for libraries.   *While target\_link\_libraries() directly defines which libraries to link with the target, target\_link\_directories() specifies where to find those libraries, enhancing the linking process.* | # Specify where to find additional libraries  target\_link\_directories(  my\_library  PRIVATE  /path/to/my/libs  )  # Link the library with the target  target\_link\_libraries(  my\_library  PRIVATE  my\_other\_library  ) |
| find\_package(  <package\_name>  [version]  [REQUIRED|QUIET|EXACT]  [COMPONENTS cpn1 cpn2 ...]  ) | Locate and configure external packages (libraries, tools, etc.) needed for your project.  *CMake can find packages in two ways:*   * ***Config mode****: Looks for a configuration file provided by the package (e.g., <PackageName>Config.cmake).* * ***Module mode****: Searches for a Find<PackageName>.cmake module, which is a script that defines how to locate and configure the package.* *These modules are part of the CMake distribution, so you don't need to create this file yourself unless you have specific needs. They’re inthe following directory:* * *On Linux: /usr/share/cmake-<version>/Modules* * *On macOS: /Applications/CMake.app/Contents/share/cmake-<version>/Modules* * *On Windows: C:\Program Files\CMake\share\cmake-<version>\Modules*   *Common modules are FindBoost.cmake, FindQt.cmake, FindPython.cmake, FindGTest.cmake, etc.*  *Additional Options*   * *QUIET: Suppresses messages if the package is not found.* * *EXACT: Ensures that the exact version specified is found.* * *CONFIG: Forces CMake to look for the package in config mode.* | # Find a boost install  # with the libraries  # "filesystem" and "system"  find\_package(  Boost 1.46.1  REQUIRED  COMPONENTS filesystem system  ) |
| find\_path(  <output\_path>  <file\_name>  [PATHS path1 path2 ...]  [PATH\_SUFFIXES suf1 suf2 ...]  ) | Locate a directory that contains a specified file.   * output\_path: Variable name where the result will be stored. * file\_name: Name of file you're searching for. * PATHS: List of directories to search in. If not specified, CMake uses its default search paths, e.g. CMAKE\_INCLUDE\_PATH, CMAKE\_LIBRARY\_PATH, CMAKE\_PREFIX\_PATH, etc. * PATH\_SUFFIXES: Additional suffixes to append to the search paths. This is useful for searching in subdirectories.   *This function is useful for finding header files or other resources necessary for building your project.* | # Find path to MyHeader.h  # The output path is HEADER\_PATH  find\_path(  HEADER\_PATH  MyHeader.h  PATHS  "/usr/include"  "/usr/local/include"  PATH\_SUFFIXES include  ) |
| **Compiler Flags** | | |
| target\_compile\_definitions(  <target-name>  [INTERFACE|PUBLIC|PRIVATE]  <definition> [...]  ) | Specify compiler preprocessor definitions for a specific target.  *This is useful for defining macros that can be used in your code, enabling or disabling features based on build configurations.* | # Define DEBUG\_MODE for MyExecutable  target\_compile\_definitions(  MyExecutable  PRIVATE  DEBUG\_MODE)  So, in your code, you can:  #ifdef DEBUG\_MODE  ...  #endif |
| target\_compile\_options(  <target>  [<INTERFACE|PUBLIC|PRIVATE>]  <options>...  ) | Specify compiler options for a specific target | target\_compile\_options(  MyExecutable  PRIVATE  -Wall -O2) |
| **Installation** | | |
| install(  <type>  <target> [...]  ) | Specify rules for installing targets, files, or directories to a specified location.  Common types of installable items:   * + TARGETS: Executables or libraries.   + FILES: Specific files (e.g., configuration files, headers).   + DIRECTORY: Entire directories (including the directories themself).   + ...   *You need to execute command cmake --install . to trigger installation. After that, CMake generates an install\_manifest.txt file*  *including details on all installed files. You should check it.* | install(TARGETS MyExecutable  DESTINATION bin)  install(FILES mylib.h  DESTINATION include)  install(DIRECTORY mydir/  DESTINATION share/mypackage)  # all files in a directory  file(GLOB ALL\_FILES "mydir/\*")  install(FILES ${ALL\_FILES}  DESTINATION /usr/lib) |
| **Testing** | | |
| add\_test(  NAME <test\_name>  COMMAND <command>  [args...]  ) | Specify a test that can be executed by the ctest command.   * + NAME of the test. Should be unique within the project. Will be displayed in the test result output.   COMMAND to run the test. Can be the name of an executable, a script, or any command-line executable. | # Assuming you have an executable  # called 'test\_shapes'  add\_test(  NAME CircleAreaTest  COMMAND test\_shapes) |
| enable\_testing() | Enable testing for current directory and below.  *Automatically invoked when the ctest module is included, except if the BUILD\_TESTING option is turned off.* |  |
| **Running** | | |
| add\_custom\_target(  target\_name  [ALL]  [COMMAND cmd1 [args...]]  [COMMAND cmd2 [args...]]  ...  [DEPENDS depend1 depend2 ...]  [WORKING\_DIRECTORY dir]  [COMMENT comment]  [VERBATIM | APPEND]  [SOURCES src1 src2 ...]  ) | Create a custom target that can be built as part of the overall build process.  *Unlike regular targets (like executables or libraries), a custom target does not produce an output file; instead, it is often used to* ***run commands or scripts*** *that perform tasks related to the project. This makes it a powerful and flexible tool.* | # The "run" target executes the myexe  add\_executable(myexe main.cpp)  add\_custom\_target(run  COMMAND myexe  DEPENDS myexe  WORKING\_DIRECTORY ${CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY}  COMMENT "Running the executable"  )  # The "clean\_all" target removes the build directory  add\_custom\_target(clean\_all  COMMAND rm -rf build/  COMMAND echo "Cleaned build directory"  COMMENT "Cleaning build directory"  ) |
| **Operator** | | |
| STREQUAL |  |  |
| MATCHES |  |  |
| LIST |  |  |
| FOR |  |  |
| function()  endfunction() |  |  |
| if()  endif() |  |  |
| option() |  |  |

### Built-In Variables

|  |  |  |
| --- | --- | --- |
| **Command** | **Usage** | **Example** |
| **General** | | |
| CMAKE\_VERBOSE\_MAKEFILE |  | set(CMAKE\_VERBOSE\_MAKEFILE ON) |
| **CMake Meta Info** | | |
| CMAKE\_ROOT | CMake installation directory. |  |
| CMAKE\_COMMAND | The path of the CMake instance which runs currently (e.g. /usr/local/bin/cmake).  *Setting this variable is useful when CMake might not be on the system PATH.* |  |
| **Language** | | |
| CMAKE\_CXX\_COMPILER | Set the C++ compiler being used for the project.  *By default, CMake automatically detects the C++ compiler available on your system during the configuration step. This is typically determined based on the environment and the compiler tools installed.* |  |
| CMAKE\_C\_COMPILER | Set the C compiler being used for the project.  *By default, CMake automatically detects the C compiler available on your system during the configuration step. This is typically determined based on the environment and the compiler tools installed.* |  |
| CMAKE\_LINKER | Set the linker being used when linking executables or shared libraries for the project.  *By default, CMake automatically detects the linker for the given platform and compiler. For example, on Unix-like systems, it might be set to ld, while on Windows, it could be set to link.exe.* |  |
| CMAKE\_CXX\_STANDARD | Set the C++ standard being used for the project. | set(CMAKE\_CXX\_STANDARD 11) |
| CMAKE\_CXX\_STANDARD\_REQUIRED | Set a boolean that determines whether the specified C++ standard (set by CMAKE\_CXX\_STANDARD) is required to compile the project. | set(CMAKE\_CXX\_STANDARD\_REQUIRED TRUE) |
| **Project Info** | | |
| CMAKE\_PROJECT\_NAME | Get the name of the first project set by project(), i.e. the top level project. |  |
| PROJECT\_NAME | Get the name of the project set by project(). |  |
| **Project Locations / Paths**  myproject/  ├── CMakeLists.txt  ├── src/  │   ├── CMakeLists.txt  │   ├── main.cpp  │   └── utils.cpp  ├── include/  │   └── utils.h  ├── build/  └── tests/      ├── CMakeLists.txt      └── test\_main.cpp | | |
| CMAKE\_SOURCE\_DIR | Get top-level source directory of the project.  *Regardless of where you're in the directory structure, it's always the top-level CMakeLists.txt.* | myproject  Regardless of where you're in the directory structure (e.g., in src/CMakeLists.txt), it will always be myproject. |
| CMAKE\_BINARY\_DIR | Get top-level binary directory where build files are generated. | myproject/build  Regardless of where you're in the directory structure (e.g., in src/CMakeLists.txt), it will always be myproject/build. |
| CMAKE\_CURRENT\_SOURCE\_DIR | Get source directory of the currently processed CMakeLists.txt.  *It changes as CMake processes different directories. If you're in a subdirectory and have a CMakeLists.txt there, it'll point to that subdirectory.* | For myproject/CMakeLists.txt:   * + myproject   For myproject/src/CMakeLists.txt:   * + myproject/src   For myproject/tests/CMakeLists.txt:   * + myproject/tests |
| CMAKE\_CURRENT\_BINARY\_DIR | Get binary directory for the currently processed CMakeLists.txt. | For myproject/CMakeLists.txt:   * + myproject/build   For myproject/src/CMakeLists.txt:   * + myproject/src/build   For myproject/tests/CMakeLists.txt:   * + myproject/tests/build |
| PROJECT\_SOURCE\_DIR | Get source directory of the current project.  *This variable is set only once when project() is called, and it reflects the directory where that command was invoked.*  *In most cases, CMAKE\_CURRENT\_SOURCE\_DIR and PROJECT\_SOURCE\_DIR print the same value.* | For myproject/CMakeLists.txt:   * + myproject   For myproject/src/CMakeLists.txt:   * + myproject/src   For myproject/tests/CMakeLists.txt:  myproject/tests |
| PROJECT\_BINARY\_DIR | Get binary directory of the current project.  *In most cases, CMAKE\_CURRENT\_BINARY\_DIR and PROJECT\_BINARY\_DIR print the same value.* | For myproject/CMakeLists.txt:   * + myproject/build   For myproject/src/CMakeLists.txt:   * + myproject/src/build   For myproject/tests/CMakeLists.txt:  myproject/tests/build |
| CMAKE\_INCLUDE\_PATH | Set additional directories to be searched for header files. This is helpful to locate header files that are not in the default search paths.  To set multiple directories, separating each other with a semicolon.  *You might need to set this variable when using find\_path()*, *include\_directories()*. | set(  CMAKE\_INCLUDE\_PATH  "/include/path1;/include/path2"  ) |
| CMAKE\_LIBRARY\_PATH | Set additional directories to be searched for library files. This is helpful to locate library files that are not in the default search paths.  To set multiple directories, separating each other with a semicolon.  *You might need to set this variable when using find\_path()*. |  |
| CMAKE\_PREFIX\_PATH | Set additional directories to be searched for packages and other resources. This is helpful to locate header files, library files, and other components in custom or non-standard installation locations.  To set multiple directories, separating each other with a semicolon.  *You might need to set this variable when using find\_path()*, *find\_package()*, *find\_program()*, *find\_library()*. |  |
| CMAKE\_MODULE\_PATH | Set a list of directories to search for additional \*.cmake file. By default, if you do not set it, this variable is empty.  *You might need to set this variable when using find\_package() or include().* | list(  APPEND  CMAKE\_MODULE\_PATH  "${CMAKE\_SOURCE\_DIR}/cmake")  # Loads ${CMAKE\_SOURCE\_DIR}/cmake/Foo.cmake  include(Foo)  # Loads ${CMAKE\_SOURCE\_DIR}/cmake/FindBar.cmake  find\_package(Bar) |
| CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY | Set the directory where executable files (binaries) will be placed after built.  *By default, if you do not set it, this variable is empty and the executables will be placed in the build directory.* | set(  CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY  "${CMAKE\_BINARY\_DIR}/bin"  ) |
|  |  |  |
|  |  |  |
| **Project Build** | | |
| CMAKE\_BUILD\_TYPE | Set the build type on *single-configuration generators* (e.g. Makefile Generators or Ninja)  Typical build types are:   * + Release - Adds the -O3 -DNDEBUG flags to the compiler. Meaning omitting debug symbols and including high optimizations.   + Debug - Adds the -g flag. Meaning including debug symbols and omitting optimizations.   + MinSizeRel - Adds -Os -DNDEBUG. Meaning omitting debug symbols and including best optimizations.   + RelWithDebInfo - Adds -O2 -g -DNDEBUG flags. Meaning including debug symbols and including medium optimizations.   Also, custom build types can be defined.   * *The value of this variable is case-sensitive. So the value in the CMakeLists.txt should be same as in command cmake --build . --config <build-type>.* * *In multi-config generators, this variable is ignored.* | if(CMAKE\_BUILD\_TYPE STREQUAL "Debug")  ...  endif() |
| CMAKE\_CONFIGURATION\_TYPES | Set the build types on *multi-config generators* (e.g. Visual Studio, Xcode, or Ninja Multi-Config) as a semicolon-separated list.  Typical build types are:   * + Release - Adds the -O3 -DNDEBUG flags to the compiler   + Debug - Adds the -g flag   + MinSizeRel - Adds -Os -DNDEBUG   + RelWithDebInfo - Adds -O2 -g -DNDEBUG flags   Also, custom build types can be defined.   * *The value of this variable is case-sensitive*. *So the value in the CMakeLists.txt shouldd be same as in command cmake --build . --config <build-type>.* * *In single-config generators, this variable is ignored.* | string(TOLOWER ${CMAKE\_BUILD\_TYPE} build\_type)  if (build\_type STREQUAL debug)  target\_compile\_definitions(exe1  PRIVATE  DEBUG\_BUILD)  endif() |
| **Project Installation** | | |
| CMAKE\_INSTALL\_PREFIX | Set default installation directory for the project.  By default, it is set to /usr/local on Unix-like systems and C:\Program Files on Windows. | # "/usr/local/bin" 🡪 "/usr/bin"  # "/usr/local/lib" 🡪 "/usr/lib"  # etc  set(CMAKE\_INSTALL\_PREFIX "/usr/") |
| **Environment** | | |
| $ENV{name} | This is not an environment variable, but this is how you can access environment variables from CMake files. It returns the content of the environment variable with the given name (e.g. $ENV{PROGRAMFILES}). |  |
| **Static Analysis** | | |
| <LANG>\_CPPCHECK | <LANG> could be C or CXX. Set command line for running Cppcheck for that language.  *The Makefile generator and the Ninja generator will run Cppcheck along with the compiler and report any problems.*  *Multiple commands are separated by semi-colons.* | set(  CMAKE\_C\_CPPCHECK  "cppcheck --enable=all") |
| CMAKE\_<LANG>\_CPPCHECK | Default value for <LANG>\_CPPCHECK.  *If <LANG>\_CPPCHECK is not defined, CMake will use CMAKE\_<LANG>\_CPPCHECK.* |  |
| <LANG>\_CPPLINT | <LANG> could be C or CXX. Set command line for running CppLint for that language.  *The Makefile generator and the Ninja generator will run CppLint along with the compiler and report any problems.*  *Multiple commands are separated by semi-colons.* |  |
| CMAKE\_<LANG>\_CPPLINT | Default value for <LANG>\_CPPLINT.  *If <LANG>\_LINT is not defined, CMake will use CMAKE\_<LANG>\_CPPLINT.* |  |
| <LANG>\_CLANG\_TIDY | <LANG> could be C, CXX, OBJC or OBJCXX. Set command line for running Clang-Tidy for that language.  *The Makefile generator and the Ninja generator will run CppLint along with the compiler and report any problems.*  *Multiple commands are separated by semi-colons.* |  |
| CMAKE\_<LANG>\_CLANG\_TIDY | Default value for <LANG>\_CLANG\_TIDY.  *If <LANG>\_CLANG\_TIDY is not defined, CMake will use CMAKE\_<LANG>\_CLANG\_TIDY.* |  |
| <LANG>\_INCLUDE\_WHAT\_YOU\_USE | <LANG> could be C or CXX. Set command line for running include-what-you-use for that language.  *The Makefile generator and the Ninja generator will run include-what-you-use along with the compiler and report any problems.*  *Multiple commands are separated by semi-colons.* |  |
| CMAKE\_<LANG>\_INCLUDE\_WHAT\_YOU\_USE | Default value for <LANG>\_INCLUDE\_WHAT\_YOU\_USE.  *If <LANG>\_INCLUDE\_WHAT\_YOU\_USE is not defined, CMake will use CMAKE\_<LANG>\_INCLUDE\_WHAT\_YOU\_USE.* |  |
|  |  |  |

Full list: <https://cmake.org/cmake/help/latest/manual/cmake-variables.7.html>

### Visibility of Target Property

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Visibility** | **Usage Scenario** |
| PRIVATE | Visible to only to the target itself | Used for internal settings not needed by other targets |
| PUBLIC | Visible to the target itself and to targets that link to it | Used for shared settings needed by both the target and its consumers |
| INTERFACE | Visibile only to targets that link to it | Used for settings that are relevant only to the consumers, not the target itself |

What is "target that link to it"? For examples:

* If the target itself is a **library** (static or shared), then the one that links to it is an **executable** that calls the library.
* If the target itself is a **header file**, then the one that links to it is a **library** or an **executable** that uses the header file.
* Etc.

Examples: Check [this session](#_Visibility_of_Target).

## CMake File

### Overview

A parent CMakeLists.txt can load and execute code from external .cmake files. This is great for maintaining modular build systems, where common code — such as functions, macros, or configuration settings—can be shared across multiple CMakeLists.txt files or projects.

All syntax of CMakeLists.txt are applied to .cmake files. No difference!

### Usage

To include a .cmake file in a parent CMakeLists.txt, use the include() command followed by the filename or module name.

CMake looks for .cmake files in a few places:

* First, it checks the directory where the parent CMakeLists.txt is.
* Then, it looks in directories listed in CMAKE\_MODULE\_PATH, which you can set to include places like a cmake subdirectory.
* Finally, it checks CMake’s own module directory for built-in files.

For example:

# If myfile.cmake is in the same directory as CMakeLists.txt

include(myfile.cmake)

# Or you need to tell CMake in a more specific way by using its built-in module path

list(APPEND CMAKE\_MODULE\_PATH ${CMAKE\_CURRENT\_SOURCE\_DIR}/cmake)

include(myfile)

# Or by using the full file path

include(${CMAKE\_SOURCE\_DIR}/cmake/myfile.cmake)

### Scope

When using include(), code in the included file is executed in the scope of the including file. This dynamic scoping means:

* Variables set in the included file are visible in the including file.
* Variables in the including file are available in the included file.

For example:

set(VAR 1)

include(mycmake.cmake)    # Inside we have: set(VAR 2)

message(${VAR})           # prints: 2

set(VAR 3)

message(${VAR})           # prints: 3

### Notes

* Variables set in included files affect the parent, causing unexpected behavior.

To **isolate scope**, you can define functions or macros within the included file:

# In myfile.cmake

function(my\_function)

    set(LOCAL\_VAR "inside function")

    # Function body

endfunction()

* For big projects, consider add\_subdirectory() for subprojects with their own targets, rather than include() for everything.

This allows each subdirectory to have its own CMakeLists.txt and manage its own targets, reducing the risk of scope conflicts.

### Tips

* Think of header file (.h) in C/C++ where the whole content of the file is copied into the implementation file (.c, .cpp) at the position of the #include. The behavior of include() in CMake is just similar.
* Use include guards in .cmake file to avoid including the same file twice, like this:

if(NOT DEFINED MYFILE\_INCLUDED)

    set(MYFILE\_INCLUDED TRUE)

    # Your CMake code here

endif()

Including a file multiple times causes redefinition of functions or variables.

* If a file might not exist, use OPTIONAL to avoid errors:

include(myfile.cmake OPTIONAL)

* Watch out for path issues, especially with spaces—always quote paths like "${MY\_DIR}".
* Use if() statements to include files based on conditions, such as platform or compiler:

if(UNIX)

    include(unix-specific.cmake)

endif()

## Examples

#### Hello World (Minimal CMakeLists.txt File)

|  |  |
| --- | --- |
| hello  CMakeLists.txt  main.cpp  $ make  Scanning dependencies of target hello\_cmake  [100%] Building CXX object CMakeFiles/hello\_cmake.dir/hello\_cmake.cpp.o  Linking CXX executable hello\_cmake  [100%] Built target hello\_cmake | # Set the minimum version of CMake that can be used  # To find the cmake version run  # $ cmake --version  cmake\_minimum\_required(VERSION 3.10)  # Set the project name  project(hello\_cmake)  # Add an executable  add\_executable(hello\_cmake main.cpp) |

#### Hello World (With Header File)

|  |  |
| --- | --- |
| hello  │ CMakeLists.txt  ├───include  │ hello.h  └───src  hello.cpp  main.cpp | # Set the minimum version of CMake that can be used  cmake\_minimum\_required(VERSION 3.5)  # Set the project name  project (HelloWorldWithHeader)  # Create a sources variable with a link to all cpp files to compile  set(SOURCES      src/hello.cpp      src/main.cpp  )  # Add an executable with the above sources  add\_executable(helloworld ${SOURCES})  # Set the directories that should be included in the build command for this target  # when running g++ these will be included as -I/directory/path/  target\_include\_directories(helloworld      PRIVATE     ${PROJECT\_SOURCE\_DIR}/include  ) |

#### Compiler Flags

##### Options

|  |  |
| --- | --- |
| Input:  ├── CMakeLists.txt  ├── main.cpp | cmake\_minimum\_required(VERSION 3.10)  project(myproject)  # Set C++ compiler options  set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall -pedantic -Wextra")  # Add an executable target  add\_executable(myproject main.cpp) |

Important:

* NEVER write set(CMAKE\_CXX\_FLAGS "-Wall -pedantic -Wextra"). This completely overwrites the existing value of CMAKE\_CXX\_FLAGS. This can lead to loss of any flags that were previously set, whether by the user via command line or by CMake itself.

##### Definitions

|  |  |
| --- | --- |
| Input:  ├── CMakeLists.txt  ├── main.cpp  **////////////////////////// EX1 ////////////////////////////**  User commands:  $ cmake ..  $ cmake --build .  Program output:  Hello EX1!  **/////////////////////// EX1 and EX2 //////////////////////**  User commands:  $ cmake .. -DCMAKE\_CXX\_FLAGS="-DEX2"  $ cmake --build .  Program output:  Hello EX1!  Hello EX2! | CMakeLists.txt:  cmake\_minimum\_required(VERSION 3.5)  # Set the project name  project(myproject)  # Add an executable  add\_executable(myproject main.cpp)  # Set a compiler flag -DEX1  target\_compile\_definitions(myproject      PRIVATE EX1  )  main.cpp:  #include <iostream>  int main() {  #ifdef EX1    std::cout << "Hello EX1!" << std::endl;  #endif  #ifdef EX2    std::cout << "Hello EX2!" << std::endl;  #endif     return 0;  } |

#### Static Libraries

|  |  |
| --- | --- |
| Input:  static-lib  │ CMakeLists.txt  ├───include  │ └───static  │ hello.h  └───src  hello.cpp  main.cpp  Output:  shared-lib  build  └───libhello\_library.a  Makefile  hello\_binary  ...  $ cmake --build .  Scanning dependencies of target hello\_library  [ 50%] Building CXX object CMakeFiles/hello\_library.dir/src/hello.cpp.o  Linking CXX static library libhello\_library.a  [ 50%] Built target hello\_library  Scanning dependencies of target hello\_binary  [100%] Building CXX object CMakeFiles/hello\_binary.dir/src/main.cpp.o  Linking CXX executable hello\_binary  [100%] Built target hello\_binary | cmake\_minimum\_required(VERSION 3.5)  project(hello\_library)  ############################################################  # Create a library  ############################################################  # Generate the static library from the library sources  add\_library(hello\_library  STATIC      src/hello.cpp  )  target\_include\_directories(hello\_library      PUBLIC    ${PROJECT\_SOURCE\_DIR}/include  )  ############################################################  # Create an executable  ############################################################  # Add an executable with the above sources  add\_executable(hello\_binary      src/main.cpp  )  # Link the new hello\_library target with the hello\_binary target  target\_link\_libraries(hello\_binary      PRIVATE     hello\_library  ) |

#### Shared Libraries

|  |  |
| --- | --- |
| Input:  shared-lib  │ CMakeLists.txt  ├───include  │ └───shared  │ hello.h  └───src  hello.cpp  main.cpp  Output:  shared-lib  build  └───libhello\_library.so  Makefile  hello\_binary  ...  $ cmake --build .  Scanning dependencies of target hello\_library  [ 50%] Building CXX object CMakeFiles/hello\_library.dir/src/Hello.cpp.o  Linking CXX shared library libhello\_library.so  [ 50%] Built target hello\_library  Scanning dependencies of target hello\_binary  [100%] Building CXX object CMakeFiles/hello\_binary.dir/src/main.cpp.o  Linking CXX executable hello\_binary  [100%] Built target hello\_binary | cmake\_minimum\_required(VERSION 3.5)  project(hello\_library)  ############################################################  # Create a library  ############################################################  # Generate the shared library from the library sources  add\_library(hello\_library  SHARED      src/hello.cpp  )  # [OPTIONAL] Set an alternative name for the target.  # You can refer the target using the alias name when linking it against other targets.  add\_library(hello::library ALIAS hello\_library)  target\_include\_directories(hello\_library      PUBLIC      ${PROJECT\_SOURCE\_DIR}/include  )  ############################################################  # Create an executable  ############################################################  # Add an executable with the above sources  add\_executable(hello\_binary      src/main.cpp  )  # link the new hello\_library target with the hello\_binary target  target\_link\_libraries(hello\_binary      PRIVATE      hello::library  ) |

#### Third-Party Libraries

##### Supported by CMake’s Built-In Modules

|  |  |
| --- | --- |
| This example requires the boost libraries to be installed in a default system location.  Input:  ├── CMakeLists.txt  ├── main.cpp  CMake output:  $ cmake ..  -- The C compiler identification is GNU 4.8.4  -- The CXX compiler identification is GNU 4.8.4  -- Check for working C compiler: /usr/bin/cc  -- Check for working C compiler: /usr/bin/cc -- works  -- Detecting C compiler ABI info  -- Detecting C compiler ABI info - done  -- Check for working CXX compiler: /usr/bin/c++  -- Check for working CXX compiler: /usr/bin/c++ -- works  -- Detecting CXX compiler ABI info  -- Detecting CXX compiler ABI info - done  -- Boost version: 1.54.0  -- Found the following Boost libraries:  -- filesystem  -- system  boost found  -- Configuring done  -- Generating done  -- Build files have been written to: /home/matrim/workspace/cmake-examples/01-basic/H-third-party-library/build  $ cmake --build .  Scanning dependencies of target third\_party\_include  [100%] Building CXX object CMakeFiles/third\_party\_include.dir/main.cpp.o  Linking CXX executable third\_party\_include  [100%] Built target third\_party\_include | cmake\_minimum\_required(VERSION 3.5)  # Set the project name  project(third\_party\_include)  # Find a boost install with the libraries filesystem and system  find\_package(Boost 1.46.1  REQUIRED  COMPONENTS filesystem system  )  # Check if boost was found  # Boost\_FOUND is defined in FindBoost.cmake.  if(Boost\_FOUND)      message("boost found")      # Add an executable      add\_executable(third\_party\_include main.cpp)        # Link against the boost libraries      target\_link\_libraries(third\_party\_include          PRIVATE          Boost::filesystem      )  else()      message(FATAL\_ERROR "Cannot find Boost")  endif() |

##### Not Supported by CMake’s Built-In Modules

|  |  |
| --- | --- |
| This example requires you to write your own Find<PackageName>.cmake file.  Input:  ├── CMakeLists.txt  ├── main.cpp | cmake\_minimum\_required(VERSION 3.10)  project(MyProject)  # Add the custom path to CMAKE\_MODULE\_PATH  # allowing CMake to look for FindMyLibrary.cmake in that location  list(APPEND CMAKE\_MODULE\_PATH "${CMAKE\_SOURCE\_DIR}/cmake/modules")  # Now find the third-party library  find\_package(MyLibrary REQUIRED)  # Check if it was found  if (MyLibrary\_FOUND)      include\_directories(${MyLibrary\_INCLUDE\_DIRS})      add\_executable(MyExecutable main.cpp)      target\_link\_libraries(MyExecutable ${MyLibrary\_LIBRARIES})  else()      message(FATAL\_ERROR "MyLibrary not found!")  endif() |

#### Installation

##### To Default Directory

|  |  |
| --- | --- |
| Input:  myproject  │ cmake-examples.conf  │ CMakeLists.txt  ├───include  │ └───installing  │ hello.h  └───src  hello.cpp  main.cpp  $ cmake --build .  Scanning dependencies of target cmake\_examples\_inst  [ 50%] Building CXX object CMakeFiles/cmake\_examples\_inst.dir/src/hello.cpp.o  Linking CXX shared library libcmake\_examples\_inst.so  [ 50%] Built target cmake\_examples\_inst  Scanning dependencies of target cmake\_examples\_inst\_bin  [100%] Building CXX object CMakeFiles/cmake\_examples\_inst\_bin.dir/src/main.cpp.o  Linking CXX executable cmake\_examples\_inst\_bin  [100%] Built target cmake\_examples\_inst\_bin  $ sudo make install  [sudo] password for ...:  [ 50%] Built target cmake\_examples\_inst  [100%] Built target cmake\_examples\_inst\_bin  Install the project...  -- Install configuration: ""  -- Installing: /usr/local/bin/cmake\_examples\_inst\_bin  -- Removed runtime path from "/usr/local/bin/cmake\_examples\_inst\_bin"  -- Installing: /usr/local/lib/libcmake\_examples\_inst.so  -- Installing: /usr/local/etc/cmake-examples.conf  $ cat install\_manifest.txt  /usr/local/bin/cmake\_examples\_inst\_bin  /usr/local/lib/libcmake\_examples\_inst.so  /usr/local/etc/cmake-examples.conf  $ ls /usr/local/bin/  cmake\_examples\_inst\_bin  $ ls /usr/local/lib  libcmake\_examples\_inst.so  $ ls /usr/local/etc/  cmake-examples.conf  $ LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/lib cmake\_examples\_inst\_bin  ---- | cmake\_minimum\_required(VERSION 3.5)  project(cmake\_examples\_install)  ############################################################  # Create a library  ############################################################  # Generate the shared library from the library sources  add\_library(cmake\_examples\_inst  SHARED      src/hello.cpp  )  target\_include\_directories(cmake\_examples\_inst      PUBLIC     ${PROJECT\_SOURCE\_DIR}/include  )  ############################################################  # Create an executable  ############################################################  # Add an executable with the above sources  add\_executable(cmake\_examples\_inst\_bin      src/main.cpp  )  # link the new hello\_library target with the hello\_binary target  target\_link\_libraries(cmake\_examples\_inst\_bin      PRIVATE     cmake\_examples\_inst  )  ############################################################  # Install  ############################################################  # Binaries  install(TARGETS cmake\_examples\_inst\_bin      DESTINATION bin)  # Library  # Note: may not work on windows  install(TARGETS cmake\_examples\_inst LIBRARY  DESTINATION lib)  # Header files  install(DIRECTORY ${PROJECT\_SOURCE\_DIR}/include/      DESTINATION include)  # Config  install(FILES cmake-examples.conf      DESTINATION etc) |

##### To Custom Directory

|  |  |
| --- | --- |
| Input:  myproject/  ├── CMakeLists.txt  ├── src/  │   └── main.cpp  └── include/      └── myproject.h  Output:  myproject/  └── build/      └── install/          ├── bin/          │   └── MyExecutable          └── include/              └── myproject.h | cmake\_minimum\_required(VERSION 3.10)  project(MyProject VERSION 1.0 LANGUAGES CXX)  # Specify the installation directories  set(CMAKE\_INSTALL\_PREFIX ${CMAKE\_BINARY\_DIR}/install)  # Include directories  include\_directories(${CMAKE\_SOURCE\_DIR}/include)  # Source files  set(SOURCES      src/main.cpp  )  # Create an executable  add\_executable(MyExecutable ${SOURCES})  # Install the executable  install(TARGETS MyExecutable DESTINATION bin)  # Install header files  install(DIRECTORY include/ DESTINATION include) |

#### Sub-Projects (Multiple CMakeLists.txt Files)

|  |  |
| --- | --- |
| Input:  myproject/  ├── CMakeLists.txt  ├── app/  │   ├── CMakeLists.txt  │   └── main.cpp  ├── lib/  │   ├── CMakeLists.txt  │   ├── mylib.cpp  │   └── mylib.h  └── build/  Output:  myproject/  └── build/      ├── CMakeFiles/      │   ├── MyApp.dir/      │   │   ├── main.cpp.o      │   │   └── ...      │   ├── MyLibrary.dir/      │   │   ├── mylib.cpp.o      │   │   └── ...      │   └── CMakeOutput.log      ├── MyApp       # Executable      ├── MyLibrary.a      ├── CMakeCache.txt      └── Makefile | Top-level CmakeLists.txt:  cmake\_minimum\_required(VERSION 3.10)  project(MyProject VERSION 1.0 LANGUAGES CXX)  # Add the library subdirectory  add\_subdirectory(lib)  # Add the application subdirectory  add\_subdirectory(app)  lib/CMakeLists.txt:  add\_library(MyLibrary mylib.cpp)  # Specify the include directory for this library  target\_include\_directories(MyLibrary PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR})  app/CMakeLists.txt:  add\_executable(MyApp main.cpp)  # Link the library to the application  target\_link\_libraries(MyApp PRIVATE MyLibrary) |

#### Build Type

|  |  |
| --- | --- |
| Input:  myproject  CMakeLists.txt  main.cpp  **/////////////////////////// RELEASE //////////////////////////**  User commands:  $ cd build  $ cmake -DCMAKE\_BUILD\_TYPE=Release ..  $ cmake --build . --target run  CMake output:  Build type: Release  Compiler flag: NODEBUG\_MODE  Compiler option: -Os  D:/MYGIT/Tutorials/C-C++/Demo\_CMake/Github/cmake-examples/01-basic/F-build-type/build  -- Configuring done (0.2s)  -- Generating done (0.0s)  -- Build files have been written to: D:/MYGIT/Tutorials/C-C++/Demo\_CMake/Github/cmake-examples/01-basic/F-build-type/build  [ 50%] Building CXX object CMakeFiles/myexe.dir/main.cpp.obj  [100%] Linking CXX executable myexe.exe  [100%] Built target myexe    Program output:  Hello in no-debug mode!  **//////////////////////////// DEBUG /////////////////////////**  User commands:  # As you set "-DCMAKE\_BUILD\_TYPE=Release" from the above step, you need to clean the build directory to make it un-specified.  $ rm -rf build; mkdir build  $ cd build  $ cmake ..  $ cmake --build . --target run  CMake output:  Build type: Debug  Compiler flag: DEBUG\_MODE  Compiler option: -g  -- Configuring done (0.2s)  -- Generating done (0.1s)  -- Build files have been written to: D:/MYGIT/Tutorials/C-C++/Demo\_CMake/Github/cmake-examples/01-basic/F-build-type/build  [ 50%] Building CXX object CMakeFiles/myexe.dir/main.cpp.obj  [100%] Linking CXX executable myexe.exe  [100%] Built target myexe  Program output:  Hello in debug mode! | CMakeLists.txt:  cmake\_minimum\_required(VERSION 3.10)  project(MyProject)  add\_executable(myexe main.cpp)  # Set the build type to Debug if not specified  # You can specify the build type by adding "-DCMAKE\_BUILD\_TYPE=<build-type" in cmake cmd  if(NOT CMAKE\_BUILD\_TYPE)      set(CMAKE\_BUILD\_TYPE "Debug")  endif()  message("Build type: ${CMAKE\_BUILD\_TYPE}")  # Set compiler flags and options based on the build type  if(CMAKE\_BUILD\_TYPE STREQUAL "Debug")      target\_compile\_definitions(myexe PRIVATE DEBUG\_MODE)      target\_compile\_options(myexe PRIVATE -g)  # Enable debug symbols  else()    target\_compile\_definitions(myexe PRIVATE NODEBUG\_MODE)    target\_compile\_options(myexe PRIVATE -Os)  endif()  # Print the compiler flags  get\_target\_property(FLAGS myexe COMPILE\_DEFINITIONS)  foreach(option ${FLAGS})      message("Compiler flag: ${option}")  endforeach()  # Print the compiler options  get\_target\_property(OPTS myexe COMPILE\_OPTIONS)  foreach(option ${OPTS})      message("Compiler option: ${option}")  endforeach()  # Create a custom CMake target to run your executable directly from the build system.  # The benefit is that you don't need to locate your executable.  add\_custom\_target(run      COMMAND myexe      DEPENDS myexe      WORKING\_DIRECTORY ${CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY}  )  main.cpp:  #include <iostream>  int main() {  #ifdef DEBUG\_MODE     std::cout << "Hello in debug mode!" << std::endl;  #else     std::cout << "Hello in no-debug mode!" << std::endl;  #endif     return 0;  } |

Notes:

* As you see, adding -g manually is not necessary. Instead, using a build type, either one of the 4 built-in build types or a custom one, is a better practice.

#### Chossing Compiler

CMake by default chooses the default C/C++ compiler installed in your system. But you can externally select a different one via CMAKE\_CXX\_COMPILER and CMAKE\_C\_COMPILER built-in variables.

In this example, we will use Clang compiler.

|  |  |
| --- | --- |
| CMake commands:  $ cd build  $ cmake .. -DCMAKE\_C\_COMPILER=clang-3.6 -DCMAKE\_CXX\_COMPILER=clang++-3.6  $ cmake --build . | cmake\_minimum\_required(VERSION 3.5)  project (hello\_cmake)  add\_executable(hello\_cmake main.cpp) |

#### Visibility of Target Properties

##### Private

|  |  |
| --- | --- |
| **Input:**  │   CMakeLists.txt  ├───include  │       util.h  └───src          main.cpp          util.cpp  **main.cpp:**  #include <iostream>  #include "util.h"  int main() {      std::string message = "Hello, World!";      std::cout << to\_upper(message) << std::endl;      return 0;  }  **util.h:**  #pragma once  #include <string>  std::string to\_upper(const std::string &input);  **util.cpp:**  #include "util.h"  #include <algorithm>  std::string to\_upper(const std::string &input) {      std::string result = input;      std::transform(result.begin(), result.end(), result.begin(), ::toupper);      return result;  } | cmake\_minimum\_required(VERSION 3.10)  project(MyApp)  ######## Lib #########  # Create a static library  add\_library(util STATIC src/util.cpp)  # Specify include directories for the library  target\_include\_directories(util PRIVATE include)  ######## Exe #########  # Create an executable that uses the library  add\_executable(MyApp src/main.cpp)  # Link the library to the executable  target\_link\_libraries(MyApp PRIVATE util)  # Specify include directories for the executable  target\_include\_directories(MyApp PRIVATE include) |

##### Public

|  |  |
| --- | --- |
| Similar to PRIVATE | cmake\_minimum\_required(VERSION 3.10)  project(MyApp)  ######## Lib #########  # Create a static library  add\_library(util STATIC src/util.cpp)  # Specify include directories for the library  target\_include\_directories(util PUBLIC include)  ######## Exe #########  # Create an executable that uses the library  add\_executable(MyApp src/main.cpp)  # Link the library to the executable  target\_link\_libraries(MyApp PRIVATE util) |

##### Interface

|  |  |
| --- | --- |
| Similar to PRIVATE  Except that:  **util.h:**  #pragma once  #include <string>  #include <algorithm>  inline std::string to\_upper(const std::string &input) {      std::string result = input;      std::transform(result.begin(), result.end(), result.begin(), ::toupper);      return result;  }  **Note**: For header-only library, you need to ensure that the function definitions are included in the header file itself. So, **both the declarations and definitions are typically placed in the same file**. | cmake\_minimum\_required(VERSION 3.10)  project(MyApp)  ######## Lib #########  # Create a header-only library target  add\_library(util INTERFACE)  # Specify include directories for the header-only library  target\_include\_directories(util INTERFACE include)  # ######## Exe #########  # Create an executable that uses the library  add\_executable(MyApp src/main.cpp)  # Link the library to the executable  target\_link\_libraries(MyApp PRIVATE util) |

#### Using CMake File

|  |  |
| --- | --- |
| │ CMakeLists.txt  ├───cmake  │ common.cmake  ├───include  │ calculator.h  └───src  calculator.cpp  main.cpp  **main.cpp:**  #include <iostream>  #include "calculator.h"  int main() {      int a = 10, b = 5;      std::cout << "Add: " << Calculator::add(a, b) << std::endl;      std::cout << "Sub: " << Calculator::subtract(a, b) << std::endl;      std::cout << "Mul: " << Calculator::multiply(a, b) << std::endl;      std::cout << "Div: " << Calculator::divide(a, b) << std::endl;      return 0;  }  **calculator.h:**  #pragma once  class Calculator {  public:      static int add(int a, int b);      static int subtract(int a, int b);      static int multiply(int a, int b);      static double divide(int a, int b);  };  **calculator.cpp:**  #include "calculator.h"  #include <stdexcept>  int Calculator::add(int a, int b) {      return a + b;  }  int Calculator::subtract(int a, int b) {      return a - b;  }  int Calculator::multiply(int a, int b) {      return a \* b;  }  double Calculator::divide(int a, int b) {      if (b == 0) {          throw std::invalid\_argument("Division by zero");      }      return static\_cast<double>(a) / b;  } | **CMakeLists.txt:**  cmake\_minimum\_required(VERSION 3.10)  project(SimpleCalculator)  # Add option for verbose output  # Note that options will be added to CMakeCache.txt,  #  so you'll need to "rm CMakeCache.txt" before "cmake .." to switch between ON/OFF value.  option(ENABLE\_VERBOSE "Enable verbose output during build" ON)  # Include the common file  include(cmake/common.cmake)  # Then you can call functions from common.cmake  # If you move these calls to above the include(), you'll get error "Unknown CMake command xxx."  set\_common\_flags()  add\_common\_includes()  enable\_verbose()  # Create a library for the calculator  add\_library(calculator src/calculator.cpp)  # Create an executable  add\_executable(SimpleCalculator src/main.cpp)  # Link the calculator library to the executable  target\_link\_libraries(SimpleCalculator PRIVATE calculator)  **common.cmake:**  # Function to add common compiler flags  function(set\_common\_flags)      message(STATUS "Setting common compiler flags...")      if(CMAKE\_CXX\_COMPILER\_ID STREQUAL "GNU")          set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall -Wextra -pedantic")      elseif(CMAKE\_CXX\_COMPILER\_ID STREQUAL "MSVC")          set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} /W4")      endif()  endfunction()  # Function to set include directories  function(add\_common\_includes)      include\_directories(include)      message(STATUS "Included directories: include")  endfunction()  # Function to enable verbose output if required  function(enable\_verbose)      if(ENABLE\_VERBOSE)          message(STATUS "Verbose output is enabled.")          set(CMAKE\_VERBOSE\_MAKEFILE ON)      else()          message(STATUS "Verbose output is disabled.")      endif()  endfunction() |

## Good Practices

### General

#### Treat CMake code like production code

CMake is code. Therefore, it should be clean. Use the same principles for CMakeLists.txt and modules as for the rest of the codebase.

#### Don’t abuse usage requirements

As an example, don’t add -Wall to the PUBLIC or INTERFACE section of target\_compile\_options, since it is not required to build depending targets.

### Targets and Properties

#### Forget the commands *add\_compile\_options*, *include\_directories*, *link\_directories*, *link\_libraries*

Those commands operate on the **directory level**. All targets defined on that level inherit those properties. This **increases the chance of hidden dependencies**. Better operate on the targets directly.

Also, by defining properties (i.e., compile definitions, compile options, compile features, include directories, and library dependencies) in terms of targets, it helps the developer to reason about the system at the target level. The developer does not need to understand the whole system in order to reason about a single target. The build system handles transitivity.

|  |  |
| --- | --- |
| DO | DON'T |
| target\_compile\_definitions  target\_compile\_features  target\_compile\_options  target\_include\_directories  target\_link\_libraries  target\_sources  get\_target\_property  set\_target\_property | add\_compile\_definitions  set\_compile\_features  add\_compile\_options  include\_directories  link\_directories |

#### Don’t use *target\_link\_libraries()* without specifying PUBLIC, PRIVATE or INTERFACE

#### Keep internal properties PRIVATE

If a target needs properties internally, add them to the PRIVATE section of the target\_\* commands.

### Modules

#### Use a *find* module for third-party libraries that do not support clients to use CMake

CMake provides a collection of *find* modules for third-party libraries. For example, Boost doesn't support CMake. Instead, CMake provides a find module to use Boost in CMake.

### Projects

#### Don't use *file(GLOB)* in projects.

**CMake is a** **build system generator, not a build system**. It evaluates the GLOB expression to a list of files when generating the build system. The build system then operates on this list of files. Therefore, the build system cannot detect that something changed in the file system.

CMake cannot just forward the GLOB expression to the build system, so that the expression is evaluated when building. CMake wants to be the common denominator of the supported build systems. **Not all build systems support this**, so CMake cannot support it neither.

### Cross Compiling

#### Use toolchain files for cross compiling

Toolchain files encapsulate toolchains for cross compilation.

#### Keep toolchain files simple

It’s easier to understand and simpler to use. Don’t put logic in toolchain files. Create a single toolchain file per platform.

## References

CMake samples:

* <https://github.com/kuanghl/cmake-samples>
* <https://cmake.org/cmake/help/latest/guide/tutorial/index.html>

Cookbooks:

<https://github.com/dev-cafe/cmake-cookbook>

Best practices:

<https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1>

# Autotools

## Overview

### Basic Steps

The basic steps to build an autotools based software component are:

1. Configuration

$ ./configure

Here we'll look at the available build environment, verify required dependencies, generate Makefiles and a config.h

1. Compilation

$ make

Here we actually build the software component, using the generated Makefiles.

1. Installation

$ make install

Here we install what has been built.

### System Types

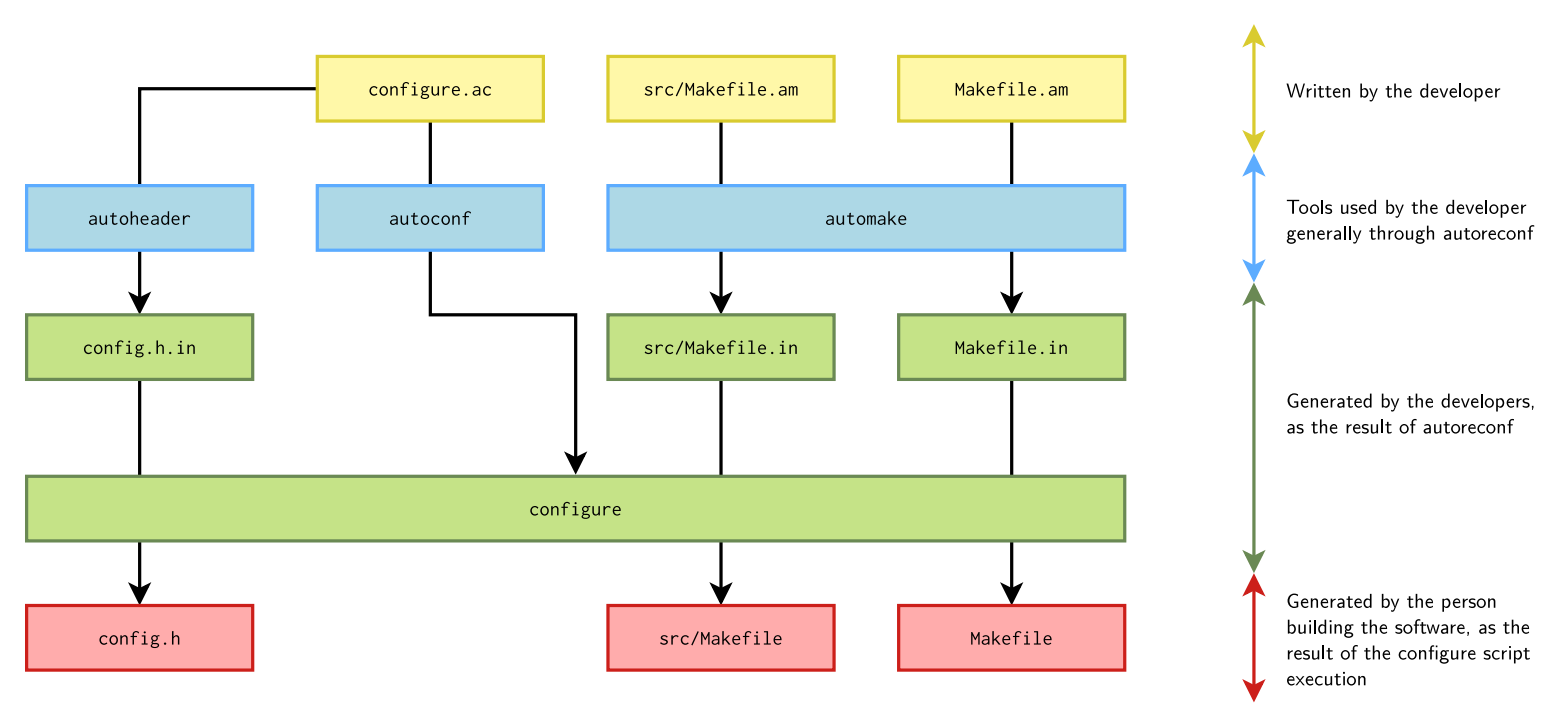
The autotools identify three system types:

* **build**: the system where the build takes place.
* **host**: the system where the execution of the compiled code will take place.
* **target**: the system for which the program will generate code. This is only used for compilers, assemblers, linkers, etc.

Corresponding configure options for these system types are:

* --build: generally does not need to be changed
* --host: must be overridden to do cross-compilation
* --target: needs to be overridden if needed (to generate a cross-compiler, for example)

Arguments to these options are configuration names, also called system tuples.



## Tool Set

### autoconf

Generate the configure script, which is a shell script, from the configure.ac.

This shell script is augmented with special constructs, called M4sh or m4 macros, for portability:

|  |  |  |  |
| --- | --- | --- | --- |
| **Macro** | **Description** | **Syntax** | **Note** |
| AC\_INIT | Process any command-line arguments.  Perform **initialization** and verification. | AC\_INIT (  package,  version,  [bug-report],  [tarname], [url]) | Every configure script must call AC\_INIT before doing anything else that produces output. |
| AC\_OUTPUT | Generates and runs config.status, which in turn creates the makefiles and any other files resulting from configuration. |  | Every configure.ac, should finish by calling AC\_OUTPUT. |
| AM\_INIT\_AUTOMAKE | Enable *automake* usage. | AM\_INIT\_AUTOMAKE([OPTIONS])  Common options:   * foreign: not require all the GNU Coding Style files such as NEWS, README, AUTHORS, etc. * dist-bzip2, dist-xz, etc. choose tarball format should be generated by make dist. * subdir-objects: tells that the objects are placed into the subdirectory of the build directory corresponding to the subdirectory of the source file. * version, e.g 1.14.1: tells the minimal *automake* version that is expected. |  |
| AC\_CONFIG\_HEADERS | Enable *autoheader* usage. | AC\_CONFIG\_HEADERS([config.h]) | A template C header file can be automatically generated by *autoheader*,  generally named config.h.in.  The final header file is generated by *configure*, generally named config.h. |
| AS\_IF | Replace for the shell if ... then .. fi |  |  |
| AS\_CASE | Replace for the shell case ... esac |  |  |
| AC\_PREREQ | Verifies that a recent enough **version of autoconf** is used. | AC\_PREREQ([2.68]) |  |
| AC\_CONFIG\_SRCDIR | Check that the source file is really where it should be by giving the path to the file in your project. | AC\_CONFIG\_SRCDIR([hello.c]) |  |
| AC\_CONFIG\_AUX\_DIR | Put the auxiliary build tools it requires in a different directory, rather than the one of onfigure.ac |  | Useful to keep cleaner build directory. |
| AC\_PROG\_CC | Make sure a C compiler is available. |  |  |
| AC\_PROG\_CXX | Make sure a C++ compiler is available. |  |  |
| AC\_PROG\_AWK |  |  |  |
| AC\_PROG\_GREP |  |  |  |
| AC\_PROG\_LEX |  |  |  |
| AC\_PROG\_YACC |  |  |  |
| AC\_CONFIG\_FILES | Generate Makefile from Makefile.in. | AC\_CONFIG\_FILES (  file...,  [cmds],  [init-cmds]) | *automake* will generate Makefile.in, and configure will generate the final Makefile. |
| AC\_DEFINE | Allows to create C definitions in the *configuration header*. | AC\_DEFINE (  variable,  value,  description) |  |
| AC\_FUNC\_FORK, AC\_FUNC\_GETLOADAVG, AC\_FUNC\_MALLOC |  | AC\_CHECK\_FUNC (  function,  [action-if-found],  [action-if-not-found]) |  |
| AC\_CHECK\_FUNC[S] | Check for generic functions. | C\_CHECK\_FUNCS (  function...,  [action-if-found],  [action-if-not-found]) |  |
| AC\_HEADER\_\* |  | AC\_CHECK\_HEADER (  header-file,  [action-if-found],  [action-if-not-found],  [includes]) |  |
| AC\_CHECK\_HEADER[S] |  | AC\_CHECK\_HEADERS (  header-file...,  [action-if-found],  [action-if-not-found],  [includes]) |  |
| AC\_SEARCH\_LIBS | Search for a library defining *function*, by linking a simple program calling f*unction*. | AC\_SEARCH\_LIBS (  function,  search-libs,  [action-if-found],  [action-if-not-found],  [other-libraries])   * Tries first with no library, and then with the different libraries in search-libs, one after the other. * If a library is found, -llibrary is prepended to the LIBS variable, so programs will be linked against it. The action-if-found is executed. * If not, action-if-not-found is executed. * other-libraries allows to pass additional -l<foo> arguments that may be needed for the link test to succeed.   Result in ac\_cv\_search\_<function> |  |
| AC\_CHECK\_PROGS |  | AC\_CHECK\_PROGS(PERL, [perl5 perl]) |  |
| AC\_CHECK\_DECLS |  |  |  |
| AC\_CHECK\_MEMBERS |  |  |  |
| AC\_CHECK\_TYPES |  | AC\_CHECK\_TYPES(int8\_t) |  |

**Notes:**

* The configure.ac used to be named configure.in but this name is now deprecated.
* It's possible to include normal shell constructs in configure.ac.
* Beware to not use *bashisms*: use only POSIX compatible constructs
* Most configure scripts use directly shell constructs, but AS\_ECHO, AS\_IF, etc. are available.

**Examples:**

* AC\_CHECK\_HEADERS example:

In configure.ac:

[...]

AC\_CHECK\_HEADERS([spawn.h],

[echo "Header spawn.h was found"; has\_spawn=yes],

[echo "Header spawn.h was not found"])

echo ${has\_spawn}

[...]

Execution of ./configure:

$ ./configure

[...]

checking for spawn.h... yes

Header spawn.h was found

yes

[...]

### automake

Generate the Makefile.in file from the Makefile.am file.

About Makefile.am:

* It’s really just a Makefile, but augmented with specific *automake* constructs that are expanded into regular *make* code.
* You can include regular *make* code, but for most situations, the *automake* constructs are sufficient to express what needs to be built.

**Syntax:**

A Makefile.am is composed of:

* **Product list variables**:
  + [modifier-list]prefix\_PRIMARY = product1 product2 ...
* prefix is the installation prefix, i.e. where it should be installed
  + All \*dir variables from autoconf can be used, without their dir suffix. E.g.: bindir, libdir, includedir, datadir, etc.
* PRIMARY describes what type of things should be built:
  + PROGRAMS, for executables
  + LIBRARIES, LTLIBRARIES, for libraries
  + HEADERS, for publicly installed header files
  + DATA, arbitrary data files
  + PYTHON, JAVA, SCRIPTS
  + MANS, TEXINFOS, for documentation
* After the = sign, list of products to be generated.
* Example:

bin\_PROGRAMS = hello

* **Product source variables**:
  + [modifier-list]product\_SOURCES = file1 file2 ...
  + product is the normalized name of the product, as listed in a *product list variable*.
    - The normalization consists in replacing special characters such as . or + by \_. For example, libfoo+.a in a *product list variable* gives the libfoo\_ \_a\_SOURCES *product source variable*.
    - \_SOURCES is always used, it’s not like a configurable *primary*.
    - Contains the list of files containing the source code for the product to be built.
    - Both source files and header files should be listed.
  + Example:

hello\_SOURCES = main.c

### autoheader

Generate the configuration header template config.h.in. Then, the final header file will be generated by *configure*, generally named config.h.

Example config.h:

/\* Define if the complete vga libraries (vga, vgagl) are installed \*/

/\* #undef HAVE\_LIBVGA \*/

/\* Define to 1 if you have the <limits.h> header file. \*/

#define HAVE\_LIMITS\_H 1

### aclocal

This is the smallest component in the Autotools suite, but it’s very important. You learned in the previous section that **autoconf** uses **m4** macros to be configured. But where do these **m4** macros come from? They’re generated by running the **aclocal** command. Simple as that. If you don’t run **aclocal** before running **autoconf**, you’ll get an error complaining about missing macros.

### autoreconf

To generate all the files used by autotools, you could call automake, autoconf, aclocal, autoheader, etc. manually. But it's not efficient.

A tool called **autoreconf automates this process**. So always use autoreconf.

Useful option: -i or --install, to ask autoreconf to copy missing auxiliary files

## Tips

* Generated files (configure, Makefile.in, Makefile) should not be modified. Reading them is also very difficult. Read the real source instead.

## Examples

Let’s build a C program:

#include <stdio.h>

int main(int argc, char\* argv[]) {

  printf("Hello World\n");

  return 0;

}

Configure.ac:

# App name, version, writer

AC\_INIT([helloworld], [0.1], [maintainer@example.com])

# Initialize automake

AM\_INIT\_AUTOMAKE

# Specify compiler (C compiler)

AC\_PROG\_CC

# Find a file called Makefile.in and replace placeholders according to what specified

AC\_CONFIG\_FILES([Makefile])

# Output the final configure script

AC\_OUTPUT

Makefile.am:

AUTOMAKE\_OPTIONS = foreign

bin\_PROGRAMS = hello

hello\_SOURCES = main.c

Commands:

aclocal

autoconf

automake --add-missing

make dist

Other docs:

[Introduction to GNU Autotools | Opensource.com](https://opensource.com/article/19/7/introduction-gnu-autotools)

[ADA553215.pdf (dtic.mil)](https://apps.dtic.mil/sti/pdfs/ADA553215.pdf)

[The configure.ac File | BuildTools (coin-or-tools.github.io)](https://coin-or-tools.github.io/BuildTools/configure.html)

# Cross Compiler

<http://eslinuxprogramming.blogspot.com/2015/04/cross-compiler.html>

<https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/>