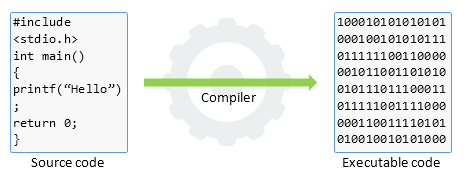
# Compilation Process

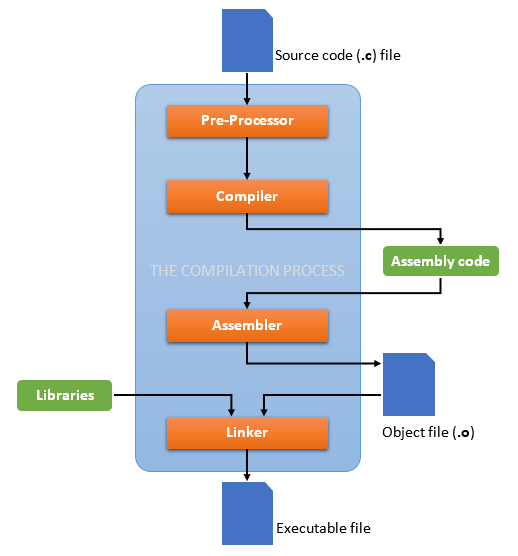
The process of translating source code written in high-level language to low-level machine code is called *compilation*. The compilation is done by a special software known as [compiler](https://codeforwin.org/2017/05/compiler-and-its-need.html). The compiler checks source code for any syntactical or 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.

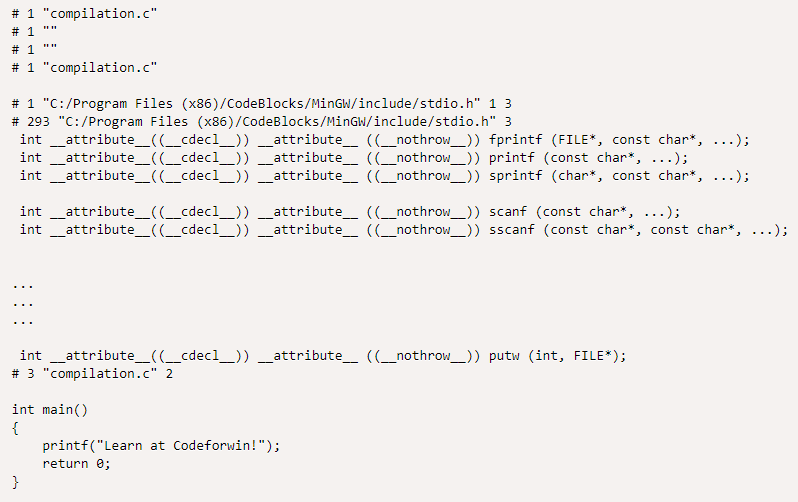
## Pre-Processing

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.

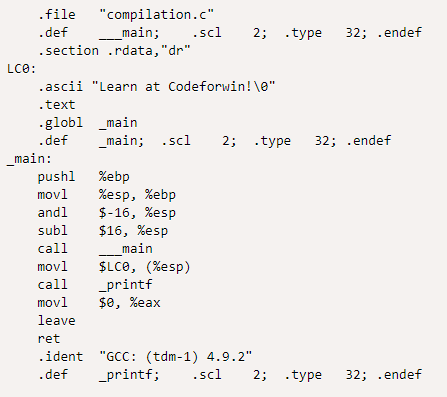
## Compilation

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.

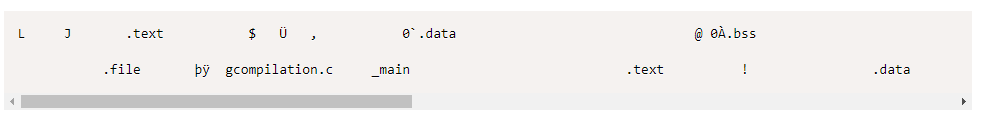
Let us look into compilation.s file:



## 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.

# 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

**Step 1: Generating build system**

# 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 ..

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 with cmake\_minimum\_required()
    - Sets **project name** and **version** with project()
    - **Finds dependencies**, if the project relies on external libraries or packages, with find\_package()
    - **Define targets**, like executables and libraries, with add\_executable() or add\_library()
    - **Set compliler flags** via, for example, set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall"
  + Generates a native build system:
    - The native build system can be Makefiles or Visual Studio project file 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.
* In the compilation and linking step, CMake will:
  + Checks for included header files.
  + Checks for static and dynamic libraries.
  + Check for dependencies.
  + Compiles the source code files into object files.
  + Links the object files and any required libraries to create the final executable or library file.
  + Links all the necessary components that a program or library requires in order to compile and run correctly.
  + The final output (executables or libraries) is placed in the specified output directory.

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**.

Alternatives:

# 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.

**Step 2: Building source code**

# In the build directory, call the build system to actually compile and link the project

$ cmake --build .

Or

# In the source directory

$ cmake --build build

**Step 3: Install …**

**cmake --install build**

## CMakeList File

### Built-In Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Usage** | **Note** | **Example** |
| **General** | | | | |
| set(  <var\_name> <value>  ) | Set value for variables |  | set(  HEADER\_PATH project/src/include  ) |
| message() |  |  |  |
| list() |  |  |  |
| **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 | Optional. Mostly not used. | configure\_file(  TutorialConfig.h.in  TutorialConfig.h  ) |
|  |  |  |  |
| **Building** | | | | |
| 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>  <src1> <src2> ...  ) | Specify library name and which source files should make up the library | Used when you want to build static/dynamic libraries | add\_library(MathFunctions STATIC  MathFuncs.cxx mysqrt.cxx  ) |
| add\_subdirectory(  <directory>  [binary\_dir]  ) | Add the subdirectory to the build.  Used when source files are located in multiple sub-directories. | This requires to create multiple CMakeList files for each sub-directory. | add\_subdirectory(  MathFuncs) |
| target\_include\_directories(  <target\_name>  PRIVATE|PUBLIC|INTERFACE  <dir1> <dir2> ...  ) | Specify where the given executable target should look for include files | This is the preferred method over the include\_directories. | target\_include\_directories(  Tutorial PUBLIC  "${CMAKE\_CURRENT\_SOURCE\_DIR}}"  ) |
| include\_directories(  dir1> <dir2> ...  ) | Specify where all the executable targets should look for include files |  | include\_directories(  "${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\_compile\_options(  <target>  [<INTERFACE|PUBLIC|PRIVATE>]  <options>...  ) | Specify compiler options for a specific target |  | target\_compile\_options(  MyExecutable  PRIVATE  -Wall -O2) |
| **Installation** | | | | |
| install() |  |  |  |
|  |  |  |  |

### Built-In Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Usage** | **Note** | **Example** |
| **C/C++ standard** | | | |
| CMAKE\_CXX\_COMPILER | C++ compiler being used for the project. |  |  |
| CMAKE\_C\_COMPILER | C compiler being used for the project. |  |  |
| CMAKE\_CXX\_STANDARD | C++ standard being used for the project. |  | set(CMAKE\_CXX\_STANDARD 11) |
| CMAKE\_CXX\_STANDARD\_REQUIRED | 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) |
| **Paths**  myproject/  ├── CMakeLists.txt  ├── src/  │   ├── CMakeLists.txt  │   ├── main.cpp  │   └── utils.cpp  ├── include/  │   └── utils.h  ├── build/  │   └── src/  └── tests/      ├── CMakeLists.txt      └── test\_main.cpp | | | |
| CMAKE\_SOURCE\_DIR | Top-level source directory of the project. |  | myproject |
| CMAKE\_BINARY\_DIR | Top-level binary directory where build files are generated. |  | myproject/build |
| CMAKE\_CURRENT\_SOURCE\_DIR | Source directory of the currently processed CMakeLists.txt. |  | For src/CMakeLists.txt: myproject/src  For tests/CMakeLists.txt: myproject/tests |
| CMAKE\_CURRENT\_BINARY\_DIR | Binary directory for the currently processed CMakeLists.txt. |  | For src/CMakeLists.txt: myproject/build/src  For tests/CMakeLists.txt: myproject/build/tests |
| CMAKE\_MODULE\_PATH | A list of directories to search for additional CMake modules. | Useful for finding custom modules. |  |
| **Others** | | | |
| CMAKE\_PROJECT\_NAME | Project name as defined by the project() function. | Useful for referencing the project name. |  |
| CMAKE\_BUILD\_TYPE | Build type (e.g., Debug, Release). | Can affect compiler flags and options. | if(CMAKE\_BUILD\_TYPE STREQUAL "Debug")  ...  endif() |
| CMAKE\_INSTALL\_PREFIX | Default installation directory for the project.  By default, it is set to /usr/local on Unix-like systems and C:\Program Files on Windows. | Can be overridden at configure time. | Check example section |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Examples

#### Minimal CMakeLists.txt File

cmake\_minimum\_required(VERSION 3.10)

project(MyProject VERSION 1.0 LANGUAGES CXX)

add\_executable(MyExecutable main.cpp)

#### Changnig Installation Directory

|  |  |
| --- | --- |
| Input:  myproject/  ├── CMakeLists.txt  ├── src/  │   └── main.cpp  └── include/      └── myproject.h  Output:  myproject/  └── build/      └── install/          ├── bin/          │   └── MyExecutable          └── include/              └── myproject.h  User input:  cmake -B build -S .  cmake --build build  cmake --install build | 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) |

#### Adding Subdirectories

|  |  |
| --- | --- |
| 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  User input:  cmake -B build -S .  cmake --build build | 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) |

# 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

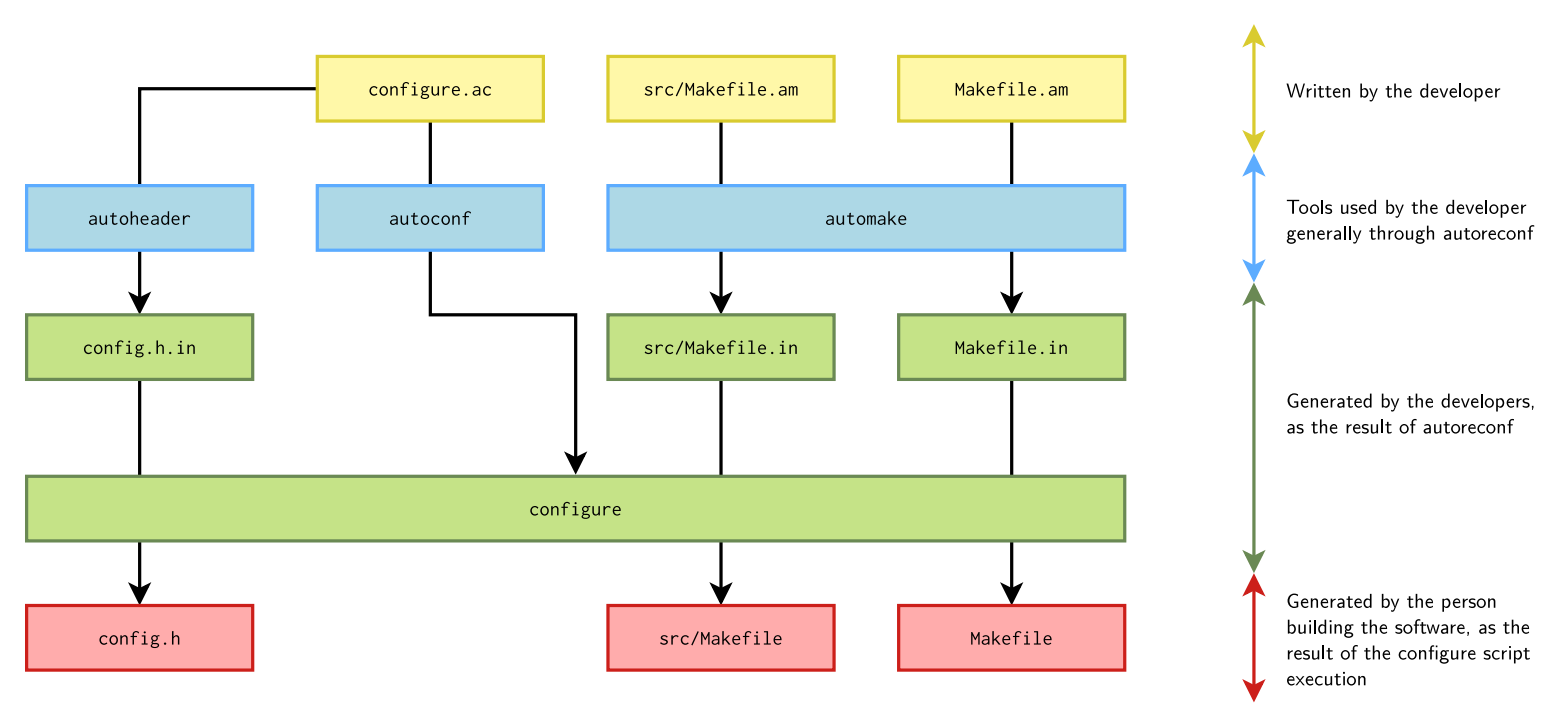
The autotools identify three system types:

* **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/>