C++ Software Engineering

for engineers of other disciplines

Module 5

"C++ Build"

1st Lecture: **g++**

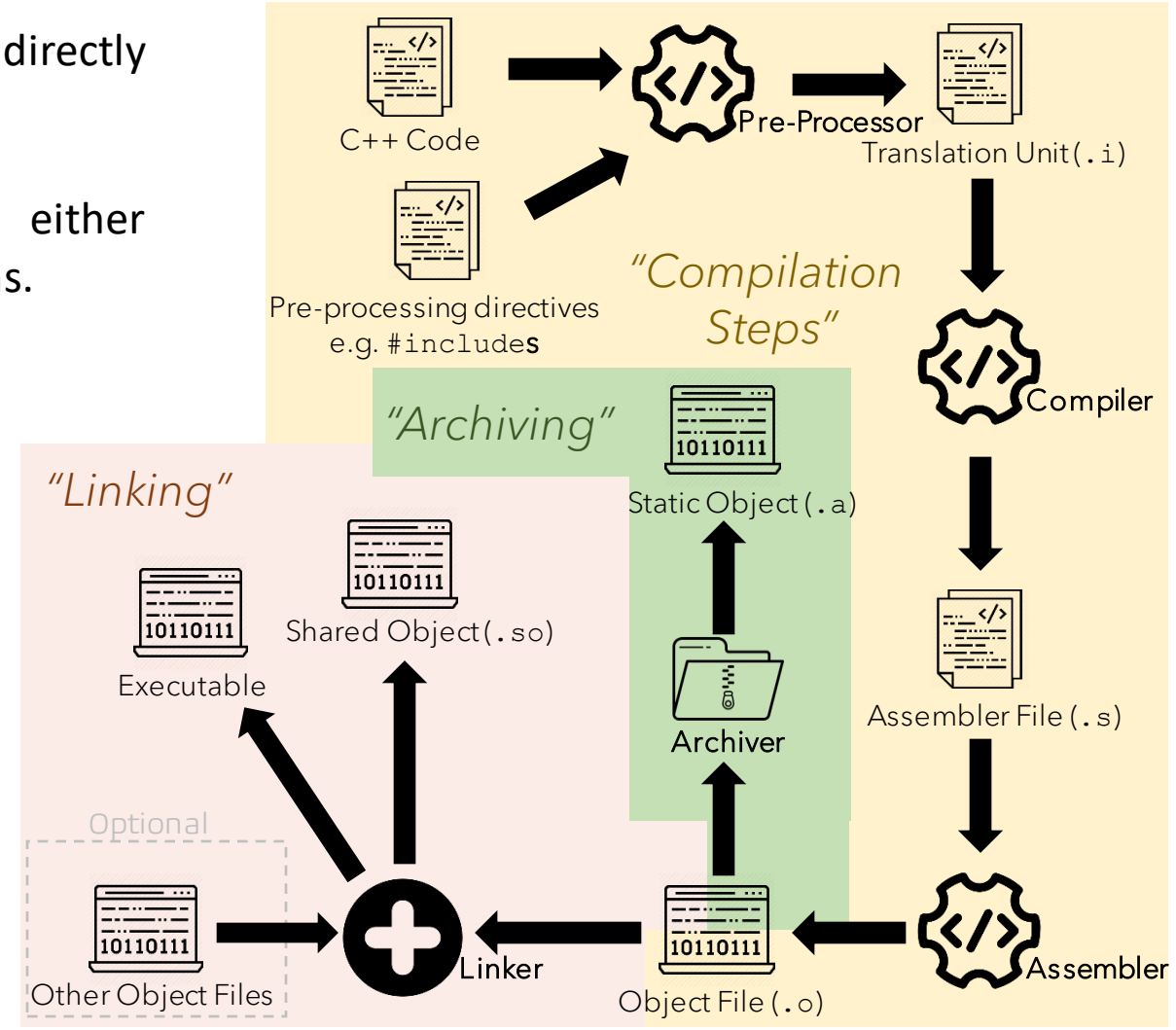
ALTEN

Spring 2022

Gothenburg, Sweden

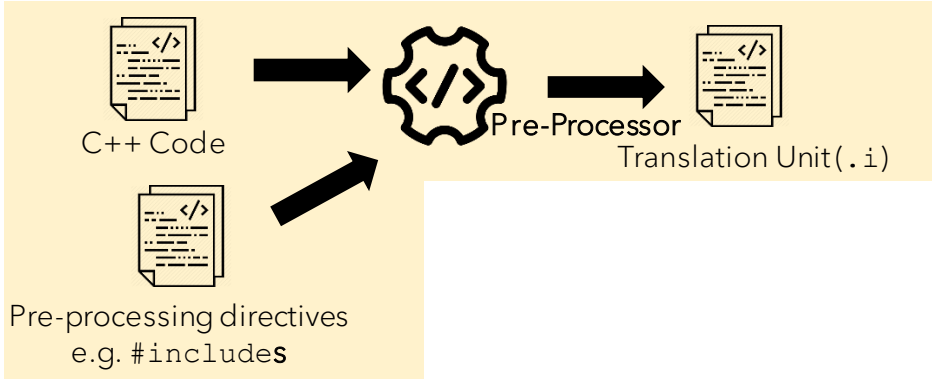
C++ into Software

- C++ is a compiled language i.e. its source code directly *compiles* into machine code using ***compilers.***
- **g++** can preform *most* of the necessary steps, either directly or indirectly i.e. by invoking other applications.

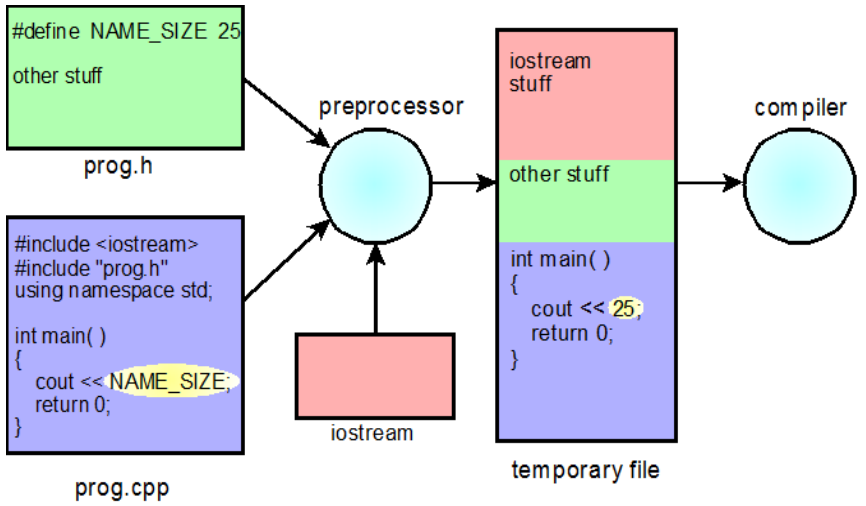


Pre-processor – g++

- All *paths* to the files ***included*** should be *visible* to "*Pre-Processor*".
- **g++** starts looking for the ***included*** header files from within the same directory of their *including* source. If the file is not found, it will then look into the *default include paths* a.k.a. *system header file directory.*



- The output of pre-processing i.e. translation unit, could be viewed using flag **-E**. Since system libraries across different platform and operation system could vary, the output of this step *could be* platform and OS dependent.

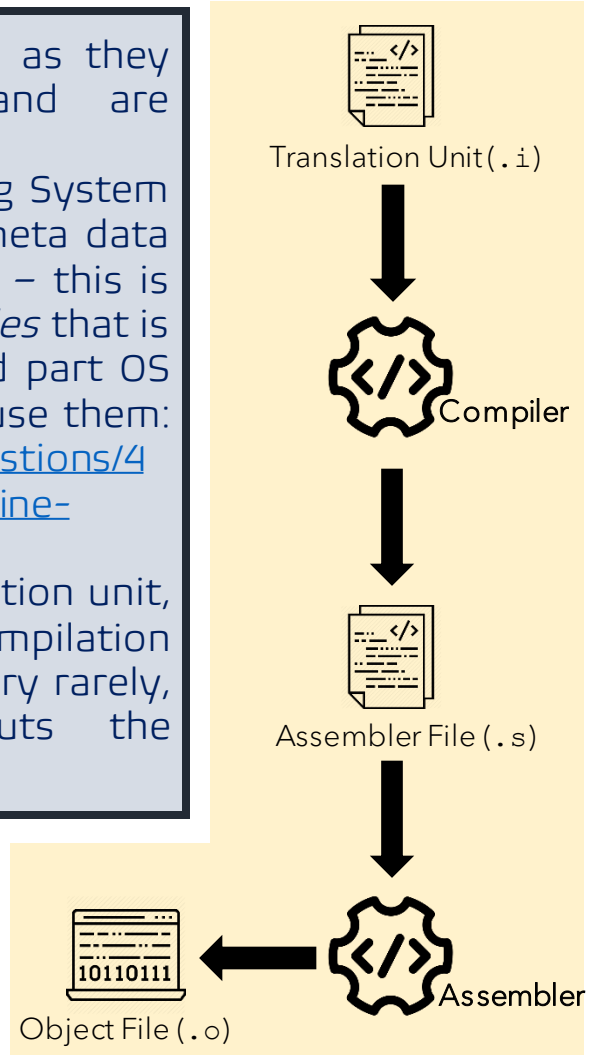


http://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/compiler_op.html

Compiler & Assembler – g++

- Compiler *converts* the translation unit into assembly code and assembler creates ***object file*** from the *assembler file*.
- **`g++ -c`** outputs the object file i.e. performs compilation step.
- C++ compilers perform ***name mangling***.

- Object files are not portable as they contain *machine codes* and are hardware dependent.
- Object files are also Operating System dependent, as they contain meta data which could vary in every OS – this is similar to general *machine codes* that is part *machine instructions* and part OS related metadata on how to use them: https://stackoverflow.com/questions/41153978/why-does-the-machine-code-depend-on-the-os-type
- The assembler file, like translation unit, are temporary files in compilation toolchain and are modified very rarely, almost never. **`g++`** outputs the assembler file using **`-S`** flag.



"An object file is a computer file containing object code, that is, machine code output of an assembler or compiler […] and not usually directly executable. There are various formats for object files, and the same machine code can be packaged in different object file formats […] In addition to the object code itself, object files may contain metadata."

https://en.wikipedia.org/wiki/Object_file

Name Mangling – g++

- A technique employed by C++ compilers to solve issues related to *identifiers' naming uniqueness.*
- A method to pass more semantic to the ***linker*** by encoding *additional information* into the names of functions, structures, classes, and other types (when/if necessary).
- C compilers do not *mangle* names, in order to link C++ Object files with C object files, keyword **`extern`** should be used to notify the compiler to skip name mangling for a given .

- Name mangling performed by C++ compilers allows function overloading, which is not permitted in C.
- Object files could use symbols which their definitions are not presented in the same object file, these symbols are *Undefined* (**U**) and need *external linkage* – the object file including *Undefined* symbols shall be *linked* to appropriate "*external*" object files which includes those symbols.

C++ Source Code

```
extern "C" void doNothingWithInt(int){}
extern "C" {
  void doNothingWithChar(char){}
  void doNothingWithFloat(float);
}
namespace nothing {
  void doNothing(int){}
}
void doNothing(int){}
void doNothing(char){}
void doNothing(float){}
```

Object File (`.o`)

```
000000000000000e T doNothingWithChar
                 U doNothingWithFloat
0000000000000000 T doNothingWithInt
000000000000002e T _ZN7nothing9doNothingEi
000000000000004a T _Z9doNothingc
000000000000005a T _Z9doNothingf
000000000000003c T _Z9doNothingi
```

Memory Address | Type | Symbol Name

"A symbol in computer programming is a primitive data type whose instances have a unique human-readable form. Symbols can be used as identifiers." https://en.wikipedia.org/wiki/Symbol_(programming)

Extern

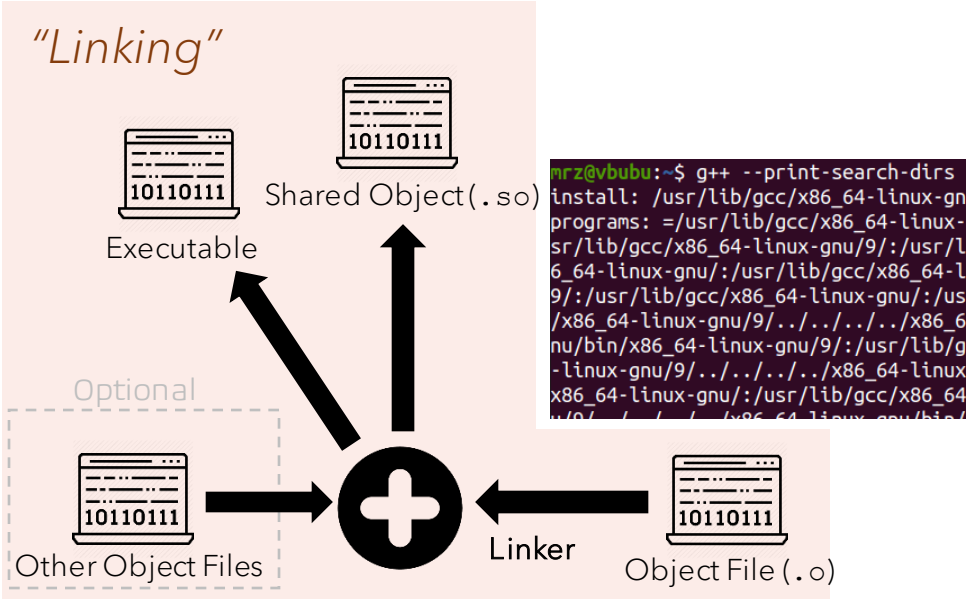

```cpp
extern "C" void hw_fromC();
int main() {
    hw_fromC();
    return 0;
}
```

```cpp
#include <stdio.h>
void hw_fromC() {
   printf("Hello, World!\n");
}
```

```
mrz@vbubu:~/cc/CXX_Course_Demo/Day7/2_extern$ nm cpp_e.o
                 U _GLOBAL_OFFSET_TABLE_
                 U hw_fromC
0000000000000000 T main
mrz@vbubu:~/cc/CXX_Course_Demo/Day7/2_extern$ nm c_e.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T hw_fromC
                 U puts
```

Linker – g++

- Linker performs *symbol resolution* through *external linkage i.e.* links *undefined* symbols and relocates memory addresses *(relocation).*
- **`g++ -o`** generates an executable from the object file with the same name as the value provided after **`-o`**.
- **`g++ --shared`** creates a *Shared Object* from the input object files.



The Archiver

- Object files which are *archived* create Static Objects.
- *The Archiver* (**ar**) could be used to *archive* object files -- **g++** will not invoke **ar**:

```
ar -rv StaticObj.a obj1.o …
```

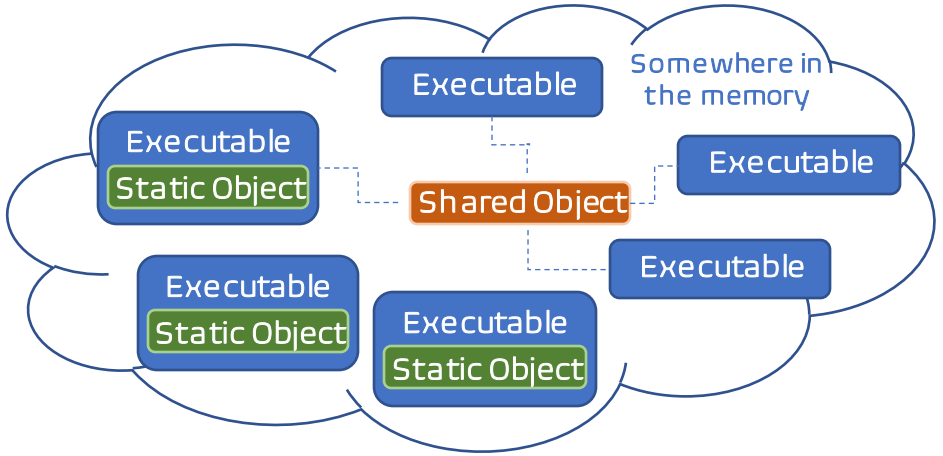
- **g++** (linker) can search for the needed *symbols* in the archive file (static object) and pull out needed definitions.

- Both static objects and shared objects provide code reusability – instead of *reimplementing* the same functionality, it could be turned out into an *object* and then *linked* into different executables.
- Static objects are also known as archive. And shared objects are known are also known as Dynamic objects.

"The archiver, also known simply as **ar***, is a Unix utility that maintains groups of files as a single archive file.* **ar** *is generally used only to create and update static library files that the link editor or linker uses and for generating* **.deb** *packages for the Debian family; it can be used to create archives for any purpose, but has been largely replaced* **tar***."* https://en.wikipedia.org/wiki/Ar_(Unix)

Static Object vs Shared Object

Shared Object	Static Object
Has no effect on the executable's size	Enlarges the executable's size
Faster compilation but slower execution	Slower compilation but faster execution
Faster load time	Constant load time
Possible compatibility issues while easily updateable	Zero compatibility issues while not updateable
Is *loaded* at run-time by OS	Is added at compile time by Linker



- In case the shared object is already loaded in the memory, the executables are loaded faster, however, this loading time for executables using static objects is always constant.
- In modern operating systems, there are techniques to reduce duplicate information in the RAM a.k.a. *Copy-On-Write* which could reduce the waste of redundant codes in static objects.

Libraries

- In C++, libraries are reusable *packages* consisting of:
  - Header file(s) defining provided functionalities a.k.a interface
  - Pre-compiled *object file(s)* of the implementation of the functionalities
- ***Static library*** has *static object* which is added to each executable at the compile time.
- ***Dynamic library*** has *shared object* which is *shared* amongst executable at run-time – only a small portion of shared object called *method stubs* are copied at *linking*.
- It is possible to load shared object at run-time without compiling them with the executable; these share objects are called **plug-ins**.

"A static library is like a bookstore, and a shared library is like... a library. With the former, you get your own copy of the book/function to take home; with the latter you and everyone else go to the library to use the same book/function. So anyone who wants to use the (shared) library needs to know where it is, because you have to "go get" the book/function. With a static library, the book/function is yours to own, and you keep it within your home/program, and once you have it you don't care where or when you got it." https://stackoverflow.com/a/2650053

- Shipping *machine code* in form of *object files* with the libraries, instead of the source code is beneficial both for the purpose of confidentiality and efficiency. As machine codes are pre-compiled; they reduce compilation time, besides, they are *very hardly* human-readable which could secure intellectual property.

Loader

- *"The loader's tasks include:*
  - *copying the program image from the disk into main memory;*
  - *validation (permissions, memory requirements etc.);*
  - *Static and global variables intilization ;*
  - *initializing registers (e.g., the stack pointer);*
  - *jumping to the program entry point."*

https://en.wikipedia.org/wiki/Loader_(computing)#Responsibilities

> *"In computer systems a loader is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution."* https://en.wikipedia.org/wiki/Loader_(computing)

- ***Dynamic linking loader*** (dynamic linker), is another part of the operating system which loads shared object into already loaded executable.

Build

- ***Build*** is the process of converting source code into binary.
- *Most* softwares have a rather *complex* build procedure, in which different libraries and source codes should be compiled and linked in the appropriate order – ***build automation tool***s facilate this.
- ***Build system*** employs *build automation tools* to build large projects – usually a build system generates needed artefacts for the build automation tool, depending on the system.

Makefile Syntax

```
Target1 : Optional_Prerequisites
    Command
    Command

Target2 : Optional_Prerequisites
    Command
Variable = Value
```

- GNU make or simply **`make`** is an application. It looks for a text file called **`Makefile`** which defines target builds. Invoking **`make`** ***`targetName`*** builds the target, if non provided, the first target would be built.
- GNU make is the most widespread *build automation tool* used in GNU/Linux systems. GNU Build System i.e. combination of *Autotools* and *Make*, is the favorite build system for many open source software. *Autotools* generates Makefiles depending on the platform and checks whether required build dependencies and system requirements are available.

- One of the richest and most-scalable **`Makefile`** or other build automation tools "*generator*"*!*
- Cmake enables software build:
  - On different platforms, OSes, using different compilers.
  - Without need of *hard-coded* dependency paths.
  - Build different versions of software and perform *more than build!*

"CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice." https://cmake.org/

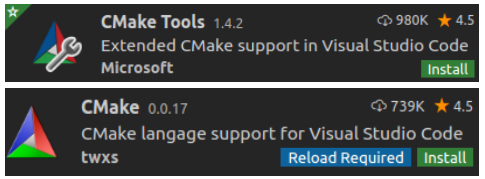
- There are many different build systems, and different projects, for very different reasons, may use specific build system. CMake is the most widespread building tool across all platforms.
- CMake could be installed on ubuntu using `apt`: `$> sudo apt install cmake`.
- CMake versions higher than 3.0 (2.8.2 to be exact) are considered *modern CMake* since the changes compared to previous versions were very major.

- CMake has its own syntax to define ***build configuration*** for a project.

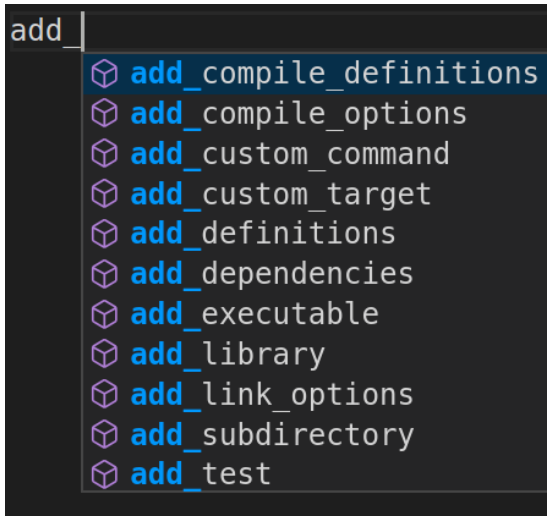
```
M CMakeLists.txt
1   cmake_minimum_required(VERSION 3.5.1)
2   project(foo)
3   add_executable(foo foo.cpp)
```

- In order to run **`cmake`** the path to the folder containing the *build configuration* – a text file called **`CMakeLists.txt`** – should be provided to the program.

- **`cmake`** *generates* files, including the **`Makefile`**, on the location it is invoked.



- Many IDEs support CMake and provide syntax highlighting features and auto-completion.




```
mrz@vbubu:~/foo$ ls
CMakeLists.txt  foo.cpp
mrz@vbubu:~/foo$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- 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
-- Detecting C compile features
-- Detecting C compile features - 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
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mrz/foo
mrz@vbubu:~/foo$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  foo.cpp  Makefile
mrz@vbubu:~/foo$ make
Scanning dependencies of target foo
[ 50%] Building CXX object CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX executable foo
[100%] Built target foo
mrz@vbubu:~/foo$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  foo  foo.cpp  Makefile
```

- Usually projects have a **`build`** folder from within which, **`cmake`** is called.
- it is a common practice to create that folder in the *root directory* of the project if it does not already exist.
- CMake could be used for build at different *levels.*

```
mrz@vbubu:~/proj$ cd build/
mrz@vbubu:~/proj/build$ cmake ..
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- 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
-- Detecting C compile features
-- Detecting C compile features - 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
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mrz/proj/build
```

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.5.1)
2  project(sample)
3
4  add_subdirectory(foo)
5  add_subdirectory(bar)
```

```
.
├── bar
│   ├── CMakeLists.txt
│   ├── include
│   └── src
├── build
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   └── Makefile
├── CMakeLists.txt
└── foo
    ├── CMakeLists.txt
    ├── include
    └── src
```

- **`cmake`** provides interface to perform all the process related to *the build*.

```
mrz@vububu:~/project$ cmake -S . -B build
mrz@vububu:~/project$ cmake -build build
```

Vs.

```
mrz@vububu:~/project$ mkdir build
mrz@vububu:~/project$ cd build
mrz@vububu:~/project/build$ cmake
mrz@vububu:~/project/build$ make
```

```
Generate a Project Buildsystem
 cmake [<options>] <path-to-source>
 cmake [<options>] <path-to-existing-build>
 cmake [<options>] -S <path-to-source> -B <path-to-build>

Build a Project
 cmake --build <dir> [<options>] [-- <build-tool-options>]

Install a Project
 cmake --install <dir> [<options>]

Open a Project
 cmake --open <dir>

Run a Script
 cmake [{-D <var>=<value>}...] -P <cmake-script-file>

Run a Command-Line Tool
 cmake -E <command> [<options>]

Run the Find-Package Tool
 cmake --find-package [<options>]

View Help
 cmake --help[-<topic>]
```

https://cmake.org/cmake/help/latest/manual/cmake.1.html

CMakeLists.txt

- Each *command* in **CMakeLists.txt** is separated with a *new line* ('\n') and comments are added using **#**.
- Every **CMakeLists.txt** starts with defining the version of the CMake to be used:

```
cmake_minimum_required(VERSION versionNumber)
```

- Each *top-level* **CMakeLists.txt** defines a project:

```
                Mandatory                Optional
project(projectName VERSION versionNumber
                    DESCRIPTION "Project's Description"
                    HOMEPAGE_URL "www.project.url"
                    LANGUAGES CXX)
```

- *Target*s (executables and libraries) are added using **add_executable** and **add_library** commands.

- Each CMake version enforces a certain policy, it is important which version is used.
- New versions of CMake (>3.12) support range for *versionNumber* in form of: "**VERSION** *minVersion* ... *maxVersion*". This means the project supports *minVersion* and it has been tested with policies upto *maxVersion*.

The CMake Policy mechanism is designed to help keep existing projects building as new versions of CMake introduce changes in behavior. https://cmake.org/cmake/help/latest/command/cmake_policy.html

Targets

ALTEN

- CMake can build as many *targets* as needed – the default target is to build *all* targets, except those which have set the **EXCLUDE_FROM_ALL** property.
- *Normal executables* are added as target with a *globally unique* name within the project.
- *Normal libraries* could be defined as either **STATIC** for static libraries, **SHARED** for dynamic libraries, and **MODULE** for plug-ins.
- There are specific commands to locate header files (**-I**), locate object files to be linked (**-l**) and their location (**-L**).
- It is possible to reference to targets outside the project i.e. *import targets.*
- There is possibility to define dependencies for targets

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
               [EXCLUDE_FROM_ALL]
               [source1] [source2 ...])
```

https://cmake.org/cmake/help/latest/command/add_executable.html

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

https://cmake.org/cmake/help/latest/command/add_library.html

```
add_dependencies(<target> [<target-dependency>]...)
```

https://cmake.org/cmake/help/latest/command/add_dependencies.html

```
target_link_libraries(<target> ... <item>... ...)
```

https://cmake.org/cmake/help/latest/command/target_link_libraries.html

```
target_link_directories(<target> [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

https://cmake.org/cmake/help/git-stage/command/target_link_directories.html

```
target_include_directories(<target> [SYSTEM] [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

https://cmake.org/cmake/help/latest/command/target_include_directories.html

Assignment 1

- Use CMake to build your mini project, preferably it uses multiple source files.

Assignment 2

- Use this tutorial (use g++ instead of gcc) to generate a shared object from the template *Shape Assignment*. Then write another application which uses the share object and creates a comparison between shapes.