CMake 2022

What is CMake

- Bill Hoffman at Kitware implemented CMake

- CMake has become industary standard for C & C++ programmers
- As programmer we design, code and test. This process is repetitive.
- We want to delegate all repetitive task to automating tooling.
    - So what this tooling should have
        - Able to execute all required workflows
        - Work across different environments
        - Support for various IDE
        - Support CI/CD pipelines
- CMake became popular because it kept updating for various general scenarios.

    > Simply it is an orchestrator of pre/post build processes.

- Available CMake Tools
    - `cmake` : Main executable for configuration, generation and building
    - `ctest` : Test driver program run and report test results
    - `cpack` : Generate installer and source packages
    - `cmake-gui` : Qt GUI wrapper over `cmake`
    - `ccmake` : Console wrapper over `cmake`

How CMake works

- On the surface it looks like CMake is a tool that reads our source code on one end and produces binaries on other end. But it is not completely true.
- CMake cannot build anything on it's own. It needs additional tools to perform its job.
- What Cmake knows is
    - What steps need to be done
    - What the end goal is
    - Find right tools and materials for the job
- CMake accomplish it's tasks in following three stages:
    - Configuration
    - Generation
    - Building

- CMake has it's own coding language.

Configuration Stage

- *Binary Tree*: Path to target or output directory
- *Source Tree*: Path to source code location
- *ListFiles*: File that contain CMake language
- In this stage, project details are read from the source tree and prepares the build tree for next stage i.e., generation stage.
- Details like architectures, compilers, linkers, archivers etc.
- Additionally it checks if simple test program can be compiled correctly.

Input

- CMakeLists.txt :
  - This file is the minimum requirement for cmake

Output

- CMakeCache.txt :
  - Stores variables for path tools and other. *Check CMakeCahe.txt files*
- Other files like system details, project configurations, logs and temp files

> For CMake to automatically detect the additional tools, the tools should be on environment PATH variable. Else we have to manually specify the path in CMake configurations.

> For various developers to have same developemnt environment , it is recommented to use Docker or like alternate tools.

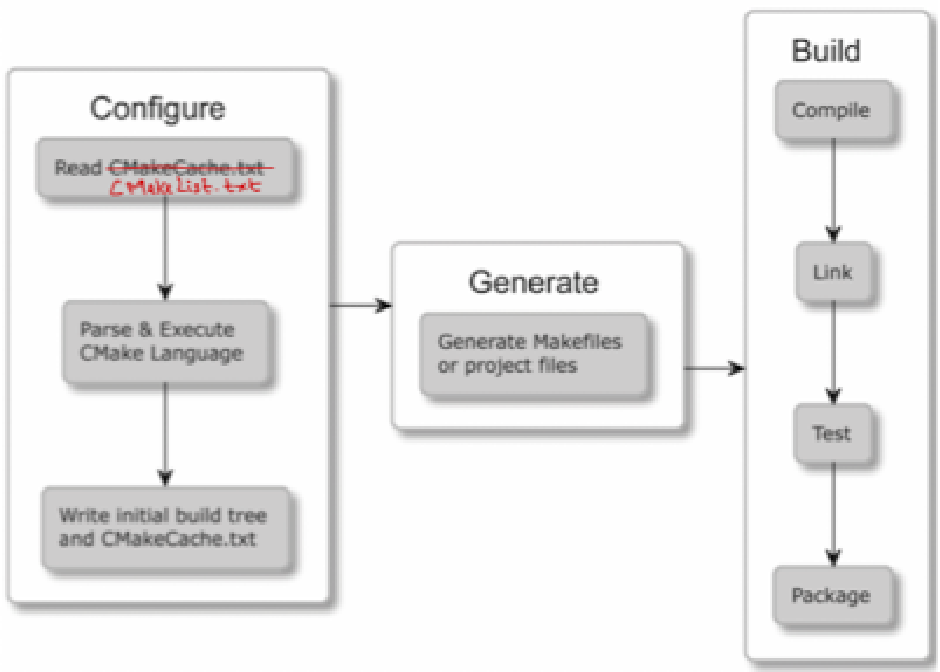
Generation Stage

- In this stage, s build system is generated based on the environment set in project configuration files.
- Generates build system files like
  - Makefiles for GNU Make
  - Ninja & IDE .vs files for etc.

> Geneartion stage is executed automatically after the configuration stage

- we can use `cmake-gui` to seperate configuration and generation stage.
-

Building Stage

- In this stage, build tools will use build system files to generate targets using CMake Command.
- We can also use IDE to use build system files generated to build targets required.
- 

Basic commands to Build

```
cmake -S /sourcetree/. -B /buildtree/.
```

```
cmake --build /buildtree/.
```

```
cmake -G <generator-name> <sourcetree>
```

- Loading cache information fromscript files

```
cmake -C <script-file> <sourcetree>
```

- Modification of cache variables

```
cmake -D <var>[:type]=<value> <sourcetree>

Type can be
- BOOL
- FILEPATH
- PATH
- STRING
- INTERNAL
- UNUNINTIALIZED

cmake -S <sourcetree> -B <buildtree> -D CMAKE_BUILD_TYPE=Release
```

- Listing cache variables

```
cmake -L <sourcetree>
```

- Print system information

```
cmake --system-information
```

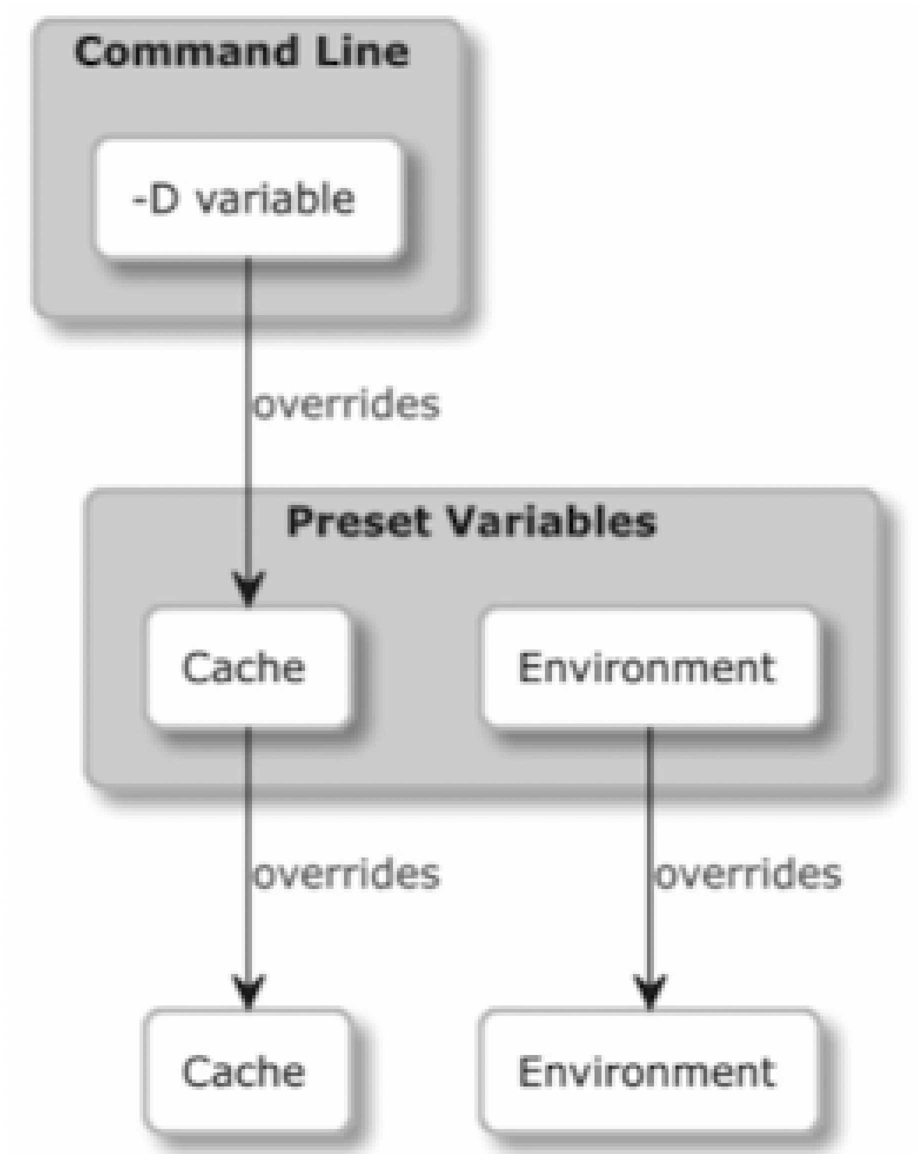
- Presets
  - There can be different preset cache environment variables that can be set to override.

```
cmake --list-presets
cmake --preset=<preset>
```

- There are two types of preset files
  - `CMakePresets.json`
  - `CMakeUserPresets.json`
- File =CMakePresets.json= looks like

```
{
  "name": "linux-default",
  "displayName": "Linux Debug",
  "description": "Sets Ninja generator, compilers, build and install directory,
debug build type",
  "generator": "Ninja",
  "binaryDir": "${sourceDir}/out/build/${presetName}",
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Debug",
    "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}"
```

```
  },
  "vendor": {
    "microsoft.com/VisualStudioSettings/CMake/1.0": {
      "hostOS": [ "Linux" ]
    },
    "microsoft.com/VisualStudioRemoteSettings/CMake/1.0": {
      "sourceDir": "$env{HOME}/.vs/$ms{projectDirName}"
   }
  }
}
```



- ListFiles
    - List files can be added using following
        - `include()`
        - `find_package()`

        - `add_subdirectory()`
    - Generally list files have `.cmake` extension (not necessory)

- Variable references

```
${} - normal or cache variables
$ENV{} - environment variables
$CACHE{} - cache variables

set(<variable> <value>)
unset(<variable>)
```

How CMake & C++ work together

Targets

- Target can be any type of artifact like executable or libray or scripts or object files etc.
- CMake can create target using folowing commands
    - `add_executable(<targetname> <parameters>)` - Generate executable files from linking object files
    - `add_library(<targetname> <parameters>)` - Generate library file or archiving
    - `add_custom_target(<targetname> <parameters>)` - These allow us to specify our own command line that will be executed without checking whether produced output is up to date

```
add_custom_target(NAME [ALL] [command1 [args1...]]
                                          [COMMAND command2 [args2...] ...]
                                          [DEPENDS depend1 depend2]
                                          [BYPRODUCTS [files...]]
                                          [WORKING_DIRECTORY dir]
                                          [COMMENT comment]
                                          [JOB_POOL job_pool]
                                          [VERBATIM] [USES_TERMINAL]
                                          [COMMAND_EXPAND_LISTS]
                                          [SOURCE src1...])
```

- Project can be build both from external internal dependencies, this is where targets are useful.
- Executables are connected with libraries using

```
target_link_libraries(<executable-target> <library-target>)
```

- CMake attempts to build directional acyclic graph. CMake doesn't know order of execution of targets unless we specify.
- Dependency between targets can be specified using

```
add_dependencies(<target-A> <target-B> <target-C> ...)
```

- Target dependency graph can be visualized using GraphViz

```
cmake --graphviz=<filename>.dot <sourcetree>
```

Target Properties

- Target properties in CMake are same as fields in C++ objects.
- Some properties can be modified and other can be read only.
- These properties varies from executable / libraries / custom targets
- We can also add our own properties

```
set_target_property(<target-A> <target-B> ...
                                        PROPERTIES <property-name> <property-value>
....)
get_target_property(<variable-name> <target> <property-name>)
```

- Properties are not unique to targets but can be for various scopes
    - GLOBAL / DIRECTORY / SOURCE / INSTALL / TEST / CACHE

```
set_property(TARGET <target-name> PROPERTY <property-name> <property-value>)
get_property(<variable-name> <scope> PROPERTY <property-name>)
```

- Transitive Usage Requirements
    - Propagation of properties between source target and destination target. (This is when we have dependency)
    - Populates COMPILE_DEFINITIONS property of source target

```
target_compile_definitions(<source> <PRIVATE|INTERFACE|PUBLIC> [items-A...])
```

- `PRIVATE` : Sets the property of the source target

    - Used when the definitions will be applied just for the current target

- `INTERFACE` : Sets the property of the destination targets

- Used when the definitions need to be applied to targets that link against the given target
    - Example usage can be header only libraries. (there is no actual linking as there are only header files & no object files to link)

- `PUBLIC` : Sets the property of the source and destination targets

- Writing custom commands

    - They will be build every single time.
    - `add_custom_command` has two signatures
    - Using custom command as generator

```
add_custom_command(OUTPUT output-A output-B ...
                                        COMMAND command-A [args...]
                                        COMMAND command-B [args...]
                                        DEPENDS [depends...])
```

- Using custom command as target hook

```
add_custom_command(TARGET <target-name>
                                        PRE_BUILD | PRE_LINK | POST_BUILD
                                        COMMAND command-A [args...]
                                        COMMAND command-B [args...])
```

- `PRE_BUILD` : Will run before any other rules of target
- `PRE_LINK` : Will run after compiling and before linking
- `POST_BUILD` : Will run after all the rules of the target are run

Working of C++ program

- Generally there are five steps in creating and running C++ program:
    - `Writing` : Designing and writing source code using c++ code constructs / syntax.
    - `Compiling` : Compile individual .cpp implementation file called translation units to object files.
    - `Linking` : Link object files together in a single executable and add all dependencies i.e., dynamic and static libraries
    - `Loading` : To run program, the OS will use a tool call `loader` to map its machine code all required dynamic libraries to the virtual memory.
      The `loader` then reads the headers to check where the program starts and hand over the control to the code.

- `Execution` : C++ runtime kicks in; a special `_start` function is executed to collect the comman-line arguments and environment variables.
  It starts threading, initializes static symbols, callbacks to register cleanup. Only then will it call main(), which is filled with code by programmer.

Compiling of C++

Basics of compilation

- Compilation can be said as the process of translating instructions written in a higher-level programming language to a low level machine code.

- Compiler removes the burden of user handling the processor-specific assembly language i.e., working with registers, manage stack frames.

- In C++ we rely on static compilation - entire program has to be translated to native code before execution.

- Java or python does this via interpreter on the fly every time a program is run.

- Output of the compilation process is `binary object file format`, which is specific for a given platform.

  - `LINUX` : Executable and Linkable Format (ELF)
  - `WINDOWS` : Portanle Executable format (PE)
  - `MAC-OS` : Mach objects (Mach-O format)

- Compiler has to execute following steps to create object files:

  - `Preprocessing`
    - Manipulation of source code in a very rudimentary way
    - Replace directives like `#include` , `#define`, `#if,#elif`
    - Basically it is more advance find & replace tool
  - `Lingustic Analysis`
    - `Lexical Analysis` - Scan the file character by character & grouping them into meaning full `Tokens` like keywords, operators, variable names etc.
    - `Syntax Analysis or Parsing` - `Tokens` are grouped into token chains & verified if their order and presence follow the c++ rules.
    - `Semantic Analysis` - Verify if the statements in a file actually make sense i.e., like type correctness
  - `Assembly`
    - Translation of above `Tokens` to CPU-specific instructions based on an instruction set available for the platform.

        - Generate assembly code
    - `Optimization`
        - Minimize the usage of register and removing unused code.
        - Like `inlining`
    - `Code emission`
        - Writing the optimized machine code into object file for a specific platform

- CMake Commands

    - `target_compile_features(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Require a compiler with specific features to compile this target
    - `target_source(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Add sources to already defined target
    - `target_include_directories(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Setup preprocessor include paths
    - `target_compile_definitions(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Setup preprocessor definitions
    - `target_compile_options(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Compiler specific options for command line
    - `target_precompile_headers(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)` - Optimize the compilation of external headers

- Managing sources of targets

    - As project size increases we may need to include multiple source files to target

        ```
        add_executable(<target-name> <src-A.cpp> <src-B.cpp> <src-C.cpp> ...)
        ```

    - It is not possible to use 100's of files. We may use like

```
file(GLOB SampleSource "*.h" "*.cpp")
add_executable(<target-name> ${SampleSource})
```

    - The above method a drawback:
        - CMake generates build files based on changes in the list files.
        - GLOB collect and stores it as one variable which is cached.
        - This can effect when multiple developers build and commitwith out reconfiguration/ regeneration.
    - To solve this problem we need to use following command :

```
target_source(<target-name> <PRIVATE|INTERFACE|PUBLIC> <value>)
```

Preprocessor Configurations

- `#include` directive is used in preprocessing to replace with header files.
- This comes in two forms:
  - `#include <path-spec>` : Angle-bracket form
    - This will check the standard include directories including C++ standard library headers directory
  - `#include "path-spec"` : Quoted form
    - First checks the directory of current file
    - And then checks the directories of angle-bracket form

```
target_include_directories(<target-name> [SYSTEM] [AFTER|BEFORE]
                                                        <PRIVATE|INTERFACE|PUBLIC>
[item1...])
```

- These are generally represented with compiler flags `-I`

```
target_compile_definitions(<target-name> <PRIVATE|INTERFACE|PUBLIC> <DEFINE-
VARIABLE>)
```

- These are generally represented with compiler flags `-D`

- Setting multiple compile definitions on command line can be diffficult. In such scenarios we can use `configure_file`

  - Create a file `configure.h.in`

  ```
  #cmakedefine SAMPLE_DEF "@DATA_PASSED@"
  ```

  - Add following definitions in CMake file

  ```
  set(DATA_PASSED "print string")
  configure_file(configure.h.in configure/configure.h)
  ```

Optimzer Configurations

- Only purpose is to make the execution faster
- Can remove code
- Can duplicate code etc.
  - Example of `inlin`
  - Examople of `for` loop

```
target_compile_options(<target-name> [BEFORE]
                                                  <PRIVATE|INTERFACE|PUBLIC>
[items...])
```

- This is a larger topic related to internal working of compilers, processors and memory.
- Here, the focus is more on general optimization levels
- There are forur levels of optimization from `-O0 to -O3` i.e., `-O<level>`
  - `-O0` : No Optimization (Default)
  - `-O1` : in between 0 & 2
  - `-O2` : Full optimization (Highly Optimized code & Slowest compilation time)
  - `-O3` : Full Optimization with more aggressive approach to subprogram inlining & loop vectorization
- CMake gives option to set the `CMAKE_CXX_FLAGS_DEBUG` & `CMAKE_CXX_FLAGS_RELEASE`. These are global scoped.
- Inline optimization can be controlled with folowing flags
  - `-f` : `-finline-functions` - Enable
  - `-fno` : `-fno-inline-functions` - Disable
- Loop unrolling
  - `-floop-unroll`
- Loop vectorization
  - `-ftree-vectorize -ftree-slp-vectorize`

Managing process of compilation

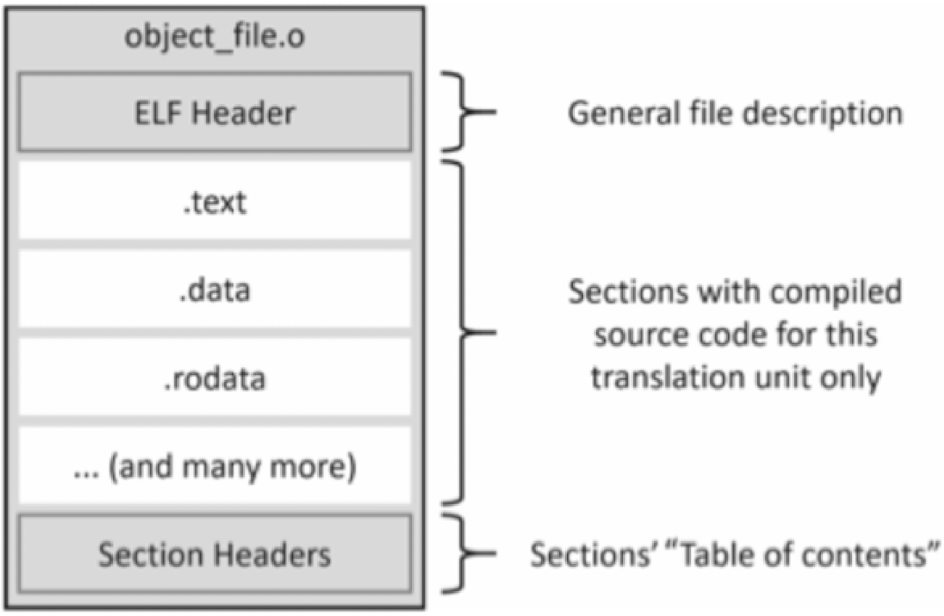
- Reducing compilation time
  - Precompilation of headers
    - `target_precompile_headers(<target-name> <PRIVATE|INTERFACE|PUBLIC> [headers...])`
    - `traget_precompile_headers(<target-name> REUSE_FROM <other-target>)`
  - Unity Builds
    - Headers included in multiple targets are compiled only once
    - In regular C++ project not so useful
    - Very useful for Qt projects, can reduce build time

Linking of C++

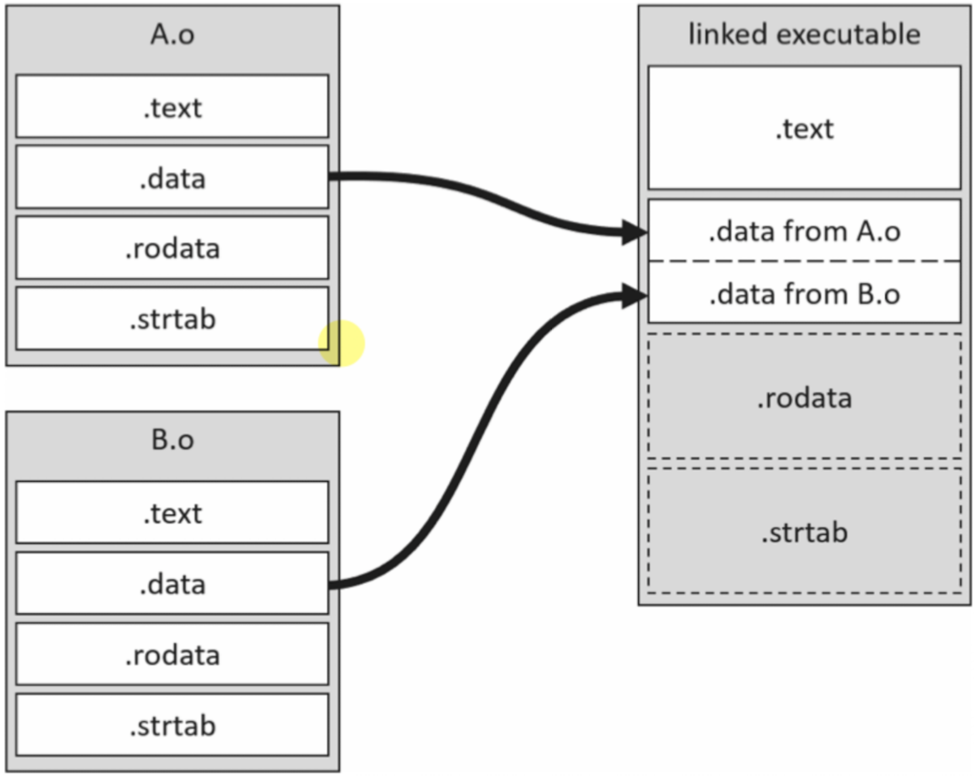
- There is only one command in terms of linking: `target_link_libraries()`

Basics of Linking

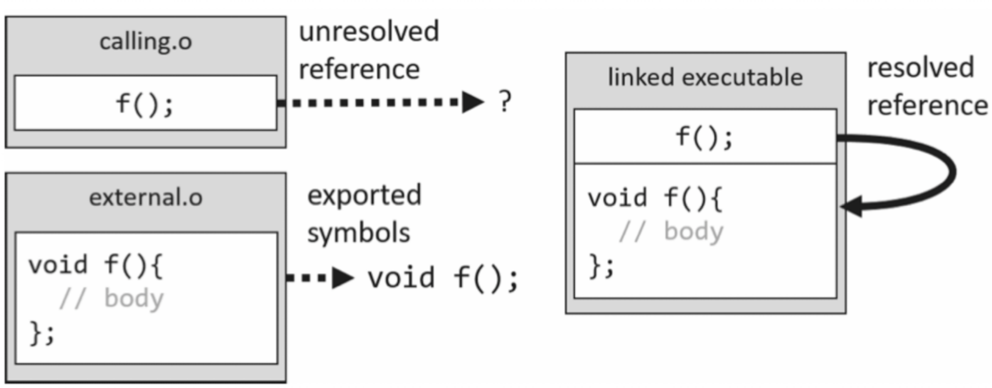
- Why the object files cannot be executed by processor ?
- Structure of object file. Unix like system (ELF)



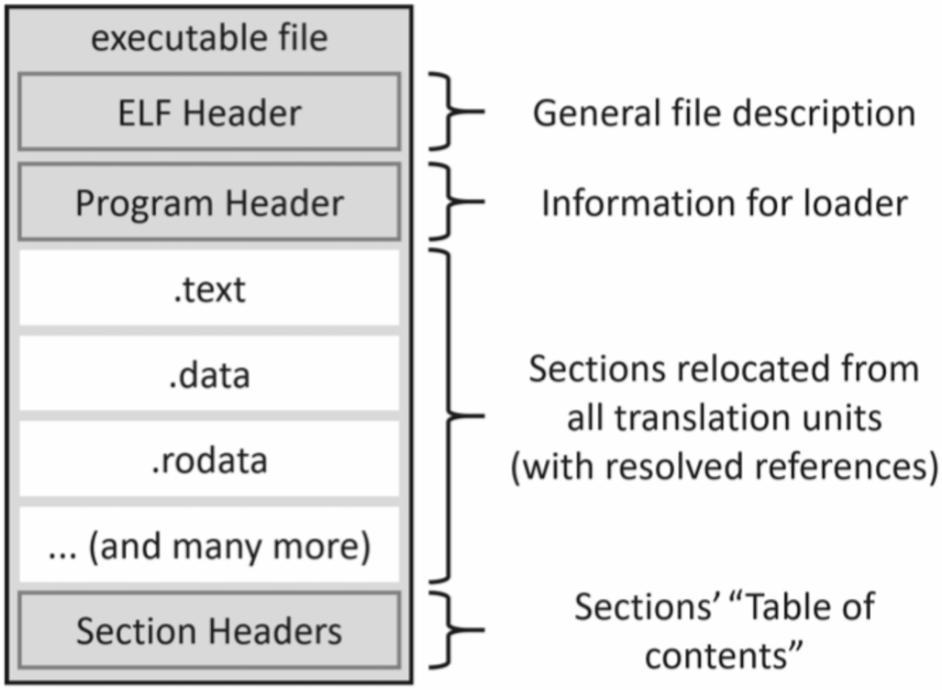
- ELF file sections are relocated to single file along with update of address of variables, functions, symbol tables etc.



- Reolved references



- Executable File Structure



Building different library types

- `Static Library` - Collection of raw object files stored in an archive
- `Shared Library` - Shared Object
- `Shared Modules` - Plugin - Shared library loaded during runtime

```
add_library(<target-name> STATIC <source...>)
```

```
add_library(<target-name> SHARED <source...>)
```

```
add_library(<target-name> MODULE <source...>)
```

- Position Independent Code
  - Shared Libraries have to be compiled with flag PIC i.e., `-fPIC`
- CPU uses memory management unit (MMU) to translate a virtual memory address to physical memory address

One Definition Rule

- Names have to be unique
- use namespace to link diffrent symbols in libraries (not madatory)

Order of Linking & unresolved symbols

- Linker processes the binaries from left to right in `target_link_libraries(<target-name> <library-name> <library-name>)`

Testing Framework

- Check GTest notes

Program Analysis Tools

CppCheck

- Check C++ Static Analysis & MISRA notes

Valgrind

- Check CUDA & QT notes

Document Generation

Doxygen

- Check DOXYGEN notes

Adobe Hyde

- Not well suitable for C++ projects

Standardese

- Still in working not stable. Many issues not resolved

Installation

- Check CMake Project

Packaging

- Need to test

Additional Parameters & Commands

- Some may parameters and changes with version of CMake. So refere reference manual.