CMake

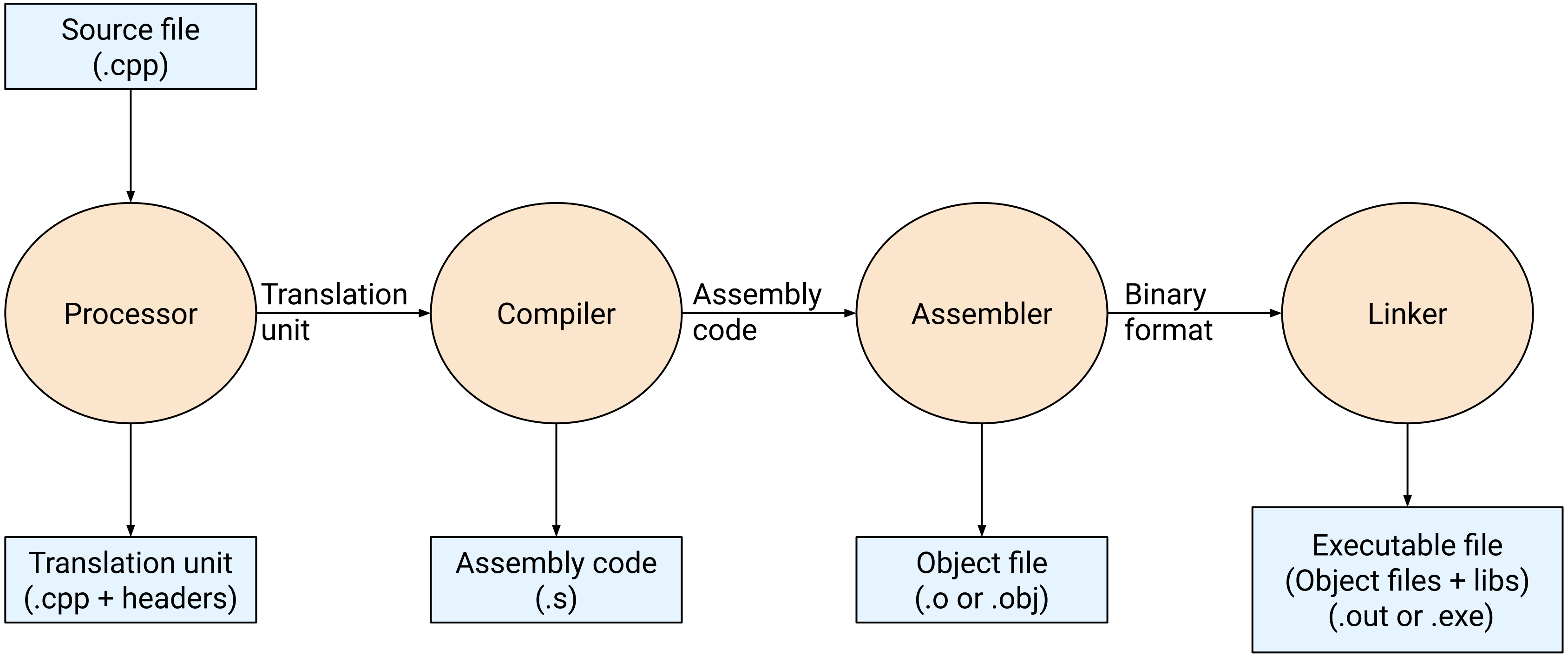
(build system)

Content

-

1) Introduction

How the program is executed



1) Preprocessor

- Macros and header files are replaced , and all comments are removed.
- The output is called translation unit.

```
#include <iostream>
using namespace std;

int add(int a, int b);

int main() {
    int result = add(3, 4);
    cout << "Result: " <<
result << endl;
    return 0;
}

int add(int a, int b) {
    return a + b;
}
```

```
// <iostream> is replaced by its
actual content,
namespace std {
    class ostream { /* ... */ };
    extern ostream cout;
    ostream& operator<<(ostream&
out, const char* str);
      ...............................
}

int add(int a, int b);

int main() {..........}

int add(int a, int b) {..........}
```

2) Compiler

The high- level C++ source code in the translation unit is translated into assembly code at this stage. In this stage:

- The compiler checks for syntax error.
- The functions signature are matched with their definitions, ensuring that the correct types are used.
- Functions names are mangled to ensure uniqueness, especially for overloaded functions.

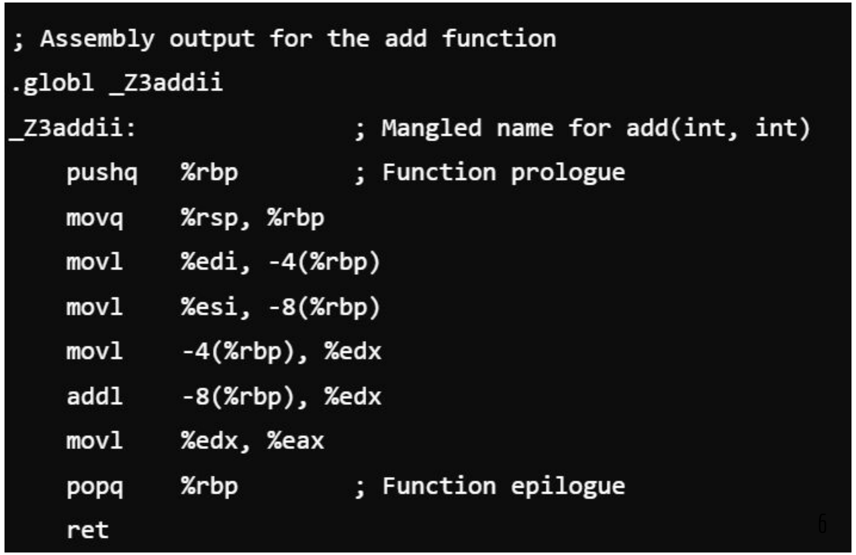
Functions signature (Unique):

Function name + parameters + return type. *Ex:* int add(int a, int b)

The function signature is: add + int a, int b + int

int add (int a, int b) is mangled into _z3addii

- _z → an indication to the start of mangled name.
- 3add → 3 if the length of the function name.
- ii → number of parameters of the function.



```
; Assembly output for the add function
.globl _Z3addii
_Z3addii:                   ; Mangled name for add(int, int)
    pushq   %rbp            ; Function prologue
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    addl    -8(%rbp), %edx
    movl    %edx, %eax
    popq    %rbp            ; Function epilogue
    ret
```

2) Compiler - cont.

Compiler creates object file whose one of the significant components is the symbol table. This tables contains all functions' signature after performing name mangling on them whether defined or not defined at the same object file, and their virtual addresses (not real ones because they haven't loaded to the memory yet).

This will help the linker letter to match the functions with their definition location/address.

print(int)
print(double)
add(int, int)

Function signature	Address where the definition be located inside/outside the same object file
_z5printi	0x2000
_z5printd	0x2010
_z3addii	0x2040

imaginary symbol table to simulate how it is created

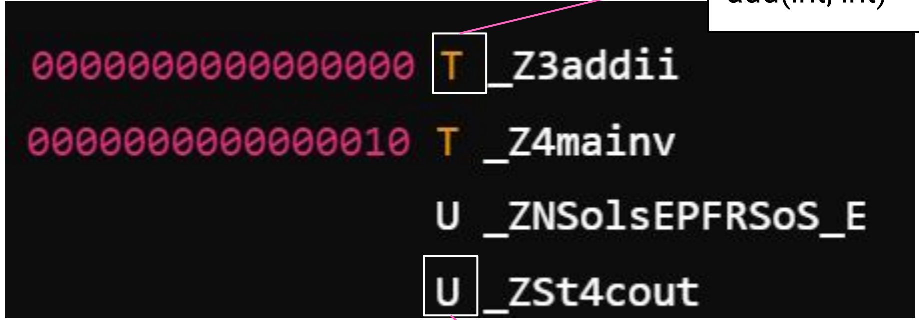
3) Assembler

It generates machine code and results in an object file (.o / obj).

This object file contains the binary code for the source file, where each translation unit has its own object file. It also includes references(mangling) to functions defined outside each translation unit.

Ex. for the same source file:

T indicates a defined symbol such as add(int, int)



```
0000000000000000 T _Z3addii
0000000000000010 T _Z4mainv
                 U _ZNSolsEPFRSoS_E
                 U _ZSt4cout
```

Note that: return type is not a part of the signature; just parameters; That's why we can't overload using return type

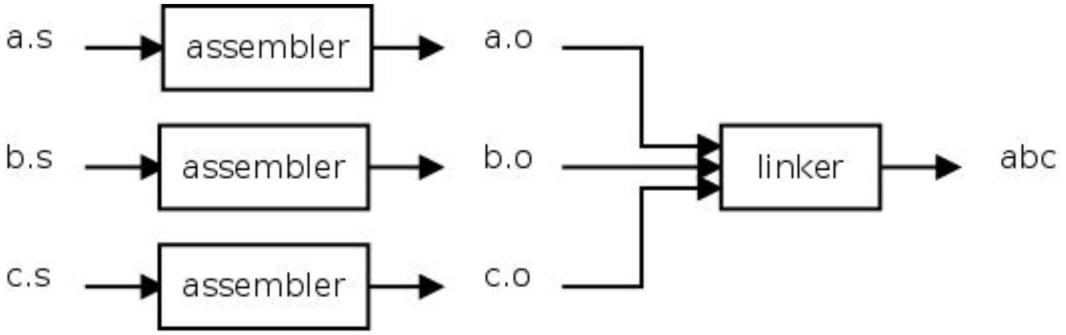
U indicates an undefined symbol, which will be resolved during linking. (e.g., std::cout)

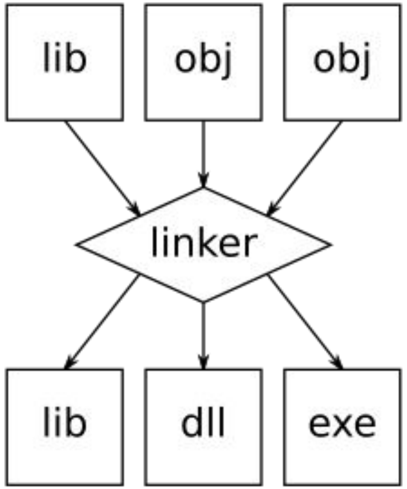
4) Linker

It combines all object files and resolves all their references to external functions and variables. It not only links functions together, it also links in external libraries, such as C++ standard library. There are two tasks in this stage:

- It matches function calls to their definitions through their signature (after mangling). This means that if a function is defined in another file, the linker matches it with its reference to that file's object code.
- It also determines whether each symbol (functions, global variables, static variables, etc.) will be allocated in memory at runtime. (Where to store them in conclusion).

The *output* is the executable file (.exe on Windows), or libraries (.dll / .lib).





Static VS. dynamic linking

Static linking	Dynamic linking
Integrated in the compile time. The precompiled code of external files and libraries is copied into the executable file, which in turn will increase the size of the executable file. Additionally, it is necessary to re-compile the executable in case of updating the library, which can be time-consuming for large applications.	Integrated on run-time The code of those external file is not copied into the executable. Instead, a table of references to those shared/ dynamic libraries is created, which reduces the size of the executable files in libraries. Additionally, shared/ dynamic libraries allow easier updating and maintenance as it can be updated dependently in the executable file. On the other hand, it can be a small cost for loading the library at runtime especially for large libraries.

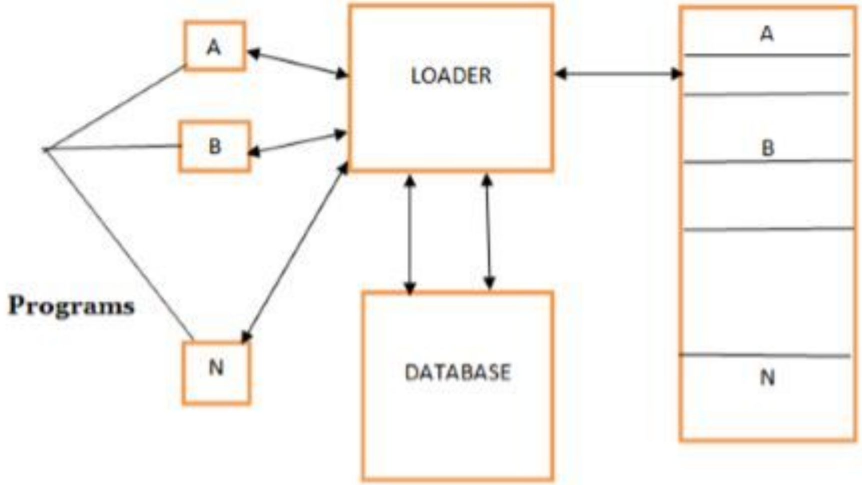
5) Loading and execution

When the program is run, the loader (which is a part of the OS) performs tasks:

- Load the executable files from disk into memory.
- Set up the memory segments through the addresses included in the executable file.
- If the executable is dynamically linked to libraries, it will resolve them.
- Jump to the entry point of the program (main()), whose address is provided in the executable file as well.

Execution flow:

- The CPU starts executing instructions at the main() function.
- When main() calls add(3, 4) for example, the control transfers to add() function in memory.
- Once add() returns, execution goes back to main(), and when main() finishes, the control returns back again to the operating system.



More about loader

Loading the executable file into memory

The executable file contains headers which is used to determine where the parts of the program should be loaded.
- Code segment: includes the machine instructions of the program.
- Data segment: where the global and static variables are loaded.
- Stack: where the local variables and function call is managed.
- Heap: where the dynamic memory allocations are done.

Setting up memory segments

Each of the memory parts (code segment, data segment, stack, and heap) is marked with different permissions:
- Text: Read-only and executable.
- Data: Readable and writable.
- Stack and Heap: readable and writable, but not executable.

Resolving dynamically linked libraries

If the program uses dynamic linked libraries, the loader must ensure that these libraries are loaded already in the memory. This is done by checking at the import table for the external libraries, will load them if not loaded, and then use relocation information to update the program's memory addresses.
For example: cout<<

Jumping to the entry point (main())

The control is transferred to the entry point through its address in the executable file by the loader. But, before this, there are some operations are done, such as initializing global variable and the standard I/O streams (inputs stdin) and (output stdout).

Before CMake

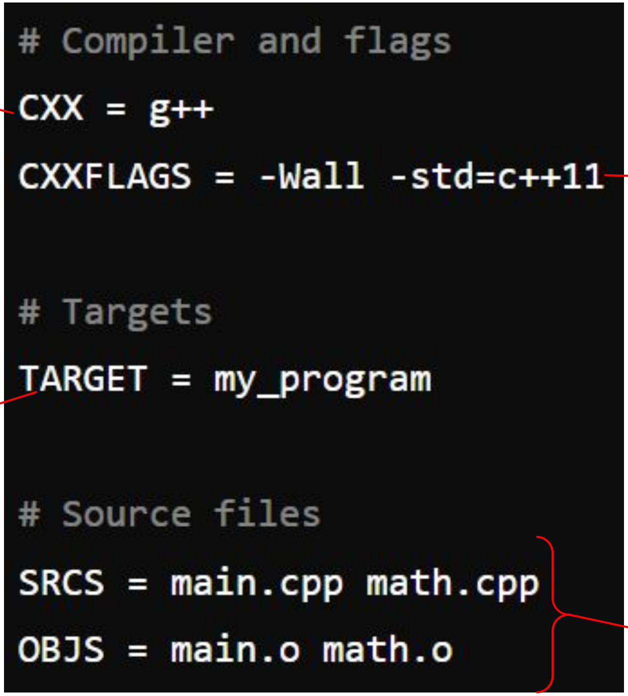
We can set the settings and properties of a project via the setting of the visual studio project. But if the project contains multiple files of multiple libs for example, it will be necessary to set them for each file/lib/etc. This is impractical situation.

To manage this issue, there is a file created with the properties of the project and the connected files for example (dependencies), and how this project will be compiled, linked, and so on.

This setting file is called:

- Makefile in Linux
- .sln file for visual studio (On windows) ⇒ includes multiple projects (files/ libs/ etc..), each inner project has its own .vccproj file. This file contains all instructions for building its specific project. But .sln contains instructions of combination of these inner projects and how they are executed and linked together.

Example for makefile in linux



```
# Compiler and flags
CXX = g++
CXXFLAGS = -Wall -std=c++11

# Targets
TARGET = my_program

# Source files
SRCS = main.cpp math.cpp
OBJS = main.o math.o
```

Compiler g++

Flags like -wall for warnings, and cpp version

.exe file name (target)

Source files and object files in the project

Example for .sln file in VS.

Note that this file contains another file for the configuration of each project (i.e. file/ lib) which is .vcxproj which needs to be created also

```
VisualStudioVersion = 16.0.28701.123
MinimumVisualStudioVersion = 10.0.40219.1
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "MyApp",
"MyApp\MyApp.vcxproj", "{A84F2C7C-17C6-4B29-AFA5-7E4C5A5B123D}"
EndProject
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "MyLib",
"MyLib\MyLib.vcxproj", "{B54A2A9D-3BF1-483A-B934-C6B3E64C68A6}"
EndProject
```

The first GUID(global unique ID) tells the project type such as c#/ c++/ ETC.
The 2nd string is the name of the project
The 3rd path is the path to the .vcxproj file of the project
The 4th GUID is a unique identifier for the project within .sln file.

After CMake

Thus, In linux, we need to create makefile for each project and one for the whole solution.
In Windows, we need to create .vcxproj files for each project and one .sln file for the whole solution!

Which is really a big headache, especially for large scale projects. Additionally, it we try to run .sln file on linux, the run won't be succeeded so we need to create also makefiles to be able to run the application on Linux, which is another hassle.

CMake will create those files for us based on the operating system we use, by writing the configuration of the whole project just once.

Conclusion

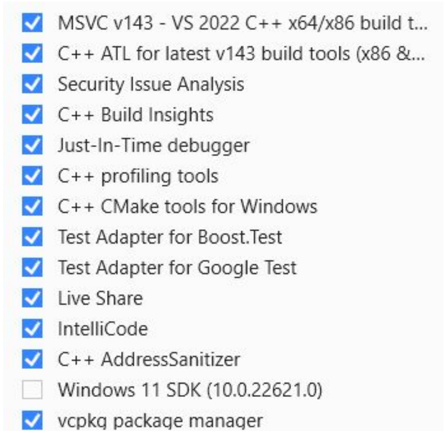
	Before CMake	After CMake
Platform specific build files	If you want to run your project on different platforms, you need to manually maintain separate build files, which can be time consuming and error-prone.	You write one CMakeLists.txt file that works on all platforms. CMake automatically adapts to the platform you are working on. No need to manually create separate build files for each system.
Third-party libraries	Each platform may have different paths or versions of libraries, and manually configuring this for each platform can be complex.	You don't have to worry about platform specific paths and linking commands for external libraries. CMake handles it for you, and the libraries work across platforms without modification. This is done through find_package() command, and using target_link_libraries() command for linking.

Friendship ended with
Makefiles
And Solution files(.sln)
Now
CMake
is my best friend

Installation: **This considers you have vscode installed on your environment.**

1) **Install a** compiler (Visual studio is recommended for vscode on windows as it is the default one).

Choose Desktop Development with C++ workload, here are the recommended ones, besides windows SDK.



- ✓ MSVC v143 - VS 2022 C++ x64/x86 build t...
- ✓ C++ ATL for latest v143 build tools (x86 &...
- ✓ Security Issue Analysis
- ✓ C++ Build Insights
- ✓ Just-In-Time debugger
- ✓ C++ profiling tools
- ✓ C++ CMake tools for Windows
- ✓ Test Adapter for Boost.Test
- ✓ Test Adapter for Google Test
- ✓ Live Share
- ✓ IntelliCode
- ✓ C++ AddressSanitizer
- ☐ Windows 11 SDK (10.0.22621.0)
- ✓ vcpkg package manager

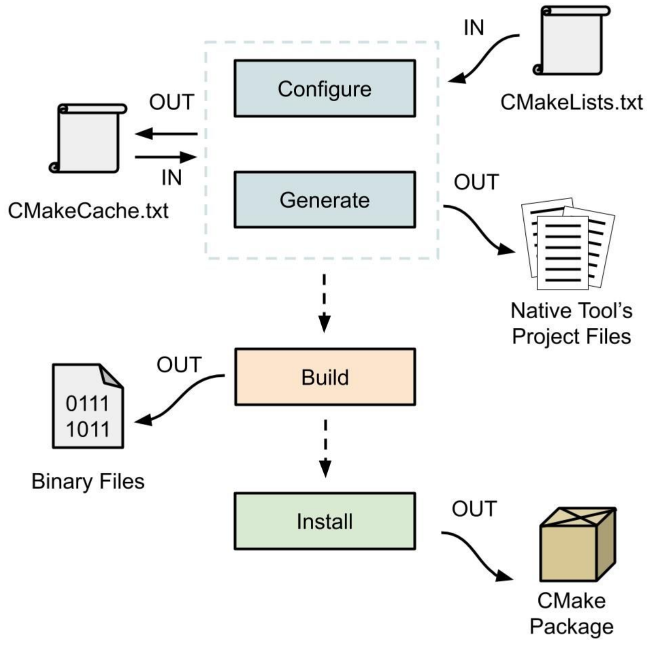
2) **Install** CMake and add it to the systen path. Ensure this is done by cmake --version command on cmd.

3) **Set Up VSCode for CMake:** by downloading CMake Tools extension

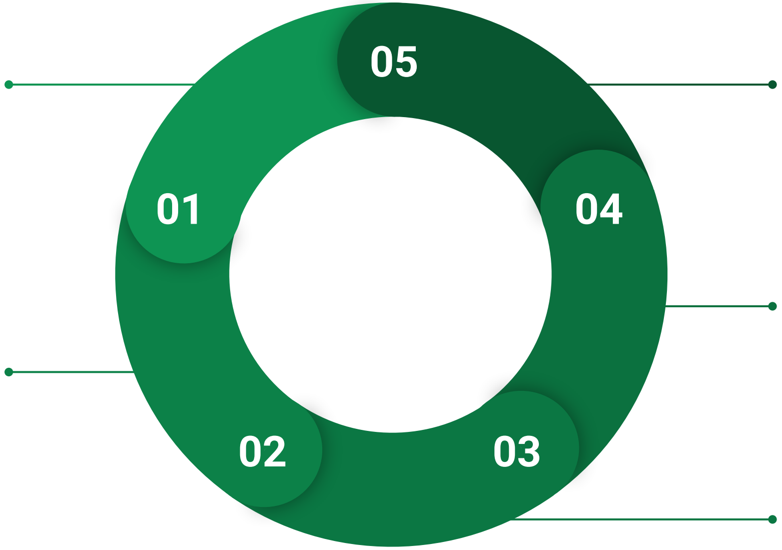
4) Finally, configure the project and build.

CMake workflow

1. The input file is `CMakeLists.txt` which is the configuration file of the project. It tells CMake how to organize the project, which source files to compile and which libraries to link, and other build settings. CMake uses it to configure later and then generate the appropriate build system (e.g. Makefile for linux, and .sln for Visual Studio).
2. CMake reads the configuration file, processes it, and then creates and updates CMakeCache.txt file where it stores information about configuration such as paths to compilers, libraries, and dependencies & variables it uses through the configuration step.
3. After configuration, CMake generates the build systems files required by the platform. The output will be the project file that can be run on any native build tool or platforms.
4. Then, the build tool uss the generated project files to compile the source code into executable files.
5. Finally, the compiled binaries are installed to the designated install locations, where we can open the application through them.



CMake build process:: Configure



Reading CMakeLists.txt

Understands the project name, source files, dependencies, and external libraries, etc…

Detecting compilers and tools

CMake finds the compiler on your system and checks if it can be used, if the compiler is missed, cmake will throw an error. It also identifies Toolchain includes linkers and MSBuild for example to link the compiled object files.

Locating dependencies and libraries

It looks for external libraries (find_package()) and check their paths if correct, it not, it will throw an error or warning.

Generating build files

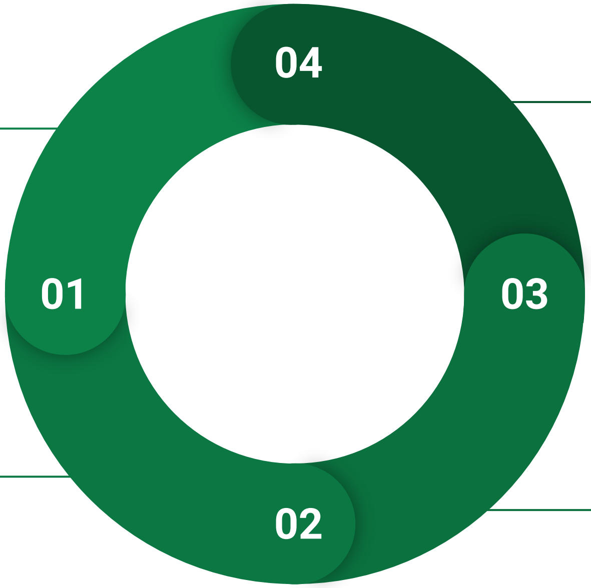
Makefile for linux, and MacOS, and MSBuild file for Visual studio on Windows.

Caching information

CMake creates CMakeCache.txt to store conf. Variables, paths to compilers, and libs, and compiler options

Triggered by the command: `cmake -S . -B build`. But we need to create a build folder firstly to store all created files in it, we can do it by command `mkdir build`.

CMake build process:: Build



Invoking the build system

Passes the control to the underlying build system to start compilation and linking (execution) process.

Compilation + Tracking dependencies

Producing final executable

Linking

Compiler vs. Generator vs. IDE

	Compiler	Generator	IDE
Purpose	Translate source code into machine code.	Generates platform-specific build files (Makefiles, MSBuild)	Provides a complete software environments for writing, combining, debugging, etc..
Examples	GCC, Clang, MSVC	Ninja, Visual Studio, Unix makefiles	Visual studio, VSCode, CLion
Interactio n	Compiles code based on commands or build files	Creates build files (e.g. Makefiles, .sln) that tell the build system how to compile and link the code	Provides a GUI for coding, debugging, etc..