1 - Getting started using Make

1.1 - Installing g++

1.1.1 - Check if g++ is installed:

The following command returns the g++ version. If not installed, will return 'not found'

```
$ g++ -v
```

If not found, go to next topic.

1.1.2 - Installing g++ :

If g++ version was not found, run the following command:

```
$ sudo apt install g++
```

Than the following command again and check if it returns the g++ version.

```
$ g++ -v
```

1.2 - Creating first C++ program

1.2.1 - Preparing the directories:

The following command will prepare the directories to store our first project and open it on VSCode:

```
$ cd ~/Desktop
$ mkdir CMakeLearning
$ mkdir CMakeLearning/1st_Module
$ mkdir CmakeLearning/1st_Module/1st_Example CMakeLearning/1st_Module/2nd_Example
$ mkdir CmakeLearning/1st_Module/3rd_Example CMakeLearning/1st_Module/4th_Example
$ mkdir CMakeLearning/1st_Module/5th_Example
$ mkdir CMakeLearning/2nd_Module
$ mkdir CMakeLearning/2nd_Module/1st_Example CMakeLearning/2nd_Module/2nd_Example
$ mkdir CMakeLearning/2nd_Module/3rd_Example
$ mkdir CMakeLearning/3rd_Module
$ mkdir CMakeLearning/3rd_Module/1st_Example CMakeLearning/3rd_Module/2nd_Example
$ mkdir CMakeLearning/3rd_Module/3rd_Example CMakeLearning/3rd_Module/4th_Example
$ mkdir CMakeLearning/4th_Module
```

```
$ mkdir CMakeLearning/4th_Module/1st_Example CMakeLearning/4th_Module/2nd_Example
$ mkdir CMakeLearning/4th_Module/3rd_Example CMakeLearning/4th_Module/4th_Example
$ mkdir CMakeLearning/4th_Module/5th_Example CMakeLearning/6th_Module/2nd_Example
$ mkdir CMakeLearning/4th_Module/7th_Example CMakeLearning/4th_Module/8th_Example
$ code ./CmakeLearning
```

1.2.2 - 1st example - Starting with g++:

Lets create a single electonics calculator. Lets create an empty C++ in the 1st example folder inside 1st module:

```
$ cd CMakeLearning/1st_Module/1st_Example
$ touch main.cpp
```

In the main.cpp, lets write the following code:

```cpp
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
    double current, voltage, power, resistence;

    cout << "What is the resistence in Ohms? " << endl;
    cin >> resistence;

    cout << "What is the voltage in Volts? " << endl;
    cin >> voltage;

    current = voltage / resistence;
    power = voltage * current;

    cout << "Current [A]: " << current << endl;
    cout << "Power   [W]: " << power << endl;

    return 0;
}
```

Lets generate an executable giving directly the arguments to g++ and run the generated executable.
Lets run the following commands: $ `./electronics_calculator_1_1`

```
$ g++ main.cpp -o electronics_calculator_1_1
```

```
$ ./electronics_calculator_1_1
```

Than we have the following output:

```
rafael:~/Desktop/CMakeLearning/1st_Module1st_Example$ ./electronics_calculator_1_1
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

1.2.3 - 2nd example:

Lets increase our last code and make it more modular. Lets create an empty C++ file and open the 2nd module directory (you may do it in VSCode terminal) and write the C++ code into it:

```
$ cd ../2nd_Example
$ touch main.cpp
```

```cpp
#include <iostream>

using namespace std;

double Current(double volt, double ohms){
    return volt / ohms;
}

double Power(double volt, double ampere){
    return volt * ampere;
}

void PrintResult(string variable_name, double value){
    cout << variable_name << ": " << value << endl;
}

int main(int argc, char* argv[])
{
    double current, voltage, power, resistence;

    cout << "What is the resistence in Ohms? " << endl;
    cin >> resistence;

```

```
24      cout << "What is the voltage in Volts? " << endl;
25      cin >> voltage;
26
27      current = Current(voltage, resistence);
28      power = Power(voltage,current);
29
30      PrintResult("Current [A]", current);
31      PrintResult("Power   [W]", power);
32
33      return 0;
34  }
```

Than run:

```
$ g++ main.cpp -o electronics_calculator_1_2
$ ./electronics_calculator_1_2
```

Than we have the following output:

```
rafael:~/Desktop/CMakeLearning/1st_Module/2nd_Example$./electronics_calculator_1_2
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

1.2.4 - 3rd example:

Lets go one more step ahead and write functions in different files. This increase the modularity capacity of our code. Lets go to 3rd example folder and create the files:

```
$ cd ../3rd_Example
$ touch main.cpp Current.cpp Power.cpp Print.cpp
```

Next steps is create the code according to each file. Note that we need to declare the functions identifiers inside main.cpp, If not, the functions calls will not e recognized.

```
1  // main.cpp
2
3  #include <iostream>
4
5  using namespace std;
```

```
 6
 7 double Current(double volt, double ohms);
 8
 9 double Power(double volt, double ampere);
10
11 void PrintResult(string result_type, double result_value);
12
13
14 int main(int argc, char* argv[])
15 {
16     double current, voltage, power, resistence;
17
18     cout << "What is the resistence in Ohms? " << endl;
19     cin >> resistence;
20
21     cout << "What is the voltage in Volts? " << endl;
22     cin >> voltage;
23
24     current = Current(voltage, resistence);
25     power = Power(voltage,current);
26
27     PrintResult("Current [A]", current);
28     PrintResult("Power   [W]", power);
29
30     return 0;
31 }
```

```
1 // Current.cpp
2
3 double Current(double volt, double ohms){
4     return volt / ohms;
5 }
```

```
1 // Power.cpp
2
3 double Power(double volt, double ampere){
4     return volt * ampere;
5 }
```

```
1 //Print.cpp
2
3 #include <iostream>
```

```
4
5 void PrintResult(std::string result_type, double result_value){
6     std::cout << result_type << ": " << result_value << std::endl;
7 }
```

We need to say g++ what files are being used, at the following way. So we can run the executable file and the output is shown below.

```
$ g++ main.cpp Current.cpp Power.cpp Print.cpp -o electronics_calculator_1_3
$ ./electronics_calculator_1_3
```

Output:

```
rafael:~/Desktop/CMakeLearning/1st_Module/3rd_Example$./electronics_calculator_1_3
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

1.2.5 - 4th example - Starting with Make tool:

Last topic there was need to declare the functions names inside *main.cpp* file. This is because the compiler need to know that file is using those functions and they are in some place, so during compiling process it is found. This is not a good practice because if any change are made, other change dependencys may be needed too. Because of this, we declare the funcions names in *.h* files.

The compiler generates a binary file *.o* for each *.cpp* file. So we have *main.o*, *Current.o, Power.o,* and *Print.o*. Without *.h* files, the compiler can find where each file call each declared function during the link process. When we use *.h* it is need to attribute them to the correct files and who will do it for us is the tool **Make**.

Make is a build system, a software that automate the linking process. It is a good tool for small projects. Now we will rewrite our code with *.h* files and use the Make tool. Lets move to 4th example folder.

```
$ cd ../4th_Example
$ touch main.cpp Current.cpp Power.cpp Print.cpp Current.h Power.h Print.h
```

```
1
2 // Current.h
3
```

```
double Current(double volt, double ohms);
```

```
// Current.cpp

double Current(double volt, double ohms){
    return volt / ohms;
}
```

```
// Power.h

double Power(double volt, double ampere);
```

```
// Power.cpp

double Power(double volt, double ampere){
    return volt * ampere;
}
```

```
//Print.h

#include <iostream>

using namespace std;

void PrintResult(string result_type, double result_value);
```

```
//Print.cpp

#include <iostream>

void PrintResult(std::string result_type, double result_value){
    std::cout << result_type << ": " << result_value << std::endl;
}
```

```
//main.cpp

#include "Current.h"
```

```
#include "Power.h"
#include "Print.h"

int main(int argc, char* argv[])
{
    double current, voltage, power, resistence;

    cout << "What is the resistence in Ohms? " << endl;
    cin >> resistence;

    cout << "What is the voltage in Volts? " << endl;
    cin >> voltage;

    current = Current(voltage, resistence);
    power = Power(voltage,current);

    PrintResult("Current [A]", current);
    PrintResult("Power   [W]", power);

    return 0;
}
```

Now we need to say to the linker compiler automatization tool (Make) which files each binary file (*.o*) it deppends. First of all, we need to install it, than create and edit the recipe file (*makefile)* and run the compilation.

```
$ sudo apt-install make
$ touch makefile
```

```
electronics_calculator_1_4: main.o Current.o Power.o Print.o
	g++ main.o Current.o Power.o Print.o -o electronics_calculator_1_4

main.o: main.cpp
	g++ -c main.cpp

Current.o: Current.cpp
	g++ -c Current.cpp

Power.o: Power.cpp
	g++ -c Power.cpp

Print.o: Print.cpp
	g++ -c Print.cpp
```

So we run the make command and we will have the following output:

```
$ make
```

Output:

```
rafael:~/Desktop/CMakeLearning/1st_Module/4th_Example$./electronics_calculator_1_4
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

1.2.6 - 5th example - make tool and multiple folders and files:

Now we will try our last development scenario using Make tool. All of this were showed to understand the linking process and demonstrate how bigger projects need an andvanced tool for compile process.
Here we gonna split our projects in folders and rewrite al the files acording to the needed changes.

```
$ cd ../5th_Example
$ mkdir obj inc src
$ cd src
$ touch main.cpp Current.cpp Power.cpp Print.cpp
$ cd ../inc
$ touch Current.h Power.h Print.h
$ cd ..
$ makefile
```

The following code shows the new file configurations. In fact, only *main.cpp* includes has changes:

```
1 // Current.h
2
3 double Current(double volt, double ohms);
```

```cpp
// Current.cpp

double Current(double volt, double ohms){
    return volt / ohms;
}
```

```cpp
// Power.h

double Power(double volt, double ampere);
```

```cpp
// Power.cpp

double Power(double volt, double ampere){
    return volt * ampere;
}
```

```cpp
//Print.h

#include <iostream>

using namespace std;

void PrintResult(string result_type, double result_value);
```

```cpp
//Print.cpp

#include <iostream>

void PrintResult(std::string result_type, double result_value){
    std::cout << result_type << ": " << result_value << std::endl;
}
```

```cpp
//main.cpp

#include "../inc/Current.h"
#include "../inc/Power.h"
#include "../inc/Print.h"

int main(int argc, char* argv[])
```

```
 8 {
 9     double current, voltage, power, resistence;
10 
11     cout << "What is the resistence in Ohms? " << endl;
12     cin >> resistence;
13 
14     cout << "What is the voltage in Volts? " << endl;
15     cin >> voltage;
16 
17     current = Current(voltage, resistence);
18     power = Power(voltage,current);
19 
20     PrintResult("Current [A]", current);
21     PrintResult("Power   [W]", power);
22 
23     return 0;
24 }
```

The following makefile shows how we describe each file path configuration for linker process. Note if we type de command *make clear* the executable will be erased.

```
 1 electronics_calculator_1_5: main.o Current.o Power.o Print.o
 2    g++ obj/main.o obj/Current.o obj/Power.o obj/Print.o -o electronics_calculator_1_5
 3 
 4 main.o: src/main.cpp
 5    g++ -c src/main.cpp -o obj/main.o
 6 
 7 Current.o: src/Current.cpp
 8    g++ -c src/Current.cpp -o obj/Current.o
 9 
10 Power.o: src/Power.cpp
11    g++ -c src/Power.cpp -o obj/Power.o
12 
13 Print.o: src/Print.cpp
14    g++ -c src/Print.cpp -o obj/Print.o
15 
16 clear:
17    rm -rf electronics_calculator_1_5
```

Output:

```
rafael:~/Desktop/CMakeLearning/1st_Module/5th_Example$./electronics_calculator_1_5
What is the resistence in Ohms?
1000
```

```
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

2 - Getting started with CMake

2.1 - Installing CMake

2.1.1 - Check if CMake is installed:

The following command returns the CMake version. If not installed, will return 'not found'

```
$ cmake --version
```

If not found, go to next step:

2.1.2 - Installing CMake:

If g++ version was not found, run the following command:

```
$ sudo apt install cmake
```

Than the following command again and check if it returns the CMake version.

```
$ cmake --version
```

2.2 – Before project using CMake

The first need to know are informations about the following topics:

- File *CMakeLists.txt*
  This is a mandatory file in the project top level folder. Will keep all information of compilation.

- Directory to store the generated build files (normally called *build*).
  This directory will store all the CMake files and is a good practice keep it in the top level of the
  project.

Thiese topics are in the example of the next topic.

2.3 – CMake examples

2.3.1 – 1st Example: Preparing and understand the environment

We gonna create the files with the already knew functions and run the following command inside build folder: *cmake ..* that initializes automatically all the configurations of CMake tool. The double dots argument say that our *CMakeLists.txt* file is in the parent folder Notice that we are not compiling the program here, only preparing the environment.

```
1 // Current.h
2
3 double Current(double volt, double ohms);
```

```
1 // Current.cpp
2
3 double Current(double volt, double ohms){
4     return volt / ohms;
5 }
```

```
1 // Power.h
2
3 double Power(double volt, double ampere);
```

```
1 // Power.cpp
2
3 double Power(double volt, double ampere){
4     return volt * ampere;
```

```
5 }
```

```
1 //Print.h
2
3 #include <iostream>
4
5 using namespace std;
6
7 void PrintResult(string result_type, double result_value);
```

```
1 //Print.cpp
2
3 #include <iostream>
4
5 void PrintResult(std::string result_type, double result_value){
6     std::cout << result_type << ": " << result_value << std::endl;
7 }
```

```
 1 //main.cpp
 2
 3 #include "Current.h"
 4 #include "Power.h"
 5 #include "Print.h"
 6
 7 int main(int argc, char* argv[])
 8 {
 9     double current, voltage, power, resistence;
10
11     cout << "What is the resistence in Ohms? " << endl;
12     cin >> resistence;
13
14     cout << "What is the voltage in Volts? " << endl;
15     cin >> voltage;
S1
6      current = Current(voltage, resistence);
17     power = Power(voltage,current);
18
19     PrintResult("Current [A]", current);
20     PrintResult("Power   [W]", power);
21
22     return 0;
23
```

```
24 }
```

```
$ cd ../../2nd_Module/1st_Example
$ mkdir build
$ touch CmakeLists.txt
$ touch main.cpp Current.cpp Power.cpp Print.cpp
$ touch Current.h Power.h Print.h
$ cd build
$ cmake ..
```

If we go through *build* folder befor *cmake ..* command, we will se the following fodlers:

```
$ ls -la build/
drwxrwxr-x 3 rafael rafael  4096 jun 20 00:27 .
drwxrwxr-x 3 rafael rafael  4096 jun 20 00:27 ..
-rw-rw-r-- 1 rafael rafael 13959 jun 20 00:27 CMakeCache.txt
drwxrwxr-x 4 rafael rafael  4096 jun 20 00:27 CMakeFiles
-rw-rw-r-- 1 rafael rafael  1704 jun 20 00:27 cmake_install.cmake
-rw-rw-r-- 1 rafael rafael  4228 jun 20 00:27 Makefile
```

Makefile is used by make tool to build the project and *CMakeCache.txt* store all variables and important stuffs for next steps of build.
As our CMakeLists.txt file is empty, nothing is compiled and no one executable file is generated.

2.3.2 – 2nd Example: Generating the First Executable using CMake

On this example, we gonna generate our first executable using CMake tool. First of all we will copy all the files from the last example to this example repository and erase all build folder files. If you have any doubt about *.cpp* and *.h* project files content, check it out in the last topic.

```
$ cp -r ../* ../../2nd_Example/
$ cd ../../2nd_Module/2nd_Example/build
$ rm -rf *
```

Now lets edit our *CMakeLists.txt* file, and lets run *cmake ..* and *make* commands to generate our executable. Then lets execute it and see the output (all these steps are shown below):

```
1 # Specify the minimum CMake version installed to run this project
```

```
2
3  cmake_minimum_required(VERSION 3.16.3)
4
5  # Creates the project named Electronics_Calculator without version especification
6  project(Electronics_Calculator)
7
8  # Creates the executable electronics_calculator_2_2 using the listed files
9  add_executable(electronics_calculator_2_2
10     main.cpp
11     Current.cpp
12     Power.cpp
13     Print.cpp
   )
```

```
$ cmake ..
$ make
$ ./electronics_calculator_2_2
```

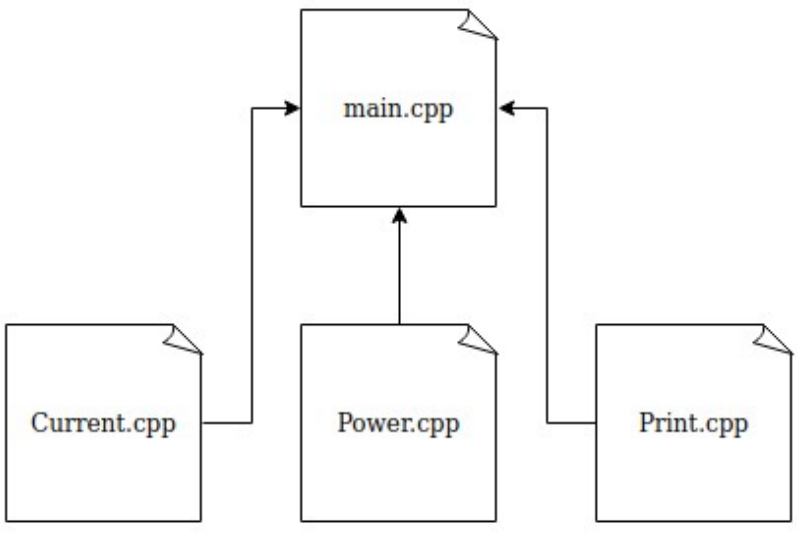
Output:

```
rafael:~/Desktop/CMakeLearning/2nd_Module/5th_Example$./electronics_calculator_2_2
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

Although it works fine, there is no sense of hierarchy here. In big projects this could be a problem. The solution will me shown in the next example, where we will create the first library.

2.3.2 – 2nd Example: Generating the First Library using CMake

If we are working in a big project, the solution of last example could not be the best practice because of the



Next step it will be created two libraries according to their functionality: *OhmsLaw* and *PrintOnScreen*. This make our project more portable because we can use these libraries in other projects. The following commands copy the files from 2nd Example (check in 2.3.1 topic their content).

```
$ cp -r ../* ../../3rd_Example/
$ cd ../../2nd_Module/3rd_Example/build
$ rm -rf *
```

The following code creates the libraries and link them to the executable. Running *cmake ..* and *make* commands will generate the output as we can see below:

```
 1 # Specify the minimum CMake version installed to run this project
 2 cmake_minimum_required(VERSION 3.16.3)
 3
 4 # Creates the project named Electronics_Calculator without version especification
 5 project(Electronics_Calculator)
 6
 7 # Creates library with ohms law calculator purpose
 8 add_library(OhmsLaw
 9     Current.cpp
10     Power.cpp
11 )
12
13 # Creates library with Printing purpose
14 add_library(PrintOnScreen
15     Print.cpp
16 )
17
18 # Creates the executable electronics_calculator_2_3 using the listed files
19 add_executable(electronics_calculator_2_3
20     main.cpp
21 )
22
```

```
23 # Link the executable to the created libraries
24 target_link_libraries(electronics_calculator_2_3
25     OhmsLaw
26     PrintOnScreen
27 )
```

Output:

```
rafael:~/Desktop/CMakeLearning/2nd_Module/5th_Example$./electronics_calculator_2_3
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

The libraries are generated and saved on *build* folder. We can create more than 1 target if the dependencies are kept.

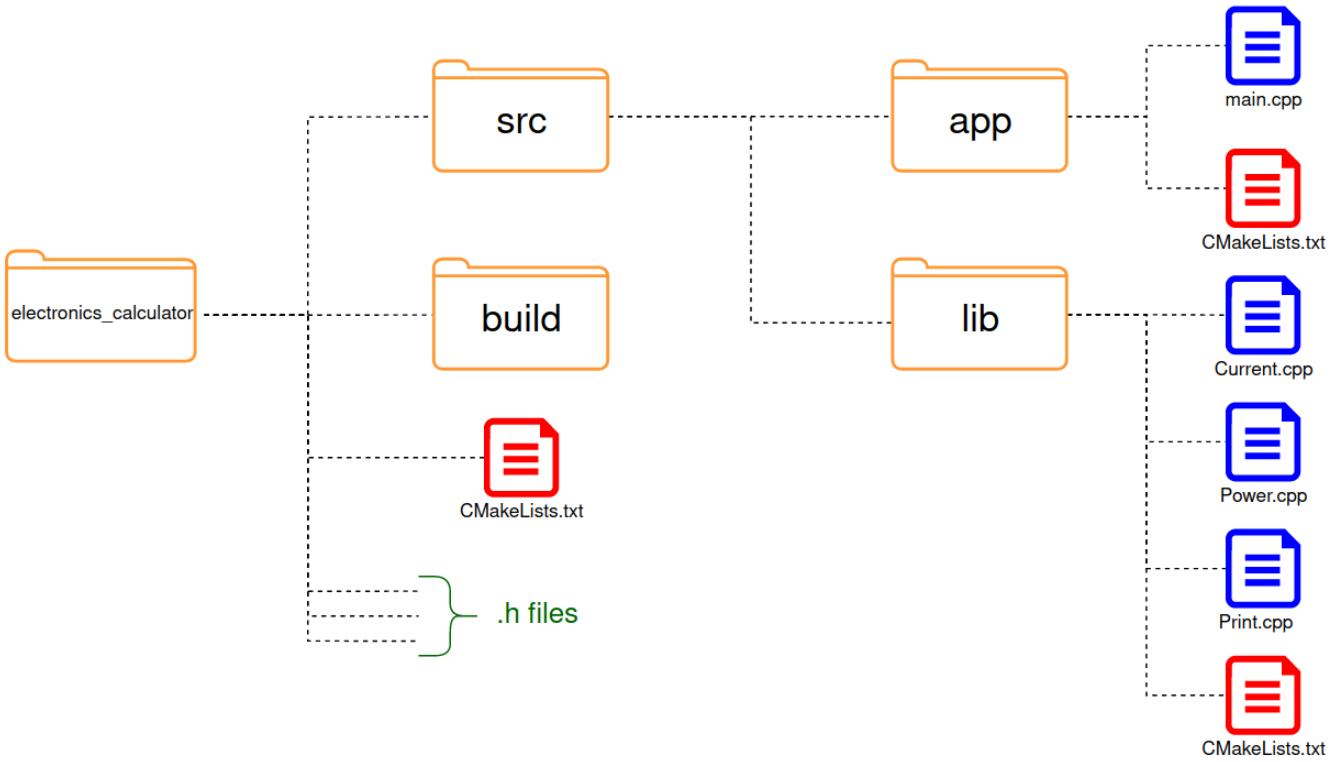
3 – Managing projects files and folders

3.1 - Managing the project repository

When the work project is too big (most of real and profissional cases) it does not make sense storage all the files in the same folder. At this sectioon will be displayed a good directory practice for the Electronics Calculator project. Of course each project is unique. Smaller or specific projects could have diferent aproaches.

3.1.1 - 1st Example - Managing directories - .cpp files

First of all lets understand the best way of source files managment. The next picture show an idea of a good practice directory organization for the Electronics Calculator project.



In the *CMakeLists.txt* of root level, it is used the *CMake* command *include_subdirectory()*. It is responsable to include the *CMakeLists.txt* files from *app* and *lib* folders in the root *CMakeLists.txt.* At this way, all code writed in the low level files will be read by top level file. You will notice that way, if a change is needed only the relative files and folders will be changed as the targets files (executable or librarys) will be generated in your relative output folder. The whole project is shown below:

```
$ cd ../../3rd_Module/1st_Example/
$ cp ../../2nd_Module/3rd_Example/* .
$ mkdir build
$ mkdir src
$ mkdir src/app src/lib
$ touch src/lib/CMakeLists.txt
$ touch src/app/CMakeLists.txt
$ mv Current.cpp Power.cpp Print.cpp src/lib/
$ mv main.cpp src/app/
$ cd build
```

```
1 // Current.h
2
3 double Current(double volt, double ohms);
```

```
1 // Power.h
2
3 double Power(double volt, double ampere);
```

```
1 // Print.h
2
3 #include <iostream>
4
5 using namespace std;
6
7 void PrintResult(string result_type, double result_value);
```

```
 1 # CMakeLists.txt (root)
 2
 3 cmake_minimum_required(VERSION 3.16.3)
 4
 5 # Creates a project Named Calculator_Project with 1.0.0 version
 6 project(Electronics_Calculator VERSION 3.1.0)
 7
 8 # CMake should go to these directories and find another CMakeLists.txt
 9 add_subdirectory(src/lib)
10
11 # CMake should go to these directories and find another CMakeLists.txt
12 add_subdirectory(src/app)
13
14 # Links the executable calculator ( defined at add_executable()
15 target_link_libraries(
16     electronics_calculator_3_1
17     OhmsLaw
18     PrintOnScreen
19 )
```

```
1 // Current.cpp
2
3 double Current(double volt, double ohms){
4     return volt / ohms;
5 }
```

```
1 // Power.cpp
```

```
2
3 double Power(double volt, double ampere){
4     return volt * ampere;
5 }
```

```
1 // Print.cpp
2
3 #include <iostream>
4
5 void PrintResult(std::string result_type, double result_value){
6     std::cout << result_type << ": " << result_value << std::endl;
7 }
```

```
 1 # CMakeLists.txt (lib)
 2
 3 # Creates a library with Ohms Law functions named OhmsLaw
 4 add_library(OhmsLaw
 5     Current.cpp
 6     Power.cpp
 7 )
 8
 9 # Creates a library with print functions named Print
10 add_library(PrintOnScreen
11     Print.cpp
12 )
```

```
 1 #include "../../Current.h"
 2 #include "../../Power.h"
 3 #include "../../Print.h"
 4
 5 int main(int argc, char* argv[])
 6 {
 7     double current, voltage, power, resistence;
 8
 9     cout << "What is the resistence in Ohms? " << endl;
10     cin >> resistence;
11
12     cout << "What is the voltage in Volts? " << endl;
13     cin >> voltage;
14
15     current = Current(voltage, resistence);
```

```
16      power = Power(voltage,current);
17
18      PrintResult("Current [A]", current);
19      PrintResult("Power   [W]", power);
20
21      return 0;
22  }
```

```
 1  # CMakeLists.txt (app)
 2
 3  # Creates the executable electronics_calculator_2_3 using the listed files
 4  add_executable(electronics_calculator_3_1
 5      main.cpp
 6  )
 7
 8  # Link the executable to the created libraries
 9  target_link_libraries(electronics_calculator_3_1
10      OhmsLaw
11      PrintOnScreen
12  )
```

Note: as the *add_executable()* to our executable target (*electronics_calculator_3_1*) is inside a subfolder, the executable will be generated there, but you can choose the local where the executable file will be generated by using the following function. Here, we are saying to generate at *build* top level folder.

```
set_target_properties(
    electronics_calculator_3_1
    PROPERTIES RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}
)
```

Because of it, to run the executable we need to execute directly in the app folder output (after rum *cmake ..).*

```
$ cmake ..
$ make
$ ./src/app/electronics_calculator_3_1
```

Output:

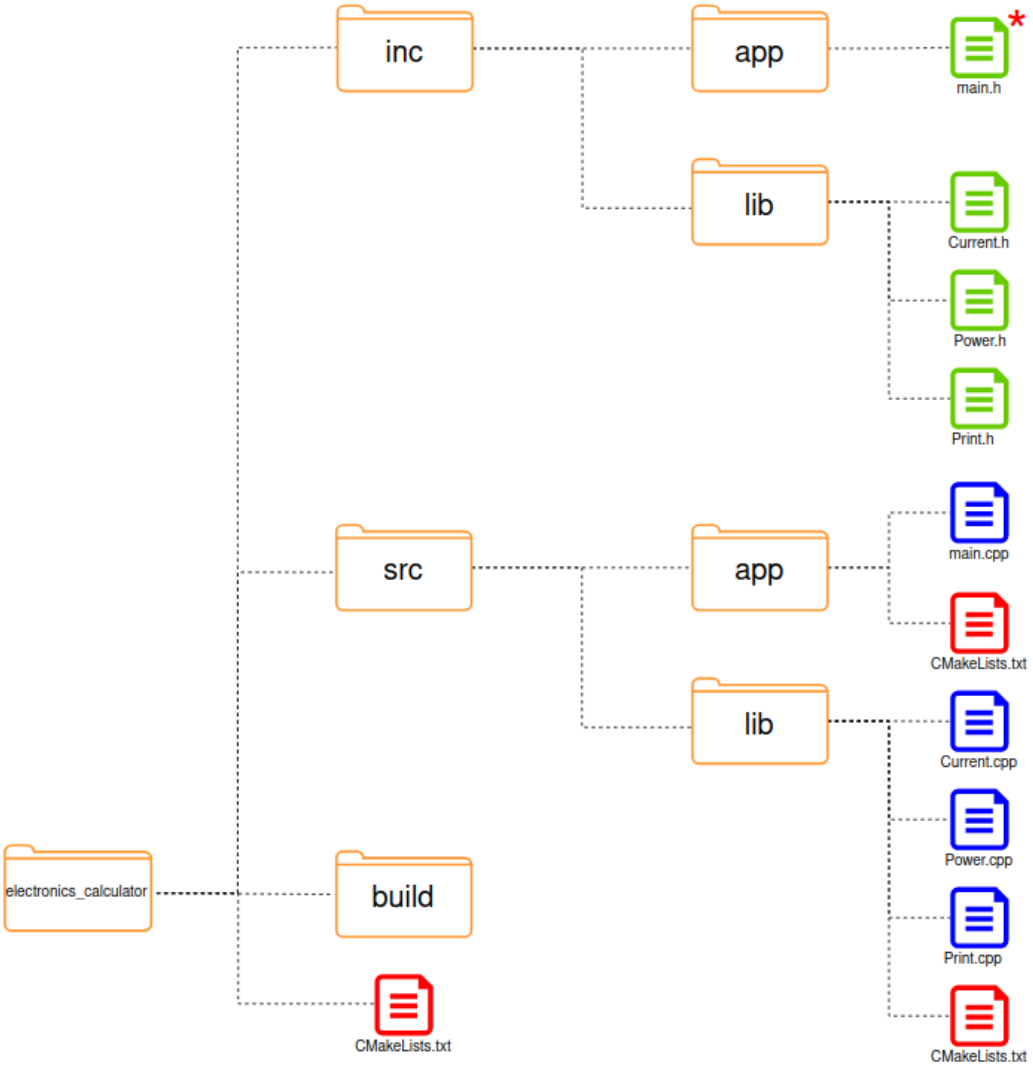
```
rafael:~/Desktop/CMakeLearning/3rd_Module/1st_Example$./electronics_calculator_3_1
What is the resistence in Ohms?
1000
What is the voltage in Volts?
```

```
200
Current [A]: 0.2
Power   [W]: 40
```

3.1.2 - 2nd Example - Managing directories - .h files

You might want to know about the header files (*.h).* Of course they need to be kept inside a logical folder according to the whole project. This make the maintence easier. Here is the project of 1st example refreshed w.*h* files in the new folders:

```
$ cd ../../2nd_Example/
$ cp -r ../1st_Example/* .
$ rm -rf build/ *
$ mkdir inc
$ mkdir inc/app inc/lib
```



* *main.h* is not applicable here but it was kept in the picture for best understanding purpose.

Another good practice is include the declaration of header files inside their program file (for exemple: Current.cpp includes Curent.h) using *#include* directive. This inclusion avoid any different kinds of inconsistences during function calls and declarations só it will be done in this example, as we can see bellow:

TODO: Insert codes

3.1.3 - 3rd Example - Managing directories – CMake take care of .h path files

If *.cpp* and *.h* files of a process are both in the same file, CMake will easly find it during the build process, else, if the *#include* directive does not point the exact *.h* path file, the process will return an error (as in the last example).

This diffuses the development because in case of any directory change, it requires all *#includes* directives refresh. It is possible CMake take care of the process of path pointing using:

```
target_include_directories(<target> <scope> <dir1> <dir2. ...)
```

<target> : the library or executable that will use the directory
<scope>: PRIVATE: directory will be visible only in the folder of the target
INTERFACE: directory will be visible only in the folder of the target
PUBLIC: directory will be visible only in the folder of the target
<dirn>: Directory that will be setted in the target property.

The directive *target_include_directory()* sets the following given target property: INTERFACE_INCLUDE_DIRECTORIES, so any dependent file of this target will inherit the value of this property. This is the reason that in the following example, we may include the libraries *.h* files without insert the *target_include_directory()* in the root *CMakeLists.txt.* Once uppon a time the target *electronics_calculator_3_3* is inherting the INTERFACE_INCLUDE_DIRECTORIES from it dependencies (*OhmsLaw* and *PrintOnScreen*) through the directive *target_link_libraries().*

```
$ cd ../../3rd_Example/
$ cp -r ../2nd_Example/* .
$ rm -rf build/ *
```

```
1 // Current.h
2
3 double Current(double volt, double ohms);
```

```
1 // Power.h
2
3 double Power(double volt, double ampere);
```

```
1 //Print.h
2
3 #include <iostream>
4
5 using namespace std;
6
7 void PrintResult(string result_type, double result_value);
```

```cpp
#include "lib/Current.h"
#include "lib/Power.h"
#include "lib/Print.h"

int main(int argc, char* argv[])
{
    double current, voltage, power, resistence;

    cout << "What is the resistence in Ohms? " << endl;
    cin >> resistence;

    cout << "What is the voltage in Volts? " << endl;
    cin >> voltage;

    current = Current(voltage, resistence);
    power = Power(voltage,current);

    PrintResult("Current [A]", current);
    PrintResult("Power   [W]", power);

    return 0;
}
```

```
# CMakeLists.txt (app)

# Creates the executable electronics_calculator_3_2 using the listed files
add_executable(electronics_calculator_3_3
    main.cpp
)

# Link the executable to the created libraries
target_link_libraries(electronics_calculator_3_3
    OhmsLaw
    PrintOnScreen
)
```

```cpp
// Current.cpp

#include "lib/Current.h"

double Current(double volt, double ohms){
    return volt / ohms;
}
```

```cpp
// Power.cpp

#include "lib/Power.h"

double Power(double volt, double ampere){
    return volt * ampere;
}
```

```cpp
//Print.cpp

#include "lib/Print.h"

void PrintResult(std::string result_type, double result_value){
    std::cout << result_type << ": " << result_value << std::endl;
}
```

```
# CMakeLists.txt (lib)

# Creates a library with Ohms Law functions named OhmsLaw
add_library(OhmsLaw
    Current.cpp
    Power.cpp
)

# Creates a library with print functions named Print
add_library(PrintOnScreen
    Print.cpp
)

target_include_directories(OhmsLaw PUBLIC ../../inc)
target_include_directories(PrintOnScreen PUBLIC ../../inc)
```

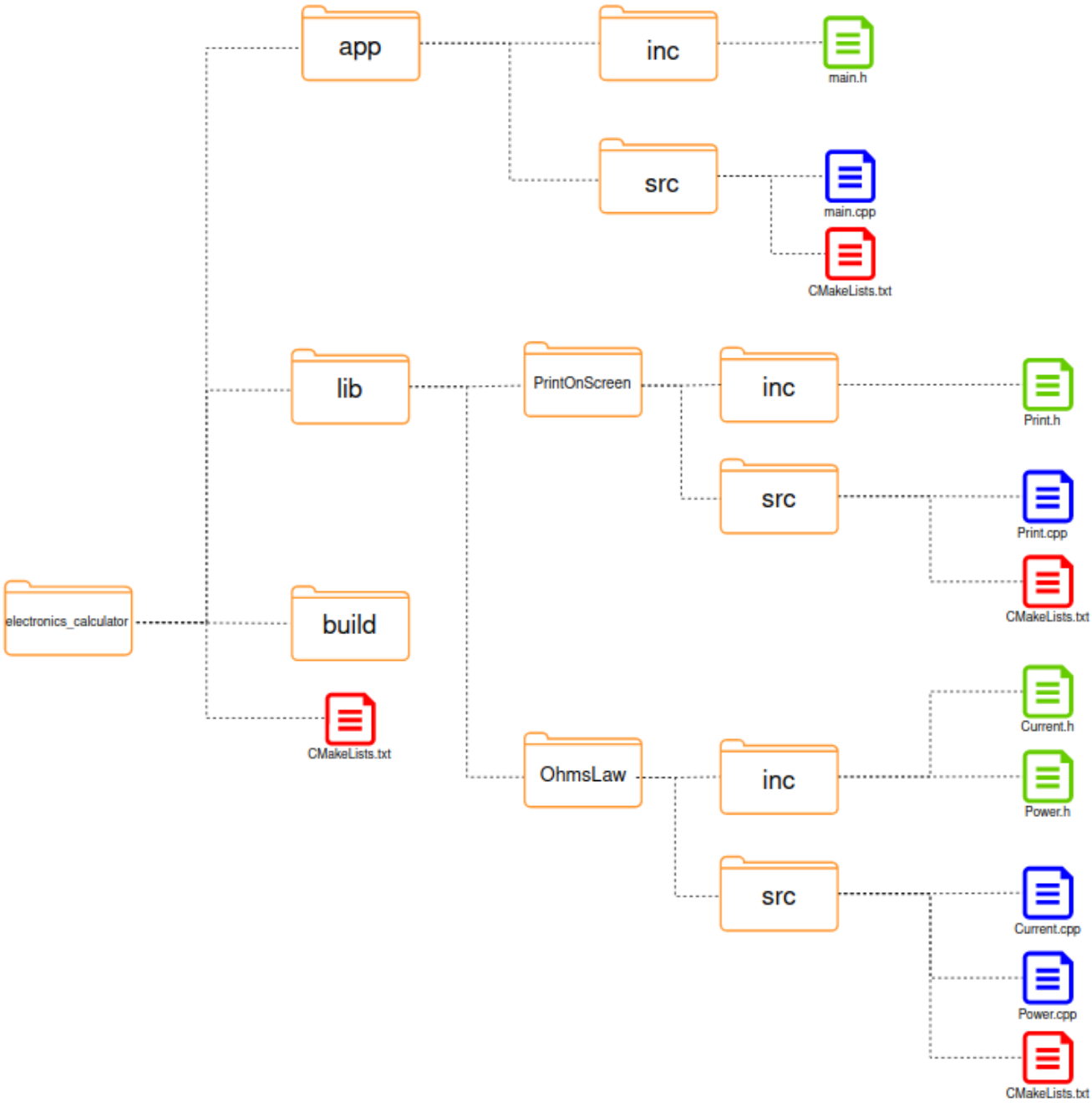
```
# CMakeLists.txt (root)

cmake_minimum_required(VERSION 3.16.3)

# Creates a project Named Calculator_Project with 1.0.0 version
project(Electronics_Calculator VERSION 3.3.0)

# CMake should go to these directories and find another CMakeLists.txt
```

```
 9 add_subdirectory(src/lib)
10
11 # CMake should go to these directories and find another CMakeLists.txt
12 add_subdirectory(src/app)
13
14 # Links the executable electronics_calculator_3_3  ( defined at add_executable()
15 target_link_libraries(
16     electronics_calculator_3_3
17     OhmsLaw
18     PrintOnScreen
19 )
```

3.1.4 - 4th Example - Another aproach of progect managment

Instead of sort de folders by includes and source files, we may do it by app vs libs apps. This allow us to have each library with its specific file. All the code is shown bellow after all commands for directory configuration:

```
$ cd ../../4th_Example/
$ mkdir build app app/inc app/src lib lib/PrintOnScreen lib/PrintOnScreen/inc
$ mkdir lib/PrintOnScreen/src lib/OhmsLaw lib/OhmsLaw/inc lib/OhmsLaw/src
$ cp ../3rd_Example/inc/lib/Print.h lib/PrintOnScreen/inc/
$ cp ../3rd_Example/inc/lib/Current.h lib/OhmsLaw/inc/
$ cp ../3rd_Example/inc/lib/Power.h lib/OhmsLaw/inc/
$ cp ../3rd_Example/src/lib/Print.cpp lib/PrintOnScreen/src/
$ cp ../3rd_Example/src/lib/Current.cpp lib/OhmsLaw/src/
$ cp ../3rd_Example/src/lib/Power.cpp lib/OhmsLaw/src/
$ cp ../3rd_Example/src/app/main.cpp app/src
$ touch CMakeLists.txt lib/OhmsLaw/src/CMakeLists.txt
$ touch lib/PrintOnScreen/src/CMakeLists.txt touch app/src/CMakeLists.txt
$ touch app/inc/main.h
```




```
1 // main.h
2
3 #include "Current.h"
4 #include "Power.h"
5 #include "Print.h"
```

```
1 // main.cpp
2 #include "main.h"
3
4 int main(int argc, char* argv[])
```

```
 5 {
 6     double current, voltage, power, resistence;
 7
 8     cout << "What is the resistence in Ohms? " << endl;
 9     cin >> resistence;
10
11     cout << "What is the voltage in Volts? " << endl;
12     cin >> voltage;
13
14     current = Current(voltage, resistence);
15     power = Power(voltage,current);
16
17     PrintResult("Current [A]", current);
18     PrintResult("Power   [W]", power);
19
20     return 0;
21 }
```

```
 1 # CMakeLists.txt (app)
 2
 3 # Creates the executable electronics_calculator_3_4 that uses main.cpp file
 4 add_executable(
 5     electronics_calculator_3_4
 6     main.cpp
 7 )
 8
 9 # Set app/include as private because only will be used here
10 target_include_directories(electronics_calculator_3_4 PRIVATE ../inc)
```

```
1 // Current.h
2
3 double Current(double volt, double ohms);
```

```
1 // Current.cpp
2
3 #include "Current.h"
4
5 double Current(double volt, double ohms){
6     return volt / ohms;
7 }
```

```
// Power.h

double Power(double volt, double ampere);
```

```
// Power.cpp

#include "Power.h"

double Power(double volt, double ampere){
    return volt * ampere;
}
```

```
# CMakeLists.txt (OhmsLaw)

# Creates a library with Ohms Law functions named OhmsLaw
add_library(OhmsLaw
    Current.cpp
    Power.cpp
)

# Set lib/OhmsLaw/inc as PUBLIC because will be used here and in the main.cpp
target_include_directories(OhmsLaw PUBLIC ../inc)
```

```
//Print.h

#include <iostream>

using namespace std;

void PrintResult(string result_type, double result_value);
```

```
//Print.cpp

#include "Print.h"

void PrintResult(std::string result_type, double result_value){
    std::cout << result_type << ": " << result_value << std::endl;
}
```

```
# CMakeLists.txt (PrintOnScreen)

# Creates a library with print functions named Print
```

```
4 add_library(PrintOnScreen
5     Print.cpp
6 )
7
8 # Set lib/PrintOnScreen/inc as PUBLIC because will be used here and in the main.cpp
9 target_include_directories(PrintOnScreen PUBLIC ../inc)
```

```
 1 # CMakeLists.txt (root)
 2
 3 cmake_minimum_required(VERSION 3.16.3)
 4
 5 # Creates a project Named Calculator_Project with 1.0.0 version
 6 project(Electronics_Calculator VERSION 3.4.0)
 7
 8 # CMake should go to these directories and find another CMakeLists.txt
 9 add_subdirectory(app/src)
10 add_subdirectory(lib/OhmsLaw/src)
11 add_subdirectory(lib/PrintOnScreen/src)
12
13 # Links the executable electronics_calculator_3_4 ( defined at add_executable() )
14 # to OhmsLaw and PrintOnScreen libraries
15 target_link_libraries(
16     electronics_calculator_3_4
17     OhmsLaw
18     PrintOnScreen
19 )
```

```
$ cd build
$ cmake ..
$ make
$ ,/app/src/electronics_calculator_3_4
```

Output:

```
rafael:~/Desktop/CMakeLearning/3rd_Module/4th_Example$./electronics_calculator_3_4
What is the resistence in Ohms?
1000
What is the voltage in Volts?
200
Current [A]: 0.2
Power   [W]: 40
```

4 Data manipulation

4.1.1 - 1st Example - Running CMake in script mode using cmake -P

The script command run the CMakeLists.txt withou generate any build file. At this way, if any build function is used inside CMakeLists.txt, the *cmake -P <fileName>* will return an error.

```
$ cd ../../../4th_Module/1st_Example
$ touch CMakeLists.txt lib/OhmsLaw/src/CMakeListsNG.txt
$ touch CMakeLists.txt lib/OhmsLaw/src/CMakeListsOK.txt
```

```
1 # CMakeListsNG.txt
2
3 cmake_minimum_required(VERSION 3.16.3)
4
5 project(PojectName VERSION 4.1.0)
```

```
1 # CMakeListsOK.txt
2
3 cmake_minimum_required(VERSION 3.16.3)
```

Then run:

```
$ cmake -P CMakeListsNG.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/1st_Example$ cmake -P CMakeListsNG.txt
CMake Error at CMakeLists.txt:4 (project):
  project command is not scriptable
```

Then run:

```
$ cmake -P CMakeListsOK.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/1st_Example$ cmake -P CMakeListsOK.txt
rafael:~/Desktop/CMakeLearning/4th_Module/1st_Example$
```

4.1.2 - 2nd Example - Running CMake in script mode using an easier way

- Inside the *CMakeLists.txt* file, run the command ***which cmake*** in the terminal (it will give the location of the cmake executable - my case: */usr/bin/cmake*);

- Open the CMakeListsFile and past the following command: ***#!/usr/bin/cmake -P***

- Give the execution permission to the *CMakeLists.txt* file running the following command: ***chmod +x CMakeLists.txt***

- Now you can either use ***./CMakeLists.txt*** or ***cmake -P CMakeLists.txt*** to run the *CMakeLists.txt* file in the script mode.

```
$ cd ../2nd_Example
$ touch CMakeLists.txt
```

```
1 #!/usr/bin/cmake -P
2
3 cmake_minimum_required(VERSION 3.16.3)
```

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/2nd_Example$ cmake -P CMakeLists.txt
rafael:~/Desktop/CMakeLearning/4th_Module/1st_Example$
```

Or run:

```
$ ./CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/2nd_Example$ ./CMakeLists.txt
rafael:~/Desktop/CMakeLearning/4th_Module/2nd_Example$
```

4.2 - 3rd Example - Printing messages on the terminal

The *message()* command it is used to print messages on the terminal. As we can see bellow, this command expects a mode of display (STATUS, DEBUG, WARNING or FATAL_ERROR - check all types here). In this example you will see some example messages:

```
$ cd ../3rd_Example
$ touch CMakeLists.txt
```

```
1 cmake_minimum_required(VERSION 3.16.3)
2
3 message("Hello world from 0 Rafael Del Pino")
4 message(STATUS "Hello world 1 from Rafael Del Pino")
5 message(WARNING "Hello world 2 from Rafael Del Pino")
6 message(VERBOSE "Hello world 3 from Rafael Del Pino")
7 message(SEND_ERROR "Hello world 4 from Rafael Del Pino")
8 message(FATAL_ERROR "Hello world 5 from Rafael Del Pino")
9 message("This will not be printed!")
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/3rd_Example$ ./CMakeLists.txt
Hello world from 0 Rafael Del Pino
-- Hello world 1 from Rafael Del Pino
CMake Warning at CMakeLists.txt:7 (message):
  Hello world 2 from Rafael Del Pino


CMake Error at CMakeLists.txt:9 (message):
  Hello world 4 from Rafael Del Pino


CMake Error at CMakeLists.txt:10 (message):
  Hello world 5 from Rafael Del Pino
```

4.4 - 4th Example - Variables and print variables on terminal

The following command is used to create and set a variable value. Each field is defined in the *set()* function bellow:

```
set(<variable_name> <variable_value>)
```

All CMake variables are of string type and for use them you need to access it by using the following way: *${variable_name}.* When using *set()* function, there are two possobilities of initialization:

```
# NAME is a string with "Rafael Del Pino" value
set (NAME "Rafael Del Pino")

# NAME is a list with [Rafael][Del][Pino] string items
set (NAME Rafael Del Pino)
```

A CMake list is a semicolon (;) separated items. A string may be considered a list of one unique item of string type. Lists can be operated using *list()* command and strings may be operated using *string()* command. The following example will show a single variable utilization application:

```
$ cd ../3rd_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (NAME Rafael)
 4 set (OCCUPATION Engineer)
 5 message ("My name is ${NAME} and I am an ${OCCUPATION}")
 6
 7 # NAME is a string: "Rafael Del Pino" value
 8 set (NAME "Rafael Del Pino")
 9 message ("My name is ${NAME} and I am an ${OCCUPATION}")
10
11 # NAME is a list: [Rafael][Del][Pino] string items
12 set (NAME Rafael Del Pino)
13 message ("My name is ${NAME} and I am an ${OCCUPATION}")
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/4th_Example$ cmake -P CMakeLists.txt
My name is Rafael and I am an Engineer
My name is Rafael Del Pino and I am an Engineer
My name is Rafael;Del;Pino and I am an Engineer
```

4.5 - 4th Example - Different ways to use variables

The most common variable usage: set a variable and print it on screen:

```
set (NAME Rafael)
message (${NAME})
```

As we know, variables are list of strings, so we can use the value of variable to reference the name of a seccond one, and print it:

```
1 set (NAME Rafael)
2 set (Rafael Pino)
3
4 message (${NAME})    # Print Rafael
5 message (${${NAME}}) # Print Pino
```

This also works:

```
1 set (NAME "Rafael")
2 set (Rafael Pino)
3
4 message (${NAME})    # Print Rafael
5 message (${${NAME}}) # Print Pino
```

It is possible concatenate variable names and get its value, as shown bellow:

```
1 set (NAME Rafael)
2 set (Rafael Pino)
3 set (NAMERafaelPino FullName)
4
5 # Print value of value inside NAME+${NAME}+${${NAME}} (same as ${NAMERafaelPino} )
6 message (${NAME${NAME}${${NAME}}})
```

A full usage variable example was made to best understand:

```
$ cd ../4th_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 message ("1st demonstration:")    # Print demonstration ID
 4 set (NAME Rafael)                 # Set Variable NAME with value Rafael
 5 message (${NAME})                 # Print value inside NAME
 6
 7 message ("")
 8
 9 message ("2nd demonstration:")    # Print demonstration ID
10 set (NAME "Rafael")                 # Set Variable NAME with value Rafael
11 set (Rafael Pino)                 # Set Variable Rafael with value Pino
12 message (${NAME})                 # Print value inside NAME
13 message (${${NAME}})              # Print value of value inside NAME (same as ${Rafael
14 message (NAME ${NAME} ${${NAME}}) # Print a list of string values
15 message (NAME${NAME}${${NAME}})   # Print a list of string values
16 message ("NAME ${NAME} ${${NAME}}") # Print one string with values
17
18 message ("")
19
20 message ("3rd demonstration:")    # Print demonstration ID
21 set (NAME Rafael)                 # Set Variable NAME with value Rafael
22 set (Rafael Pino)                 # Set Variable Rafael with value Pino
23 set (NAMERafaelPino FullName)     # Set Variable NAMERafaelPino with value FullName
24
25 # Print value of value inside NAME+${NAME}+${${NAME}} (same as ${NAMERafaelPino} )
26 message (${NAME${NAME}${${NAME}}})
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/4th_Example$ cmake -P CMakeLists.txt
1st demonstration:
Rafael
```

```
2nd demonstration:
Rafael
Pino
NAMERafaelPino
NAMERafaelPino
NAME Rafael Pino

3rd demonstration:
FullName
```

4.6.1 - 5th Example - Lists declaration

There are different ways of initialize a variable (as we know, variable is a **list** of strings). All of the above codes will output the same result (all of them are VARIABLE list). The code is sotred on 5th_Example folder CMakeLists.txt file:

```
$ cd ../5th_Example
$ touch CMakeLists.txt
```

```
1 set (VARIABLE_1 11 22 33 44) # VARIABLE = 11;22;33;44
2 message (${VARIABLE_1})      # Print the list values
3 message ("${VARIABLE_1}")    # Put togheter the list items and print
```

```
1 set (VARIABLE_2 11;22;33;44) # VARIABLE = 11;22;33;44
2 message (${VARIABLE_2})      # Print the list values
3 message ("${VARIABLE_2}")    # Put togheter the list items and print
```

```
1 set (VARIABLE_3 "11" "22" "33" "44") # VARIABLE = 11;22;33;44
2 message (${VARIABLE_3})      # Print the list values
3 message ("${VARIABLE_3}")    # Put togheter the list items and print
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/5th_Example$ cmake -P CMakeLists.txt
```

```
11223344
11;22;33;44
```

We have a couple of frequently list operations:

list(<subcommand> <name_of_list> ...)
list(<subcommand> <name_of_list> ... <return_variable>)

<subcommand> : the list operation: APPEND, INSERT, FILTER, GET, JOIN (most used)
<name_of_list>: Target list to be operated
<return_variable> : If the operation returns a value, the last argument will be the output result.

4.6.2 - 6th Example - Lists operation (without return parameter)

In the 6th example file a variable (as we know, a list of strings) is being created and some without return parameter list operation example are used to best understanding:

```
$ cd ../6th_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (VARIABLE a b c;d "e;f" 16.09 "Hello world") # VARIABLE = 11;22;33;44
 4
 5 message (${VARIABLE})      # Print the list values
 6 message ("${VARIABLE}\n")  # Join the list items and print
 7
 8 list (APPEND VARIABLE 01.08 EFDP)
 9
10 message (${VARIABLE})      # Print the list values
11 message ("${VARIABLE}\n")  # Join the list items and print
12
13 list (REMOVE_AT VARIABLE 2 -2) # Remove 3rd (0+2) and last-second element (-2)
14
15 message (${VARIABLE})      # Print the list values
16 message ("${VARIABLE}\n")  # Join the list items and print
17
18 list (REMOVE_ITEM VARIABLE 16.09) # Remove 16.09 value
19
20 message (${VARIABLE})      # Print the list values
21 message ("${VARIABLE}\n")  # Join the list items and print
22
23 list (INSERT VARIABLE 2 EFDP) # Insert value at 0+2 position
24
25 message (${VARIABLE})      # Print the list values
```

```
26 message ("${VARIABLE}\n")  # Join the list items and print
27
28 list (REVERSE VARIABLE )   # Mirror list order
29
30 message (${VARIABLE})      # Print the list values
31 message ("${VARIABLE}\n")  # Join the list items and print
32
33 list (REMOVE_DUPLICATES VARIABLE) # Remove duplicate values
34
35 message (${VARIABLE})      # Print the list values
36 message ("${VARIABLE}\n")  # Join the list items and print
37
38 list (SORT VARIABLE)       # Sort list values
39
40 message (${VARIABLE})      # Print the list values
41 message ("${VARIABLE}\n")    # Join the list items and print
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/6th_Example$ cmake -P CMakeLists.txt
abcdef16.09Hello world
a;b;c;d;e;f;16.09;Hello world

abcdef16.09Hello world01.08EFDP
a;b;c;d;e;f;16.09;Hello world;01.08;EFDP

abdef16.09Hello worldEFDP
a;b;d;e;f;16.09;Hello world;EFDP

abdefHello wordlEFDP
a;b;d;e;f;Hello world;EFDP

abEFDPdefHello worldEFDP
a;b;EFDP;d;e;f;Hello world;EFDP

EFDPHello worldfedEFDPba
EFDP;Hello world;f;e;d;EFDP;b;a

EFDPHello worldfedba
EFDP;Hello world;f;e;d;b;a
```

```
EFDPHello wordlabdef
EFDP;Hello wordl;a;b;d;e;f
```

4.6.3 - 7th Example - Lists operation (with return parameter)

In the next example file a variable (as we know, a list of strings) is being created with a return parameter list operation:

```
$ cd ../7th_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (VARIABLE EFDP;"Hello world";f;e;d;b;a)
 4
 5 message (${VARIABLE})      # Print the list values
 6 message ("${VARIABLE}\n")  # Join the list items and print
 7
 8 list (GET VARIABLE 2 5 6 list_get)       # list_get = VARIABLE index 2;5;6
 9 list (JOIN VARIABLE --- list_join)       # add "---" between each list item
10 list (SUBLIST VARIABLE 2 3 list_sublist) # list_sublist = VARIABLE index 2 to 4
11 list (FIND VARIABLE f variable_find)     # find index of f
12 list (LENGTH VARIABLE variable_lengh)    # show variable lengh
13
14 # Output printing
15 message("      list_get: " ${list_get})
16 message("     list_join: " ${list_join})
17 message("  list_sublist: " ${list_sublist})
18 message(" variable_find: " ${variable_find})
19 message("variable_lengh: " ${variable_lengh})
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/6th_Example$ cmake -P CMakeLists.txt
EFDP"Hello world"fedba
EFDP;"Hello world";f;e;d;b;a

      list_get: fba
     list_join: EFDP---"Hello world"---f---e---d---b---a
  list_sublist: fed
```

```
 variable_find: 2
variable_lengh: 7
```

4.7 - 8th Example - Strings

Similar to *list* commands, the *string()* command also offers a lot of subcommands and we gonna see their operation in the next *CMakeLists.txt* example. The following commands allow find and replace the substring of any string. If substring is not found, it will return -1 value.

```
$ cd ../8th_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (VARIABLE "Rafael Oliveira Del Pino 31")
 4
 5 message (${VARIABLE})
 6 message ("${VARIABLE}\n")
 7
 8 # With return parameter operations
 9 # Find index of string "Oliveira" and stores on variable_find
10 string (FIND ${VARIABLE} "Oliveira" variable_find)
11
12 # Replace "Rafael" by "Rafa" and stores replaced on variable_replace
13 string (REPLACE "Rafael" "Rafa" variable_replace ${VARIABLE})
14
15 # Replace all letters to lower case and stores replaced on variable_toLower
16 string (TOLOWER ${VARIABLE} variable_toLower)
17
18 # Replace all letters to lower case and stores replaced on variable_toUpper
19 string (TOUPPER ${VARIABLE} variable_toUpper)
20
21 # Compare strings and returns 1 if equal or 0 if not equal
22 string (COMPARE EQUAL ${variable_toUpper}
23         "RAFAEL OLIVEIRA DEL PINO 31" variable_compareEqual)
24
25 message("variable_find: " ${variable_find})
26 message("variable_replace: " ${variable_replace})
27 message("variable_toLower: " ${variable_toLower})
28 message("variable_toUpper: " ${variable_toUpper})
29 message("variable_compareEqual: " ${variable_compareEqual})
30
31 # without return parameter operations
32 # Insert string before
```

```
33 string (PREPEND VARIABLE "Name/Age: ")
34 message (${VARIABLE})
35
36 # Insert string after
37 string (APPEND VARIABLE " years old.")
38 message (${VARIABLE})
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/4th_Module/8th_Example$ cmake -P CMakeLists.txt
Rafael Oliveira Del Pino 31

variable_find: 7
variable_replace: Rafa Oliveira Del Pino 31
variable_toLower: rafael oliveira del pino 31
variable_toUpper: RAFAEL OLIVEIRA DEL PINO 31
variable_compareEqual: 1
Name/Age: Rafael Oliveira Del Pino 31
Name/Age: Rafael Oliveira Del Pino 31 years old.
```

5 Flow control commands

5.1 - 1st Example - If-Else

If command will execute the code inside the if() - endif() body one time (see the next section – loop comands – to execute it repeatedly) when the condition inside parentheses is **true.**

When we use *if* command, the following values are defined as **true**:
1, ON, YES, TRUE, Y, a non-zero number.

When we use *if* command, the following values are defined as **false**:
0, OFF, NO, FALSE, N, IGNORE, NOTFOUND, empty string, string ending with -NOTFOUND

Here we will show some conditions arguments. All the types can be checked in the documentation page.

```
$ cd ../../../5th_Module/1st_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set(VAR OFF)
 4 set(VAR1 VAR)
 5
 6 # Test the value pointed by variable assigned to VAR1
 7 if (${VAR1})
 8     message ("Block IF executed")
 9 else()
10     message ("Block ELSE executed")
11 endif()
12
13
14 set (Name Rafael)
15
16 # Test if Name is defined
17 if (DEFINED Name)
18     message ("Name is DEFINED")
19 else()
20     message ("Name is not DEFINED")
21 endif()
22
23 # Test if Age is defined
24 if (DEFINED Age)
```

```
25     message ("Age is DEFINED")
26 else()
27     message ("Age is not DEFINED")
28 endif()
29
30 # Test if target_link_library is a command
31 if (COMMAND target_link_library)
32     message ("target_link_library is a command")
33 else()
34     message ("target_link_library is NOT a command")
35 endif()
36
37 # Test if target_link_libraries is a command
38 if (COMMAND target_link_libraries)
39     message ("target_link_libraries is a command")
40 else()
41     message ("target_link_libraries is NOT a command")
42 endif()
43
44 # Test if CMakeLists.txt exists in this module example
45 if (EXISTS ${CMAKE_CURRENT_SOURCE_DIR}/CMakeLists.txt)
46     message ("File exists")
47 else()
48     message ("File does not exists")
49 endif()
50
51 set (Name1 Rafael)
52 set (Name2 Eduardo)
53
54 # Comparing alphabetic (string smaller is found first in dictionary)
55 # Note: Uppercase chars come before lowercase chars
56 if (Name1 STRLESS Name2)
57     message ("${Name1} is less than ${Name2}")
58 elseif(Name1 STRGREATER Name2)
59     message ("${Name1} is greater than ${Name2}")
60 elseif (Name1 STREQUAL Name2)
61     message ("${Name1} is equal than ${Name2}")
62 endif()
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/5th_Module/1st_Example$ cmake -P CMakeLists.txt
Block ELSE executed
Name is DEFINED
Age is not DEFINED
target_link_library is NOT a command
target_link_libraries is a command
File does not exists
Rafael is greater than Eduardo
```

5.2 - 2nd Example - Boolean Operators: NOT , OR, AND

This operators can be combined with unary and binary tests to write more complex conditons. We can see an example bellow:

```
$ cd ../2nd_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 if (NOT DEFINED Test1 AND NOT DEFINED Test2)
 4     message ("There is an undefined test var!")
 5 elseif(NOT ${Test1} OR NOT ${Test2})
 6     message ("There is an empty test var!")
 7 endif()
 8
 9 set (Test1 "Rafael Del Pino")
10 set (Test2 "")
11
12 if (NOT DEFINED Test1 OR NOT DEFINED Test2)
13     message ("There is an undefined test var!")
14 elseif(NOT ${Test1} OR NOT ${Test2})
15     message ("There is an empty test var!")
16 endif()
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/5th_Module/2nd_Example$ cmake -P CMakeLists.txt
There is an undefined test var!
```

```
There is an empty test var!
```

5.3 - 3rd Example - Loop Command: while

While command will execute the code inside the while() - endwhiele() body while the condition inside parentheses is **true.**

When we use *while* command, the following values are defined as **true**:
1, ON, YES, TRUE, Y, a non-zero number.

When we use *while* command, the following values are defined as **false**:
0, OFF, NO, FALSE, N, IGNORE, NOTFOUND, empty string, string ending with -NOTFOUND

The next example will show hot to set a variable with new values while a limit size is not reached:

```
$ cd ../3rd_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (outString "")
 4 set (buffer "")
 5 set (defaultString "abcdefghijklmnopqrstuvxz")
 6 set (lenght 10)
 7 set (index 0)
 8
 9
10 while (NOT index EQUAL lenght)
11     string(SUBSTRING ${defaultString} ${index} 1 buffer)  # buffer = defaultString[ind
12     string(APPEND outString ${buffer} )                   # outString += buffer
13     message("Index: " ${index} " | Buffer: " ${buffer})   # Print index and defaultStr
14     math(EXPR index "${index} + 1" OUTPUT_FORMAT DECIMAL) # i=i+1
15 endwhile()
16
17 message("Out String: " ${outString})
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/5th_Module/3rd_Example$ cmake -P CMakeLists.txt
Index: 0 | Buffer: a
Index: 1 | Buffer: b
Index: 2 | Buffer: c
Index: 3 | Buffer: d
Index: 4 | Buffer: e
Index: 5 | Buffer: f
Index: 6 | Buffer: g
Index: 7 | Buffer: h
Index: 8 | Buffer: i
Index: 9 | Buffer: j
Out String: abcdefghij
```

5.4 - 4th Example - Loop Command: foreach

The *foreach* command evaluates a group of commands for each value in a referenced list. There are different aproaches in *foreach* using.

```
foreach(<loop_var> <items>)
  <commands>
endforeach()

foreach(<loop_var> RANGE <stop>)
  <commands>
endforeach()

foreach(<loop_var> RANGE <start> <stop>[<step>])
  <commands>
endforeach()

foreach(<loop_var> IN [LISTS [<lists>]] [ITEMS [<items>]])
  <commands>
endforeach()
```

<loop_var> Restrict to loop scope, is the variable that is being iterated
<items> A list of item separated by semicolon or whitespace
<start> The first item to be iterated
<stop> The last item to be iterated
<steps> (not mandatory): step between each iteration

The next example will be shown all aproaces for best understanding:

```
$ cd ../4th_Example
$ touch CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.16.3)

set (variable_1 abcdefghijklmnopqrstuvwxyz)
set (variable_2 a b c d e f g h i j k l m)
set (variable_3 Rafael Oliveira Del Pino)

# Print the only n list item
foreach(n 123456789)
    message (${n})
endforeach()

# Print each n list item
foreach(n 1 2 3 4 5 6 7 8 9)
    message (${n})
endforeach()

# Print value from 1 to 9
foreach(n RANGE 1 9)
    message (${n})
endforeach()

# Print the only v (same as variable_1) list item
foreach(v IN LISTS variable_1)
    message (${v})
endforeach()

# Print each v (same as variable_2) list items
foreach(v IN LISTS variable_2)
    message (${v})
endforeach()

# Print each v (same as variable_3) list items
foreach(v IN LISTS variable_3)
    message (${v})
endforeach()

# Print each v (same as variable_1, variable_2, ariable_3, ) list items
foreach(v IN LISTS variable_1 variable_2 variable_3)
    message (${v})
endforeach()
```

Run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/5th_Module/4th_Example$ cmake -P CMakeLists.txt
123456789
1
2
3
4
5
6
7
8
9
1
2
3
4
5
6
7
8
9
abcdefghijklmnopqrstuvwxyz
a
b
c
d
e
f
g
h
i
j
k
l
m
Rafael
Oliveira
Del
Pino
abcdefghijklmnopqrstuvwxyz
a
b
c
```

```
d
e
f
g
h
i
j
k
l
m
Rafael
Oliveira
Del
Pino
```

6 - Functions and Macros

CMake allow code groupping to avoid repeated code for a cleaner code. This could be made using the parameterizable command *function()* that allow create a group of commands called by an only name.

6.1 - 1st Example - Functions

Lets see an example of functions application. Note that we may create an umparametrized function, or a function with parameter(s) and it is possible to use parameter name inside functions according to function call method.

```
$ cd ../../6th_Module/1st_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (name Rafael)
 4
 5 function(print_standard)
 6     message("Function print standard message")
 7 endfunction()
 8
 9 function(print_variable variable)
10     message("Function print: ${variable}")
11 endfunction()
12
13 function(print_variable_with_name variable)
```

```
14      message("Function print: My ${variable} is ${${variable}}")
15  endfunction()
16
17  print_standard()
18  print_variable(${name})
19  print_variable_with_name(name)
```

Then run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/6th_Module/1st_Example$ cmake -P CMakeLists.txt
Function print standard message
Function print: Rafael
Function print: My name is Rafael
```

6.2 - 2nd Example – Function with optional arguments

The functions of last topic has no argument or a mandatory argument. All CMake functions allows optional arguments. Lets see an example bellow:

```
$ cd ../2nd_Example
$ touch CMakeLists.txt
```

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set (name Rafael)
 4 set (age 31)
 5
 6 function(print_variable variable)
 7     if (DEFINED ARGV1)
 8         message("Function print: My ${variable} is ${${variable}}/
 9                 and my ${ARGV1} is ${${ARGV1}} years old")
10     endif()
11 endfunction()
12
13 function(check_arguments)
14
```

```
15 endfunction()
16
17 print_variable(name age)
```

Then run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/6th_Module/2nd_Example$ cmake -P ./CMakeLists.txt
Function print: My name is Rafael/
                and my age is 31 years old
```

6.3 - 3rd Example – Macros

Macros uses the same functions syntax. When it is invoked, the commands recorded in the macro are first all modified before any is run by replacing formal parameters (${arg1}) with the arguments passed (similar to #define directive from C/C++ languages). In other words, function pushes and pops new variable scope (variables created and changed exist only in the function), macro does not. Because of this, in the following example code, the variable inside function allow changing values while macro not.

```
 1 cmake_minimum_required(VERSION 3.16.3)
 2
 3 set(var_1 "Rafael")
 4
 5 macro(macro_1 arg)
 6   message("macro_1(arg) = ${arg}")
 7   set(arg "Del Pino")
 8   message("macro_1(arg) = ${arg}")
 9 endmacro()
10
11 function(function_1 arg)
12   message("function_1(arg) = ${arg}")
13   set(arg "Del Pino")
14   message("function_1(arg) = ${arg}")
15 endfunction()
16
17 macro_1(${var_1})
18 function_1(${var_1})
19
```

```
20 message ("var = ${var_1}")
```

Then run:

```
$ cmake -P CMakeLists.txt
```

Output:

```
rafael:~/Desktop/CMakeLearning/6th_Module/3rd_Example$ cmake -P ./CMakeLists.txt
macro_1(arg) = Rafael
macro_1(arg) = Rafael
function_1(arg) = Rafael
function_1(arg) = Del Pino
var = Rafael
```

7 - Cache Variables

All the shown variables that we used are called Normal Variables. These variables has local scope. There are two types of global scope variables: Environment Variables and Cache Variables. If we go back to the xxxx example and run the following command, we will see that we have XXXXX Cache Variables and it is possible check their values by acessing the file *CmakeCache.txt*.
The following commands will recompile XXXXXX example and count the quantity (in this example, 137) of Cache Variables available:

```
$ cmake ~/Desktop/CmakeLearning/3rd_Module/4th_Example/ -B
~/Desktop/CmakeLearning/3rd_Module/4th_Example/build

$ grep -c "=" ~/Desktop/CMakeLearning/3rd_Module/4th_Example/build/CMakeCache.txt
```

As we could see, the cache variables are stored inside the file *CMakeCache.txt* inside build folder.

7.1 - Creating and setting a Cache Variable

It is possible create a cache variable using de command *set()*. Here is a step by step to set a new cache variable and use its value in a CMake program:

- Create a *CMakeLists.txt* empty file;

- Create new variable using the command *set()*
- Run *cmake ..* command (the cache variable will be created inside *CMakeCache.txt* file).
- For using he variable value in the CMake program, insert the directive *CACHE* befor bracket variable name.

Now, we will see a complete and practical example:

```
$ cd ../../7th_Module/1st_Example/
$ mkdir build
$ cd build
$ touch CMakeLists.txt
```

```
1 cmake_minimum_required(VERSION 3.16.3)
2 project(CacheExample VERSION 7.1.0)
3
4 set (NAME "Eduardo 16/09/2020" CACHE STRING "This is an example")
5
6 message ("CACHE VALUE: $CACHE{NAME}")
```

Then run:

```
$ cmake ..
```

Output:

```
rafael:~/Desktop/CmakeLearning/7th_Module/1st_Example $ cmake ..
CACHE VALUE: Eduardo 16/09/2020
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/rafaeldelpino/Desktop/CMakeLearning/7th_Module/1st_Example/build
```

7.2 - Changing Cache Variable value

By logical thinking, to change the value of a cache variable the only work to do is insert the function *set (variable_name variable_value CACHE)* but **not! This will not change the cache variable value**. The set comand only works in the first definition of cache variables. This happens because *CMakeCache.txt* is generated in the first *cmake ..* command only.

The options to edit cache variables are:

- Directly edit *CmakeCache.txt* file using a text editor ;

- Using the *FORCE* keyword
- Using the flag *-D* on command line interface.

7.3 - Some Default Cache Variables