“UniTs” - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

Advanced Programming

Lecturer:
Prof. Pasquale Claudio Africa

Author:
Christian Faccio

December 23, 2024

github.com/christianfaccio christianfaccio@outlook.it

Abstract

As a student of the “Data Science and Artificial Intelligence” master’s degree at the University of Trieste, I have created these notes to study the course “Advanced Programming” held by Prof. Pasquale Claudio Africa. The course aims to provide students with a solid foundation in programming, focusing on the C++ programming language. The course covers the following topics:

- Bash scripting
- C++ basics
- Object-oriented programming
- Templates
- Standard Template Library (STL)
- C++11/14/17/20 features
- Parallel programming (not covered in the lectures but useful for the HPC course)

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in this field.

Contents

1 Makefile

2 CMake

2.1 Introduction

CMake stands for ”Cross-Platform Make.” It is a **build-system generator**, meaning it creates the files (e.g., `Makefile`, Visual Studio project files) needed by your build system to compile and link your project. CMake abstracts away platform-specific build configurations, making it easier to maintain code that needs to run on multiple platforms.

It works the following way:

1. You write a `CMakeLists.txt` file that describes your project’s configuration and structure.
2. You run CMake on the `CMakeLists.txt` file to generate the build system files (e.g. `Makefile` on Linux or .snl for Visual Studio).
3. You use the generated build system to compile and link your project.

2.2 CMakeLists.txt

Contains the configuration and structure of your project. It is a script that CMake uses to generate the build system files. It has the following structure:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_executable(my_project_main.cpp)
```

2.2.1 Minimum Version

Here is the first line of every `CMakeLists.txt`, which is the required name of the file CMake looks for:

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
```

The version on CMake dictates the policies. Starting in CMake 3.12, this supports a range like `3.12...3.15`. This is useful when you want to use new features but still support older versions.

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.12...3.15)
```

2.2.2 Setting a project

Every top-level CMake file will have this line:

CMakeLists.txt

```cmake
project(MyProject VERSION 1.0
    DESCRIPTION "My Project"
    LANGUAGES CXX)
```

Strings are quoted, whitespace does not matte and the name of the prokect is the first argument. All the keywords are optional. The `version` sets a bunch of variables, like `MyProject_VERSION` and `PROJECT_VERSION`. The `LANGUAGES` keyword sets the languages that the project will use. This is useful for IDEs that support multiple languages.

2.2.3 Making an executable

CMakeLists.txt

```cmake
add_executable(my_project_main my_project_main.cpp)
```

`my_project` is both the name of the executable file generate and the name of the CMake target created. The source file comes next and you can add more than one source file. CMake will only compile source file extensions. The headers will be ignored for most purposes; they are there only to be showed up in IDEs.

2.2.4 Making a library

CMakeLists.txt

```cmake
add_library(my_library STATIC my_library.cpp)
```

`STATIC` is the type of library. It can be `SHARED` or `MODULE`. The source files are the same as for executables. Often you'll need to make a fictional target, i.e., one where nothing needs to be compiled, for example for header-only libraries. This is called an `INTERFACE library`, and the only difference is that it cannot be followed by filenames.

2.2.5 Targets

Now we've specified a target, we can set properties on it. CMake is all about targets and properties. An executable is a target, a library is a target. Your application is built as a collection of targets depending on each other.

CMakeLists.txt
```cmake
target_include_directories(my_library PUBLIC include)
```

This sets the include directories for the target. The `PUBLIC` keyword means that the include directories will be propagated to any target that links to `my_library`. We can then chain targets:

CMakeLists.txt
```cmake
add_library(my_library STATIC my_library.cpp)
target_link_libraries(my_project PUBLIC my_library)
```

This will link `my_project` to `my_library`. The `PUBLIC` keyword means that the link will be propagated to any target that links to `my_project`.

Targets can have include directories, linked libraries (or linked targets), compile options, compile definitions, compile features and more.

2.2.6 Variables

`Local variables` are used to store values that are used only in the current scope:

CMakeLists.txt
```cmake
set(MY_VAR "some_file")
```

The names of the variables are case-sensitive and the values are strings. You access a variable by using `${}`. CMake has the concept of scope; you cna access the value of the variable after you set it as long as you are in the same scope. If you leave a function or a file in a sub directory, the variable will no longer be defined. You can set a variable in the scope immediately above your current one with `PARENT_SCOPE` at the end.

One can also set a list of values:

CMakeLists.txt
```cmake
set(MY_LIST "value1" "value2" "value3")
```

which internally becomes a string with semicolons. You can access the values with `${MY_LIST}`.

If you want to set a variable from the command line, CMake offers a variable cache. `Cache variables` are used to interact with the command line:

CMakeLists.txt

```
set(MY_CACHE_VAR "VALUE" CACHE STRING "Description")

option(MY_OPTION "Set from command line" ON)
```

Then:

CMakeLists.txt

```
cmake /path/to/src/ \
-DMY_CACHE_VAR="some_value" \
-DMY_OPTION=OFF
```

`Environment variables` are used to interact with the environment:

CMakeLists.txt

```
# Read
message(STATUS $ENV{MY_ENV_VAR})

# Write
set(ENV{MY_ENV_VAR} "some_value")
```

But it is not recommended to use environment variables in CMake.

2.2.7 Properties

The other way to set properties is to use the `set_property` command:

CMakeLists.txt

```
set_property(TARGET my_library PROPERTY CXX_STANDARD 17)
```

This is like a variable, but it is attached to a target. The `PROPERTY` keyword is optional. The `CXX_STANDARD` is a property that sets the C++ standard for the target.