"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

January 8, 2025

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)
- Libraries
- Makefile
- CMake
- C++11/14/17/20 features
- Integration with Python
- 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_minimum_required(VERSION 3.12...3.15)
```

2.2.2 Setting a project

Every top-level CMake file will have this line:

CMakeLists.txt

```
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

```
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

```
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

```
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

```
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

```
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

```
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.

3 Integration with Python

3.1 Introduction

C++ and Python are powerful in their own right, but they excel in different areas. C++ is renowned for its performance and control over system resources, making it ideal for CPU-intensive tasks and systems programming. Python, on the other hand, is celebrated for its simplicity, readability, and vast ecosystem of libraries, especially in data science, machine learning, and web development.

- In research areas like machine learning, scientific computing, and data analysis, the need for processing speed and efficient resource management is critical.
- The industry often requires solutions that are both efficient and rapidly developed.

By integrating C++ with Python, you can create applications that harness the raw power of C++ and the versatility and ease-of-use of Python. Python, despite its popularity in these fields, often falls short in terms of performance. Knowledge of how to integrate C++ and Python equips with a highly valuable skill set.

Several libraries are available for this, each with its own set of advantages and drawbacks. We will use `pybind11`, a lightweight header-only library that exposes C++ types in Python and viceversa.

3.2 pybind11

3.2.1 Overview

pybind11 is a lightweight, header-only library that connects C++ types with Python. This tool is crucial for creating Python bindings of existing C++ code. Its design and functionality are similar to the Boost.Python library but with a focus on simplicity and minimalism. pybind11 stands out for its ability to avoid the complexities associated with Boost by leveraging C++11 features.

To install it on your system, you can use the following command:

bash

```
pip install pybind11
```

or with conda:

bash

```
conda install -c conda-forge pybind11
```

or with brew:

```bash
brew install pybind11
```

You can also include it as a submodule in your project:

```bash
git submodule add -b stable https://github.com/pybind/
    pybind11 extern/pybind11
git submodule update --init
```

This method assumes dependency placement in extern/. Remember that some servers might require the .git externsion. After setup, include extern/pybind11/include in your project, or employ pybind11's integration tools.

3.2.2 Basics

All pybind11 code is written in C++. The following lines must always be included in your code:

```C++
#include <pybind11/pybind11.h>
namespace py = pybind11;
```

The first line includes the pybind11 library, while the second line creates an alias for the pybind11 namespace. This alias is used to simplify the code and make it more readable. In practice, implementation and binding code will generally be located in separate files.

The `PYBIND11_MODULE` macro is used to create a Python module. The first argument is the module name, while the second argument is the module's scope. The module name is the name of the Python module that will be created, while the scope is the C++ namespace that contains the functions to be exposed and is the main interface for creating bindings. Example:

```C++
#include <pybind11/pybind11.h>
namespace py = pybind11;

int add(int i, int j) {
    return i + j;
}
PYBIND11_MODULE(example, m) {
    m.def("add", &add, "A function that adds two numbers");
}
```

Here we define a simple function that adds two numbers and bind it to a Python module named example. The function add is exposed to Python using the m.def() function. The first argument is the function name in Python, the second argument is the C++ function, and the third argument is the function's docstring.

Observation: *pybind11*

Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

Being it a header-only library, pybind11 does not require any additional linking or compilation steps. You can compile the code as you would any other C++ code. The resulting shared library can be imported into Python using the import statement. Compile the example in Linux with the following command:

terminal

```
g++ -O3 -Wall -shared -std=c++11 -fPIC
\$(python3 -m pybind11 --includes)
example.cpp -o example\$(python3-config --extension-suffix)
```

If you included pybind11 as a submodule, you can use the following command:

terminal

```
g++ -O3 -Wall -shared -std=c++11 -fPIC
-Iextern/pybind11/include
\$(python3-config --incldudes) Iextern/pybind11/include
example.cpp -o example\$(python3-config --extension-suffix)
```

This assumes that pybind11 has been installed with `pip` or `conda`, otherwise you can manually specify `-I <path-to-pybind11>/include` together with the Python includes path `python3-config --includes`.

Warning: *On macOS*

the build command is almost the same but it also requires passing the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

Building the C++ code will produce a binary module file that can be imported in Python with the `import` statement. The module name is the name of the shared library file without the extension. In this case, the module name is example. The shared library file is named example.so on Linux and example.dylib on macOS.

python

```
import example
print(example.add(1, 2)) #output: 3
```

With a simple modification, you can inform Pythn about the names of the arguments:

C++

```cpp
    m.def("add", &add, "A function that adds two numbers",
    py::arg("i"), py::arg("j"));
```

You can now call the function using keyword arguments:

python

```python
    import example
    print(example.add(i=1, j=2)) #output: 3
```

Observation: *Documentation*

The docstring is automatically extracted from the C++ function and displayed in Python. This feature is useful for documenting the function's purpose and parameters. The docstring can be accessed in Python using the `__doc__` attribute.

help(example)

…

FUNCTIONS

add(…)

Signature: add(i: int, j: int) -> int

A function that adds two numbers

There is a shorthand notation

C++

```cpp
    using namespace py::literals;
    m.def("add", add, "Docstring", "i"_a, "j"_a=1);
```

The _a suffix is a user-defined literal that creates a py::arg object. This object is used to specify the argument's name and type. The shorthand notation is more concise and easier to read than the previous method. The second argument is optional and specifies the `default` value of the argument. If the argument is not provided, the default value is used.

`py::cast` is used to convert between Python and C++ types. The following example demonstrates how to convert a Python list to a C++ vector:

C++

```cpp
std::vector<int> list_to_vector(py::list l) {
    std::vector<int> v;
    for (auto item : l) {
        v.push_back(py::cast<int>(item));
    }
    return v;
}
PYBIND11_MODULE(example, m) {
    m.def("list_to_vector", &list_to_vector, "Convert a
       Python list to a C++ vector");
}
```

To export variables, use the `attr` function. Built-in types and general objects are automatically converted when assigned as attributed, and can be explicitly converted using `py::cast`.

C++

```cpp
int value = 42;
m.attr("value") = value;
```

python

```python
import example
print(example.value) #output: 42
```

3.2.3 Binding OO code

pybind11 supports object-oriented programming, allowing you to bind classes, methods, and attributes. The following example demonstrates how to bind a simple class:

C++

```cpp
Pet(const std::string &name, int age) : name(name), age(age)
   {}
void set_name(const std::string &name_) { name = name_; }
void set_age(int age_) { age = age_; }
std::string get_name() const { return name; }
int get_age() const { return age; }
```

C++

```cpp
PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &, int>())
        .def("set_name", &Pet::set_name)
        .def("set_age", &Pet::set_age)
        .def("get_name", &Pet::get_name)
        .def("get_age", &Pet::get_age);
}
```

The `py::class_` function is used to bind a C++ class to Python. The first argument is the module, the second argument is the class name, and the third argument is the class type. The `def` function is used to bind class methods to Python. The first argument is the method name in Python, and the second argument is the C++ method.

python

```python
import example
pet = example.Pet("Tom", 5)
print(pet.get_name()) #output: Tom
print(pet.get_age()) #output: 5
pet.set_name("Jerry")
pet.set_age(3)
print(pet.get_name()) #output: Jerry
print(pet.get_age()) #output: 3
```

In the case above, the `print(pet)` statement will output the memory address of the object. To change this behavior, you can define a `__str__` method in the C++ class:

C++

```cpp
std::string __str__() const {
    return name + " is " + std::to_string(age) + " years old"
        ;
}
```

C++

```cpp
.def("__str__", &Pet::__str__);
```

python

```python
import example
pet = example.Pet("Tom", 5)
print(pet) #output: Tom is 5 years old
```

You can also expose the name field with the `def_readwrite` function:

C++

```
    .def_readwrite("name", &Pet::name)
```

python

```
    import example
    pet = example.Pet("Tom", 5)
    print(pet.name) #output: Tom
    pet.name = "Jerry"
    print(pet.name) #output: Jerry
```

Dynamic attributes can be added to the class using the `def_property` function:

C++

```
    .def_property("description", &Pet::__str__, &Pet::set_name)
```

python

```
    import example
    pet = example.Pet("Tom", 5)
    print(pet.description) #output: Tom is 5 years old
    pet.description = "Jerry"
    print(pet.get_name()) #output: Jerry
```

3.2.4 Inheritance and Polymorphism

pybind11 supports inheritance and polymorphism, allowing you to bind base classes and derived classes. The following example demonstrates how to bind a base class and a derived class:

C++

```
    class Animal {
    public:
        virtual std::string speak() const {
            return "I am an animal";
        }
    };
    class Dog : public Animal {
    public:
        std::string speak() const override {
            return "I am a dog";
        }
    };
```

There are two ways to bind the derived class. The first method is to bind the base class and derived class separately, specifying the base class as an extra template parameter of the `class_`:

C++

```cpp
PYBIND11_MODULE(example, m) {
    py::class_<Animal>(m, "Animal")
        .def("speak", &Animal::speak);
    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());
        .def("speak", &Dog::speak);
}
```

python

```python
import example
animal = example.Animal()
dog = example.Dog()
print(animal.speak()) #output: I am an animal
print(dog.speak()) #output: I am a dog
```

the second method is to assign a name to the previously bound `Pet class_` object and reference it when binding the Dog class:

C++

```cpp
py::class_<Pet> pet(m, "Pet");
pet.def(\dots);
py::class_<Dog>(m, "Dog", pet)
    .def(py::init<>());
    .def("speak", &Dog::speak);
```

python

```python
import example
animal = example.Pet()
dog = example.Dog()
print(animal.speak()) #output: I am an animal
print(dog.speak()) #output: I am a dog
```