OOP for Scientific Computing Notes - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

CMake Tutorial

Notes from the official CMake Tutorial link

1 Step 1

- Introduce CMake basic syntax, commands, and variables.
- Do three exercises and create a simple project.

1.1 Exercise 1

- Most basic CMake project is an executable built from a **single file**. Only `CMakeLists.txt` with **three** components is required. This is our **goal** with this exercise.

> **Note**
>
> Stylistically lower case commands are preffered in CMake

1.1.1 The Three Basic Commands

1. Any project's top most `CMakeLists.txt` must start by specifying a minimum CMake version using using the `cmake_minimum_required()` command.
2. Afterwards we use the `project()` command to set the **project name**.
3. Finally we use the `add_executable()` to make CMake create an executable using the specified source code files

1.1.2 Getting Started

We will build the following c++ file that computes the square root of a number:

- We complete the initial 3 TODOS of the `CMakeLists.txt`:

Listing 1.1 `tutorial.cxx`

```
// A simple program that computes the square root of a number
#include <cmath>
#include <cstdlib> // TODO 5: Remove this line
#include <iostream>
#include <string>

// TODO 11: Include TutorialConfig.h

int main(int argc, char* argv[])
{
  if (argc < 2) {
    // TODO 12: Create a print statement using Tutorial_VERSION_MAJOR
    //          and Tutorial_VERSION_MINOR
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }

  // convert input to double
  // TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
  const double inputValue = atof(argv[1]);

  // calculate square root
  const double outputValue = sqrt(inputValue);
  std::cout << "The square root of " << inputValue << " is " << outputValue
            << std::endl;
  return 0;
}
```

1.1.3 Build and Run

1. create a build directory:

   ```
   mkdir build
   ```

2. change into the build directory and build with `cmake`:

   ```
   cd build
   cmake ../
   ```

3. Actually compile/link the project with

Listing 1.2 `CMakelists.txt`

```
# TODO 1: Set the minimum required version of CMake to be 3.10
cmake_minimum_required(VERSION 3.10)

# TODO 2: Create a project named Tutorial
project(Tutorial)

# TODO 7: Set the project version number as 1.0 in the above project command

# TODO 6: Set the variable CMAKE_CXX_STANDARD to 11
#         and the variable CMAKE_CXX_STANDARD_REQUIRED to True

# TODO 8: Use configure_file to configure and copy TutorialConfig.h.in to
#         TutorialConfig.h

# TODO 3: Add an executable called Tutorial to the project
# Hint: Be sure to specify the source file as tutorial.cxx
add_executable(Tutorial tutorial.cxx)

# TODO 9: Use target_include_directories to include ${PROJECT_BINARY_DIR}
```

```
cmake --build .
```

Now an executable `Tutorial` has been created and can be run with

```
./Tutorial 3.0
```

with the output

```
The square root of 3 is 1.73205
```

All good!

1.2 Exercise 2

- CMake has some special variables that have meanig to CMake when set by project
- Many of these variables start with `CMAKE_`. Two of these special variables:

  - `CMAKE_CXX_STANDARD`
  - `CMAKE_CXX_STANDARD_REQUIRED`

- These two together may be used to specify the C++ standard needed to build the project
- **Goal**: Add a feature that requires C++11 and utilize above two variables. TODO4 - TODO6

1.2.1 Getting Started

- TODO 4 & 5 - adding C++11 code to the source `tutorial.cxx`:

Listing 1.3 `tutorial.cxx`

```cpp
// A simple program that computes the square root of a number
#include <cmath>
//#include <cstdlib> // TODO 5: Remove this line
#include <iostream>
#include <string>

// TODO 11: Include TutorialConfig.h

int main(int argc, char* argv[])
{
  if (argc < 2) {
    // TODO 12: Create a print statement using Tutorial_VERSION_MAJOR
    //          and Tutorial_VERSION_MINOR
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }

  // convert input to double
  // TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
  const double inputValue = std::stod(argv[1]);

  // calculate square root
  const double outputValue = sqrt(inputValue);
  std::cout << "The square root of " << inputValue << " is " << outputValue
            << std::endl;
  return 0;
}
```

TODO 6 - set the aforementioned variables:

- `set(CMAKE_CXX_STANDARD 11)`
- `set(CMAKE_CXX_STANDARD_REQUIRED True)`

Listing 1.4 `CMakelists.txt`

```
# TODO 1: Set the minimum required version of CMake to be 3.10
cmake_minimum_required(VERSION 3.10)

# TODO 2: Create a project named Tutorial
project(Tutorial)

# TODO 7: Set the project version number as 1.0 in the above project command

# TODO 6: Set the variable CMAKE_CXX_STANDARD to 11
#         and the variable CMAKE_CXX_STANDARD_REQUIRED to True
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)


# TODO 8: Use configure_file to configure and copy TutorialConfig.h.in to
#         TutorialConfig.h

# TODO 3: Add an executable called Tutorial to the project
# Hint: Be sure to specify the source file as tutorial.cxx
add_executable(Tutorial tutorial.cxx)

# TODO 9: Use target_include_directories to include ${PROJECT_BINARY_DIR}
```

1.2.2 Build and Run

We already created a build directory adn ran `cmake ../` in the previous exercise, which created the project configurations. We don't need to redo this steps, instead we simply rebuild the project:

```
cd build
cmake --build .
```

We run the executable

```
./Tutorial 10
```

to obtain:

```
The square root of 10 is 3.16228
```

1.3 Exercise 3

Sometimes it is useful to have a variable that is defined in `CMakelists.txt` file also be available in source code. In our case we will define the **version number** in `CMakelists.txt` and make it available in a header file.

We can accomplished this with a **configured header file**, where there are two variables that can be replaced marked with `@VAR@`. We use `configure_file()` command to copy the contents of the configured header file to a standard header file, where the `@VAR@` variables are automatically replaced by CMake.

We include this header file generated by CMake in our source code and use the variables defined therein.

We could edit these variables directly in the source code, but using `CMake` avoids duplication and creates a single source of truth.

Goal: Define and report the project's version number. TODOS: 7 - 12.

1.3.1 Getting Started

First we define the version number with `project()` command:

```
project(
  Tutorial
  VERSION 1.0
)
```

Now CMake automatically sets in the background two variables:

- `Tutorial_VERION_MAJOR` as 1
- `Tutorial_VERION_MINOR` as 0

since we defined the VERSION as 1.0.

Now we can utilize these variables in a `TutorialConfig.h.in` file that we will use as an input to CMake to generate a `TutorialConfig.h`.

We create `TutorialConfig.h.in` an add following two lines

Listing 1.5 `TutorialConfig.h.in`

```
//File: TutorialConfig.h.in
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
:::
```

Note that we access the CMake variables that were previously automatically set by the `project()` command via the `@VAR@` syntax.

Next we instruct CMake to generate a `TutorialConfig.h` from `TutorialConfig.h.in` with the `configure_file()` command:

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

The generated header file will be written into the **project binary directory**. In our case it is simply `build/` directory.

We must add this directory to the list of paths that CMake searches for include files with the `target_include_directories()` command:

```
target_include_directories(
  Tutorial
  PUBLIC "${PROJECT_BINARY_DIR}"
)
```

Finally we modify `tutorial.cxx` to include the generated header file:

```
#include "TutorialConfig.h"
```

and include the print directives that utilize the variables from the header file:

```
  if (argc < 2) {
    // TODO 12: Create a print statement using Tutorial_VERSION_MAJOR
    //          and Tutorial_VERSION_MINOR
    std::cout << argv[0] << " Version " << Tutorial_VERSION_MAJOR << "."
                                        << Tutorial_VERSION_MINOR << std::endl;
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
```

1.3.2 Build & Run

Again we only need to rebuild:

```
cd build
cmake --build .
```

If we run `Tutorial` with wrong argument list we get the Version number and the usage message:

```
./Tutorial
```

Output:

```
./Tutorial Version 1.0
Usage: ./Tutorial number
```

The end!

Part II

Basic Concepts of C++

- variables and types
- pointers and references
- control structures
- functions and templates
- classes and inheritance
- namespaces and structure

2 Variables, Temporaries, Literals

2.1 Variables

2.2 Temporaries

2.3 Literals

3 Data Types

3.1 Introducing New Types

3.1.1 Enum

```
enum Color = {RED, BLUE, GREEN}
```

3.1.2 Struct

...

3.2 Const-Correctness

Marks something that can't be modified.

```
include <iostream>

int main(int argc, char const *argv[])
{
    int n = 5;
    const int j = 4;
    const int &k = n; //k can't be modified, equivalently n can't be modified over k
    n++; //but this changes n and indirectly k (because k references n)


    const int *p1 = &n; // modifiable pointer to const int
    int const *p2 = &n; // same thing
    int *const p3 = &n; // constant pointer to modifiable int

    // p1 = &j -- ok
    // *p1 = 3 -- not ok!
    // p3 = &j -- not ok
```

```
    // *p3 = 10 -- ok

    std::cout << "n: " << n << std::endl
              << "j: " << j << std::endl
              << "p1: " << p1 << std::endl
              << "p2: " << p2 << std::endl
              << "p3: " << p3 << std::endl;


    return 0;
}
```

4 Indirection

4.1 Pointers

```
include <iostream>

int main(int argc, char const *argv[])
{
    int i = 5;
    int *p1 = &i;
    int *p2 = new int;

    std::cout << "i: " << i << std::endl
              << "*p1: " << *p1 << std::endl
              << "p1: " << p1 << std::endl
              << "&p1: " << &p1 << std::endl
              << "p2: " << p2 << std::endl
              << "*p2: " << *p2 << std::endl;
    delete p2;
    return 0;
}
```

output:

```
i: 5
*p1: 5
p1: 0x7fff8d568184
&p1: 0x7fff8d568188
p2: 0x55c014358eb0
*p2: 0
```

- release memory with `delete`.
- deleting too early -> bugs, too late -> memory leaks

4.2 References

References are **aliases for an existing entity**. k

```
include <iostream>

int main(int argc, const char** argv) {

    int a = 4;
    std::cout << "a: " << a <<std::endl;
    int &b = a;
    b = 5;
    std::cout << "a: " << a << std::endl
              << "b: " << b << std::endl;

    return 0;
}
```

output:

```
a: 4
a: 5
b: 5
```

4.3 Rvalue (double) References

Two uses:

- **range-based** `for` loops
- **move semantics**

lvalue references refer to entities, rvalue references refer to literals.

5 Control Flow

5.1 If

```
include <iostream>

int main(int argc, char const *argv[])
{
    int i;
    std::cin >> i;

    if (i % 2 == 0) std::cout << i << " is even" << std::endl;
    else std::cout << i << " is odd" << std::endl;
    return 0;
}
```

5.2 Switch

```
include <iostream>

enum Color {RED, BLUE, GREEN};

int main(int argc, char const *argv[])
{
    int i;
    Color c = RED;

    std::cin >> i;

    switch(i) {
        case 0:
            c = RED;
            break;
```

```
        case 1 :
            c = BLUE;
            break;
        case 2 :
            c = GREEN;
            break;
        default :
            std::cout << "error: invalid color" << std::endl;
    }

    std::cout << c << std::endl;

    return 0;
}
```