C/C++ PRIMER

LECTURE 1: INTRODUCTION AND COMPILING CODE

Fabian Wermelinger
Harvard University

OUTLINE

- Procedural languages and Object Oriented Programming
- Compiled languages
  - Stages of compilation
  - Know your compiler
- Building code
  - `make`
  - `meson`
  - `cmake`

SOME REFERENCES

- B. W. Kernighan and D. M. Ritchie, *C Programming Language*, 2nd Edition, Pearson, 1988
- S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd Edition, Addison-Wesley Professional, 2005
- https://en.cppreference.com

PROCEDURAL AND OBJECT ORIENTED PROGRAMMING

PROCEDURAL PROGRAMMING

A programming paradigm that uses *procedures* to modify the *state* in a program.

- **procedure:** function (in `C/C++`) or subroutine (in Fortran)
- **state:** represented by *data*

PROCEDURAL PROGRAMMING

EXAMPLE: C CODE

```c
void reset(int a[], int n) // function definition
{
    for (int i = 0; i < n; ++i) {
        a[i] = 0;
    }
}

#define N 4
int main(void)
{
    int a[N] = {0, 1, 2, 3}; // state (array of N numbers)
    reset(a, N); // procedure or function call changes values in a
    return 0;
}
```

OBJECT ORIENTED PROGRAMMING

In *Object Oriented Programming* (OOP) we also use "functions" (here they are called *methods*), but these methods can *only modify state that is **owned** by the object*. This implies that such methods are *member functions* of some object.

A key concept in OOP is *data encapsulation*. It means that the object has state which is modeled by the data *it* owns. The state can only be changed by the rules of an *interface*, implemented by member functions.

OBJECT ORIENTED PROGRAMMING

EXAMPLE: C++ CODE

```cpp
#define N 4
class A
{
public:
    A() : a{0, 1, 2, 3} {} // initialization of object

    void reset() // publicly accessible member function
    {
        for (int i = 0; i < N; ++i) {
            a[i] = 0;
        }
    }

private:
    int a[N]; // (private) state owned by class A
};

int main(void)
{
    A a;
    a.reset(); // method call modifies state in a (instance of A)
    return 0;
}
```

OBJECT ORIENTED PROGRAMMING

This should give you an idea for one of the core differences between `C` and `C++`. Object oriented programming is not the same as classic procedural programming like Fortran.

Other important concepts in OOP are:

- Inheritance
- Polymorphism

We will discuss them in later lectures.

COMPILED LANGUAGES

COMPILED LANGUAGES

INTERPRETED LANGUAGE

The source code is interpreted by a front-end processor that executes the program in a virtual environment. Python is an example for an *interpreted* language.

COMPILED LANGUAGE

The source code is interpreted by a (front-end) compiler that generates machine code for a particular target platform (e.g. Intel or AMD CPU). `C` and `C++` are examples for *compiled* languages.

STAGES OF CODE COMPILATION

1. ***Preprocessing:***
   - Expand preprocessor macros (lines starting with " `#` ", e.g. `#include`)
   - Remove comments from code
2. ***Compilation:***

   Translate source code into (human readable) assembly code for the target platform.

3. ***Assembly:***

   Translate assembly code into object code that is executable on the target.

4. ***Linking:***

   If you have used external libraries in your program, the linker will add these symbols to your executable.

THE COMPILER IS YOUR FRIEND!

The compiler is an important tool in your toolbox. If you know it well, it will help you analyze your code and warn you when something does not look right.

Commonly used open-source compilers are:

- GNU Compiler Collection (GCC)
- LLVM's clang

THE COMPILER IS YOUR FRIEND!

EXAMPLE: GNU COMPILER

```
$ g++ main.cpp  # see `man g++` for basic usage
$ ls
a.out  main.cpp
$ ./a.out
Hello World!
```

Hands-On: 10 minutes (`hands-on/01/README.md`)
Check the `man`-page for `g++` and compile the source code in `main.cpp` to produce an output executable called `main`. When done, execute your binary.

TOOLS TO BUILD CODE

For a single source file it is OK to invoke the compiler directly. If you have your source code distributed over multiple files it can be tedious and error prone.

CLASSICAL MAKEFILES

- A `Makefile` contains certain rules to transform an input (dependency) to a desired output.
- In our case the input(s) are the source files, the rules are implemented by calling the compiler and the output is our executable.
- You can apply such transformations to anything you like, not only `C/C++` source code. (E.g. generating LaTeX documents)
- Online tutorial: https://makefiletutorial.com

CLASSICAL MAKEFILES

EXAMPLE: MAKEFILE

```
1 # Anatomy of a Makefile:
2 # target: dependency [other dependencies]
3 # <TAB>recipe
4 #
5 main: main.cpp
6     g++ -o main main.cpp
```

The indentation for the recipe *must* be a TAB and not normal space!

```
1 $ ls
2 main.cpp  Makefile
3 $ make
4 g++ -o main main.cpp
5 $ ls
6 main  main.cpp  Makefile
```

CLASSICAL MAKEFILES

EXAMPLE: MAKEFILE (BETTER SOLUTION)

```
1 # Use special tokens that allow you to easily add dependencies later
2 main: main.cpp
3     g++ -o $@ $^
```

- `$@` : will be substituted with the *target* name.
- `$^` : will be substituted with *all* listed dependencies.
- `$<` : will be substituted with *the first* dependency only.

CLASSICAL MAKEFILES

- A `Makefile` is a good solution if your project is small
- It can (and is) used for large projects as well but suffers some optimizations in certain cases (like compiling in parallel)
- `make` is a historical tool and knowing how to use it is valuable for your day to day practice
- Other more recent and more sophisticated tools exist. Examples are `meson` or `cmake`

MESON

The Meson Build System is an open-source project for managing the build process in your code.

- Multiplatform (Linux, OSX, Windows)
- Many supported languages
- Build definitions can be written in a more natural way (python-like syntax, variables)
- In contrast, the syntax in `cmake` is string based (you can not assign values to a variable)
- Easy integration of code coverage as well as testing and benchmarking frameworks
- `meson` is implemented in Python
- Uses `ninja` as a back-end (fast builds)

MESON

Installing `meson` is easy:

```
1 $ python -m pip install --user meson ninja
```

Hands-On: 5 minutes
Install `meson` and `ninja` on your system.

MESON

With `meson` you define your build definitions in `meson.build` files (similar to a `Makefile`).

EXAMPLE: MESON BUILD FILE

```
# This must be be the first call in your meson.build file
project('My_Project', ['cpp'], # project name and a list programming languages
  version: '1.0.0',            # the current version of your code
  license: 'MIT',              # optional: the license of your code
  # more options: https://mesonbuild.com/Reference-manual.html#project
)

# build our executable: you can assign it to a variable (here I called it
# `main`) and use it later, for example when you want to run a test with it.
main = executable('main', # the name of your executable
  files(['main.cpp']),    # a list of source files needed to build it
)
```

MESON

HOW TO BUILD YOUR CODE?

- Meson builds *out of source*
- By default your code is built with debugging flags enabled
- You need to setup your build directory to start building your code
- You can have many build directories in parallel

MESON

SETUP A BUILD DIRECTORY

See `meson help setup` for more details.

```
$ ls
main.cpp  meson.build
$ meson setup my_build
The Meson build system
Version: 0.59.1
Source dir: /path/to/source/dir
Build dir: /path/to/build/dir
Build type: native build
Project name: My_Project
Project version: 1.0.0
C++ compiler for the host machine: c++ (gcc 11.1.0 "c++ (GCC) 11.1.0")
C++ linker for the host machine: c++ ld.bfd 2.36.1
Host machine cpu family: x86_64
Host machine cpu: x86_64
Build targets in project: 1
$ ls
main.cpp  meson.build  my_build
```

MESON

BUILD YOUR CODE

Once you have a build directory set up, you can compile your code.

```
$ meson compile -C my_build
ninja: Entering directory `my_build'
[2/2] Linking target main
$ ls my_build
build.ninja  compile_commands.json  main  main.p  meson-info  meson-logs  meson
$ ./my_build/main
Hello World!
```

You can compile a build directory using `meson compile`. The option `-C <path/to/build/dir>` tells `meson` which build configuration should be used.

MESON

- Meson is a very powerful tool to build your code
- Even more so if your code is large (many files and depends on other libraries)
- It adds aggressive warning flags to the compiler to warn you early if something is not right in your code.

Hands-On: remainder (`hands-on/02/README.md`)
Complete the `meson.build` file to build the code in `main.cpp` with `meson`. The name of your executable should be `main`.