B4 - Year-End Project

B-YEP-400

Introduction to CMake

Indie Studio Bootstrap

2.0

{EPITECH.}

As you already know, Makefiles are a rather simple build system that lets you define targets, their dependencies, and the commands to execute to create them, via a specific syntax.
Makefiles have the advantage of being versatile on Unix systems: they are compatible with any programming language, and can even be used for tasks other than development.

The main problem with Makefiles, however, is that they only work on Unix systems.
Compatibility layers can be added to Windows, but the fact that there is no official or easy-to-configure support for it makes using Makefiles tedious.

But how do the big boys do it then?
How do cross-platform applications get built?
Well, they use a **dedicated build system**.
These are cross-platform applications that, just like Makefiles, let you easily define targets and their dependencies, and then build them no matter what platform you are running them on.

The most common build system used for C++ is CMake.
It is available as a package on most Linux distributions, and a Windows installer is available on their website.
Go ahead and **install it**, you'll need it for as long as you write C++ code.

We'll start by working on a Unix system, just so you get an understanding of the syntax used by CMake and what it actually does.
Just like `make` uses Makefiles to define projects, CMake uses a file called **CMakeLists.txt**.
Create a directory with the following structure:

```
cross_platform/
   |--- CMakeLists.txt
   |--- src/
          |--- main.cpp
```

In `main.cpp`, write any C++ code you like.
What the program does doesn't matter right now, all we're trying to do is get it to compile.

Inside `CMakeLists.txt`, **define an executable target**.

Whereas you would normally just run `make` to compile your project, CMake adds another step: it generates a platform-dependent way to compile.

Unix

By default, on Unix systems, CMake generates a Makefile.
Along the way, it's actually going to create a whole bunch of other stuff too. In order to keep our directories clean, let's create a `build` directory in which we'll let CMake generate its stuff:

```
cross_platform/
   |--- build/
   |--- CMakeLists.txt
   |--- src/
        |--- main.cpp
```

Now, head into build and **run the `cmake` command**, giving it as an argument the path to the directory containing your `CMakeLists.txt`.

If you did everything right, CMake should have generated a Makefile for you. You can now run `make` and execute your program. Magic, isn't it?

Windows

On Windows, CMake generates a Visual Studio solution for C++ programs.
Solutions are typically configured at the start of a project using a dedicated (and cumbersome) GUI.

I'm sure you can already guess all the problems with this: solutions are not portable to other platforms, the GUI configuration means developers can't easily use git or other tools to control their project configuration…

Copy your directory to your Windows machine (you'll want to find a way to do that easily, without having to push your files to git as they may not be working yet) and delete the contents of the `build` directory.

Start a Windows command prompt and navigate to your project directory (you may want to run the `help` command to see how `cmd` differs from a standard shell). As you did on Linux, go into the `build` directory and **run `cmake ...`**

If you did everything right, CMake must have generated a Visual Studio solution.

But how do you build the project now?
With Makefiles, you simply had to run `make`, but you have no idea how to build a Visual Studio project!
You could open the solution in Visual Studio and hit the big "Build" button, but that implies starting up Vi-

sual Studio just to build your project, which might take a while if you do it regularly.

Thankfully, CMake has implemented a cross-platform option to build the project.

```
cmake --build .
```

And there you have it.
CMake just ran the **MSVC (MicroSoft Visual C++)** compiler to generate an executable, just like it did on Linux.

THERE BE DRAGONS

You now know how to set up a cross platform project using CMake.
Easy, isn't it?
However, setting up the project isn't the only tough part in writing cross platform applications.
Up until now, you've only compiled your projects using one compiler: `g++` (perhaps some of you toyed with `clang`, but generally just to get different error messages, right?).

Starting now, you'll be working with compilers which sometimes have slightly different behaviors.
Some things may compile under `gcc` but not under `msvc`, and some things might work under `msvc` but not under `gcc`.
It'll be small things, like forgetting to include a header file or to specify an override specifier… But it's important that you compile on both platforms regularly!

If you wait until the end of a project to try and compile it on one of the two platforms, you'll be flooded with tons of little errors and it'll take you a very long time to fix them all and test your program again.

Try to use CMake to compile your old projets **with at least two differents compilers**.

Compile often, commit often!