`C++` development issues

CI/CD

When working with C++ projects, development pipelines were created with GitHub Actions or CircleCI. There are repositories which contain sources and documentation for various issues which appeared when building CI/CD pipelines on MacOS and other Linux distributions.

- A project that contains a CI/CD pipeline via GitHub action is available here.
- Documentation on building CI/CD pipelines and also integrating CMake into C++ projects can be accessed here
  - This repo contains pipelines for deploying projects to **GitHub Actions**, **CircleCI**
    - it contains also sources and docs for different issues: docs
- Docs and sources for implementing CMake into any C++ project: this GitHub repo.

Python Bindings

Extending the Python functionality with the help of C++-based modules. This repository contains a collection of working Python extensions, built for multi-platform use-case.

- Project has a CircleCI pipeline, which builds the extensions on multiple platforms.
- The repo has a documentation and a description with issues that appeared during development cycle.
- More details can be seen at the following links:
  - Official GitHub repo
  - Docs & issues

Dev-issues

This document contains the issues which prevented a proper workflow in `C++`projects:

- prevent proper source(s) compilation
- prevent proper code execution on the machine (via command line)
- prevent proper linking of different executables with their respective necessary libraries.

Missing `string.h` header file

When compiling some C++ code that contained the header `string` (basically when trying to work with strings), the compiler also tried to search

for the header `string.h` (due to it being present in the former header, as an `include_next<string.h>`).

That made the the compilation of any source file which contained string header to fail. An in-depth fix about this issue was made in a previous repository, which is found here.

Solution: just create a `CPATH` that points to the `include` directory which has both header files. In the present case, the headers were located in the XCode's header files for `C++` development.

This GitHub issue mentions the fact that this is mandatory for compiling with *llvm*'s clang++ .

Exporting the CPATH final fix

```
export CPATH=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
```

Only XCode is needed (no Command Line Tools).

Issue with `rand` and `math` headers

When trying to compile source files that used methods from the `cmath` and `random` headers (in fact, `random` is ultimately driven by the formerly mentioned header), the compilation would fail, throwing errors like this:

```
In file included from main.cpp:2:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../in
      'signbit' in the global namespace
using ::signbit;
      ~~^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../in
      'fpclassify' in the global namespace
using ::fpclassify;
      ~~^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../in
      'isfinite' in the global namespace; did you mean 'finite'?
using ::isfinite;
      ~~^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX1
      'finite' declared here
extern int finite(double)
```

Solution: In a question from SO, the solution is mentioned at the end, where, according to this issue, compiling with the `CXXFLAGS` properly set to point to XCode C++ SDK, the source files will compile with success.

Setting the `~/.zshrc` file to export the `CXXFLAGS` with the path to the SDK, and then creating an alias with the compiler (e.g. clang++) to always use the flag will solve the compilation issue.

`-isysroot /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDK`

The path to the SDK.

`alias silang++="/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolcha`

An alias which allows clang++ compiler to properly find the SDK and compile the source code.

Exporting the flag:

`export CPPFLAGS="-isysroot/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platfo`

This comment shows on how to set other variables which might help with the proper include paths on compilation.

Another solution

Export the actual `SDKROOT` environment variable like this

```
$ export SDKROOT="$(xcrun --sdk macosx --show-sdk-path)"
```

Setting up environment variables for OSX

In order to have a proper development environment, some flags should be passed to the `C++/C` compiler. This is a useful post for setting up flags which the compiler can take.

```
export CFLAGS="-I$LOCAL/include"     # for the C compiler
export CXXFLAGS="-I$LOCAL/include"   # for the C++ compiler
export LDFLAGS="-L$LOCAL/lib"        # for the linker
```

From the post:

> *If you often need more than one custom library, this will quickly become inconvenient; you should instead specify pkg-config path and cflags for all libraries at once.*
> *You may also need to point the loader to your library location while compiling. For Linux, it would be:*

```
export LD_LIBRARY_PATH="$LOCAL/lib"
```

> *This shouldn't be necessary when running compiled programs; the linker will have added the correct paths to the compiled binary itself. For Mac OS X, setting DYLD_LIBRARY_PATH should have a similar effect; otool can be used for inspecting applications.*

From this post > Ensure clang knows where to find the macOS SDK. You can include something like this in your ~/.bash_profile (if you use bash) or ~/.zshrc file (if you use zsh). Note that Catalina's default shell is now Zsh.

```
XCBASE=`xcrun --show-sdk-path`
export C_INCLUDE_PATH=$XCBASE/usr/include
```

```
export CPLUS_INCLUDE_PATH=$XCBASE/usr/include
export LIBRARY_PATH=$XCBASE/usr/lib
```

- Another useful guide for setting up env. vars: https://stackoverflow.com/questions/12102125/how-to-add-this-line-to-environment-variables-for-osx

This post on SO about *Include search path on Mac OS X Yosemite 10.10.1* has a great answer that describes a set of env vars which can be properly exported.

```
# Put cross compile tools on the PATH first
export PATH="/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/

# C compiler
export CC=/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr

# C++ compiler
export CXX=/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/us

# SYSROOT brings in platform headers and libraries, No need for -I, -L and -l.
SYSROOT=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDK

# Compiler flags
export CFLAGS="-march armv7 -march armv7s --sysroot=$SYSROOT"

# Compiler flags
export CXXFLAGS="-march armv7 -march armv7s --sysroot=$SYSROOT"

# Using defualt C++ runtime
$CXX $CXXFLAGS foo.c -o foo.exe
```

I'd like to change library search path... >At compile time, you augment the library search path with -L. You cannot delete paths; you can only add paths that have a "higher" preference than existing paths.

> At runtime, you use `DYLD_LIBRARY_PATH`, `DYLD_FALLBACK_LIBRARY_PATH` and friends to change the library search path. See dyld(1) OS X Man Pages.
>
> Usually you use `DYLD_LIBRARY_PATH` to ensure a particular library is loaded rather than the system one. Its a way to override default behavior. `DYLD_FALLBACK_LIBRARY_PATH` is used to provide a library that's not a system one. Its less intrusive because you don't displace system paths.

I need /usr/local/include first...

```
$ export DYLD_LIBRARY_PATH=/usr/local/include
$ clang ...
```

Setting the environment for compiling C++ via cmake projects

This post describes how to set up cmake in order to compile C++ sources, looking into the proper SDK on OSX.

From the post:

> You must point the build system of the code you're trying to compile to the right headers:
>
> 1. Make sure Xcode is up to date. There's no telling what an outdated Xcode on Catalina might do to your build environment.
> 2. Use the `-isysroot /sdk/path` compiler flag, where /sdk/path is the result of `xcrun --show-sdk-path`.

```
set(CMAKE_OSX_SYSROOT /sdk/path) # any of these should work
set(CMAKE_CXX_FLAGS "[...] -isysroot /sdk/path")
```

This is also considered a solution to missing headers:

```
XCBASE=`xcrun --show-sdk-path`
export CPLUS_INCLUDE_PATH=$XCBASE/usr/include
```

Example of cmake build

```
1 cd ~/gcc_all/gcc-10.1.0
2 mkdir build && cd build
3
4 ../configure --prefix=/usr/local/gcc-10.1.0 \
5              --enable-checking=release \
6              --enable-languages=c,c++,fortran \
7              --disable-multilib \
8              --with-sysroot=/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk \ # the sy
9              --program-suffix=-10.1
```

Can't run executable from the install project tree (CMAKE)

This issue has been discussed here as well. The solution is to use proper flags in the `CMakeLists.txt` file.

The `set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib")` command will make the executable to look into the correct path of the `libs` directory.

This is an awesome answer from an SO post that explains how the rpath actually works. See documentation for other links.

The solution in the present situation (developing on OSX Catalina with CMake3) was found in this answer:

```
set(CMAKE_MACOSX_RPATH 1)
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib")
```