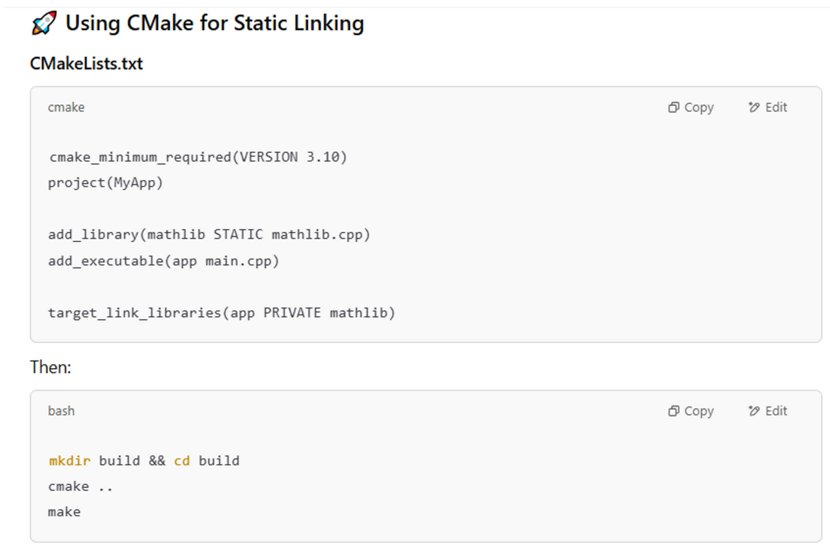
49 - Using Libraries in C++ (Static Linking)

sexta-feira, 14 de março de 2025 07:16

- Libraries in c++
- Use external libraries in our code
- Everyone as problems with libs
- Hate the idea of packag manages or link to other repositories
- The idea is to have everything you need to compile and run the projects
- Should be no sync with other repostories
- For c== I just want things to work, clone the repo and compile it
- so, add all the libs we need in out project
- Should I be compiling it my self, or bind in old compiled binary
    - If you don't have access to the source code... link against the binary
    - Otherwise, build it yourself
- Compile our selfs means we can make some changes a well, more freedon

[Using Libraries in C++ (Static Linking)](#)

- Using **libraries in C++ with static linking** means that all the code from the library is copied into your executable at compile time. This results in a standalone binary that doesn't depend on the library being present at runtime.
- Static linking **increases executable size** but makes deployment easier since all code is bundled.
- In **Windows**, you might see .lib instead of .a.
    - You don't need the .a or .lib file at runtime—only at compile time.
- Static, the lib is inside the executable when the dinamic is linked in runtime
    - At the application running, oad the ap and you get access to the functions
- Better to use static libs when we can
    - No extra depencency to be copied together... like other DLLs with the executable
    - tenicly faster, because of otimiaztions because we know what we are linking at that time
    - usually static is the way to go



🚀 Using CMake for Static Linking

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.10)
project(MyApp)

add_library(mathlib STATIC mathlib.cpp)
add_executable(app main.cpp)

target_link_libraries(app PRIVATE mathlib)
```

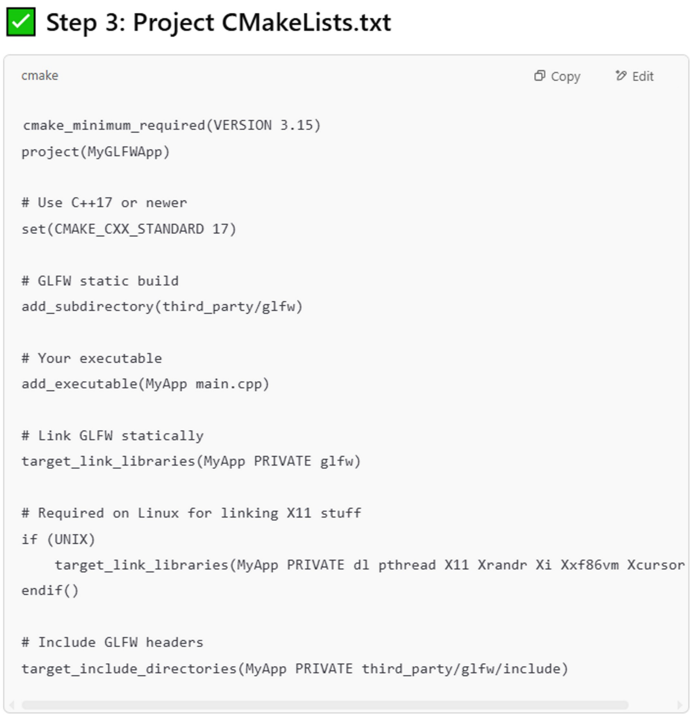
Then:

```bash
mkdir build && cd build
cmake ..
make
```

Build GLFW staticly

1. Clone GLFW:
    git clone https://github.com/glfw/glfw.git
    cd glfw

2. Build it statically:
    cmake -B build -DGLFW_BUILD_DOCS=OFF -DGLFW_BUILD_EXAMPLES=OFF -DGLFW_BUILD_TESTS=OFF -DBUILD_SHARED_LIBS=OFF .
    cmake --build build
3. Setup Your Project Folder
    a. place the GLFW binary + headers in a third_party folder:
4. Project CMakeLists.txt
    a. Follow the exemple n step 3, same structure as used in the exeme
5. Build the Project
    mkdir build && cd build
    cmake ..
    make
6. This will build the dependency locally... and attache to the main project binary
    a. The include files are under the include "include/GFW/include"
    b. And the binaries ".a" is under "include/GLFW/build/src"



✅ Step 3: Project CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.15)
project(MyGLFWApp)

# Use C++17 or newer
set(CMAKE_CXX_STANDARD 17)

# GLFW static build
add_subdirectory(third_party/glfw)

# Your executable
add_executable(MyApp main.cpp)

# Link GLFW statically
target_link_libraries(MyApp PRIVATE glfw)

# Required on Linux for linking X11 stuff
if (UNIX)
    target_link_libraries(MyApp PRIVATE dl pthread X11 Xrandr Xi Xxf86vm Xcursor
endif()

# Include GLFW headers
target_include_directories(MyApp PRIVATE third_party/glfw/include)
```

Using Dynamic Libraries in C++

- Link that happens in runtime
    - When lounch the eecutable, it gets incorporated to memory
    - load na external file into memory
- The optimzations did'nt happens because the compler desn't know everything that is coded there with dynamic
- Rquires some external files to exist before lounch the aplication

✅ 1. **Install GLFW as a shared (dynamic) library**

If you're using a package manager like `vcpkg` or `apt`:

Option A: Using `vcpkg` (recommended)

```bash
vcpkg install glfw3
```

Then integrate it:

```bash
./vcpkg integrate install
```

Option B: Ubuntu/Debian

```bash
sudo apt install libglfw3-dev
```

✅ 2. **CMake Configuration (Dynamic Linking)**

Here's a minimal CMake setup that links to GLFW dynamically:

`CMakeLists.txt`

```cmake
cmake_minimum_required(VERSION 3.10)
project(MyGLFWApp)

set(CMAKE_CXX_STANDARD 17)

# Find GLFW
find_package(glfw3 REQUIRED)

# Add your source files
add_executable(my_app main.cpp)

# Link GLFW dynamically
target_link_libraries(my_app glfw)
```

> 💡 `find_package(glfw3 REQUIRED)` **assumes GLFW is installed and available through** `CMAKE_PREFIX_PATH`**,** `vcpkg`**, or system-wide (** `/usr/lib`**,** `/usr/local/lib`**, etc).**

✅ 3. **Make Sure to Provide the Shared Library at Runtime**

Dynamic linking means the GLFW `.so` or `.dll` file must be available at runtime.

- On **Linux**: make sure the library is in a path known to the linker.

  ```bash
  export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
  ```

- On **Windows**: place `glfw3.dll` in the same directory as your `.exe`.
- On **macOS**: make sure `.dylib` is in the correct system path or use `install_name_tool`.

📄 3. `CMakeLists.txt`

cmake Copy Edit

```cmake
cmake_minimum_required(VERSION 3.10)
project(GLFWDemo)

set(CMAKE_CXX_STANDARD 17)

find_package(glfw3 REQUIRED)

add_executable(glfw_demo main.cpp)
target_link_libraries(glfw_demo glfw)
```

🐳 4. `Dockerfile` (Ubuntu + GLFW + CMake)

Dockerfile Copy Edit

```Dockerfile
FROM ubuntu:22.04

# Install basic tools
RUN apt update && apt install -y \
    cmake \
    build-essential \
    libglfw3-dev \
    libgl1-mesa-dev \
    libx11-dev \
    libxrandr-dev \
    libxi-dev \
    libxext-dev \
    libxinerama-dev \
    libxcursor-dev \
    x11-utils \
    && apt clean

WORKDIR /app

COPY . .

RUN cmake -S . -B build \
 && cmake --build build
```

🚀 5. Build & Run the Container

bash Copy Edit

```bash
docker build -t glfw-app .
docker run -it --rm \
  -e DISPLAY=$DISPLAY \
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  glfw-app
```

🧠 Note: This assumes you're running Docker on Linux with access to the host display server (for GUI apps). If you're on WSL or Mac, let me know — you'll need X server support (e.g., XQuartz or VcXsrv).