Template C++ library

v1.0.0

Table of contents

Overview

TemplateLibrary C++ library is an example of the library structure. It defines data structures and rules to stand as an example for other repositories. In accordance with our documentation standards, it is mandatory for every repository within our project to include a README file like this one. This README should provide comprehensive information about the repository, including its purpose, usage guidelines, and any essential instructions for contributors. Please ensure that the README file follows the format and

content similar to the one you are currently reviewing. This practice contributes to consistency, transparency, and ease of use across all our repositories. This library, as example, depends on open source **ConfigReader** library which provides methods to work with JSON files.

Overview section must include explanation of the purpose of library, C++ standard, important dependencies and main features. This text will be also used for publishing on website. Provide enough information, so that the user can immediately understand the purpose of the library and its compatibility with their project.

Generally any library repository should have following structure:

```
3rdparty ------------- Folder with third-party libraries (dependencies)
    Submodule1 ------- Third-party library folder.
    Submodule2 ------- Third-party library folder.
    CMakeLists.txt --- CMake file to include third-party libraries.
demo ----------------- Demo appllication folder.
example -------------- Simple example folder, which uses in documentation.
src ------------------ Source code folder.
test ----------------- Forlder for tests.
_static -------------- Folder with images for documentation.
.gitignore ----------- File to exclude particular files/folders from repository.
.gitmodules ---------- File defining submodules used in the library.
CMakeLists.txt ------- Main CMake file.
README.md ------------ Documentation.
```

Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0 [MAJOR.MINOR.PATCH]	DD.MM.YYYY	- Make sure to update library versions table after each release. - Briefly describe every important change or update. - Use bullet list style.

Library files

The **TemplateLibrary** is a CMake project. Library files chapter should briefly describe what's inside this repository. It is very important so every associate or customer will understand what particular repository includes:

```
CMakeLists.txt ------------------- Main CMake file of the library.
3rdparty ------------------------- Folder with third-party libraries.
    CMakeLists.txt --------------- CMake file which includes third-party libraries.
    ConfigReader ----------------- Source code of the ConfigReader library.
example -------------------------- Folder with the simplest example of library usage.
    CMakeLists.txt --------------- CMake file for simple example.
```

```
    main.cpp --------------------- Source code file of simple example application.
test ----------------------------- Folder for internal tests of library.
    CMakeLists.txt --------------- CMake file for tests application.
    main.cpp --------------------- Source code file tests application.
src ------------------------------ Folder with source code of the library.
    CMakeLists.txt --------------- CMake file of the library.
    TemplateLibrary.cpp ---------- Source code file of the library.
    TemplateLibrary.h ------------ Header file which includes TemplateLibrary class
declaration.
    TemplateLibraryVersion.h ----- Header file which includes version of the library.
    TemplateLibraryVersion.h.in -- CMake service file to generate version file.
```

TemplateLibrary interface class description

Class declaration

TemplateLibrary interface class declared in **TemplateLibrary.h** file. Class declaration:

```
class TemplateLibrary {
public:
    /// Class constructor.
    TemplateLibrary();

    /// Class destructor.
    ~TemplateLibrary();

    /// Get the version of the TemplateLibrary class.
    static std::string getVersion();

    /// Set the value for a specific library parameter.
    bool setParam(TemplateLibraryParam id, float value);

    /// Get the value of a specific library parameter.
    float getParam(TemplateLibraryParam id);

    /// Get the structure containing all library parameters.
    void getParams(TemplateLibraryParams& params);

    /// Execute a template command.
    bool executeCommand(TemplateLibraryCommand id);

    /// Any useful method of library.
    bool doSomething(int& value);
};
```

getVersion method

getVersion() method returns string of current class version. Method declaration:

```
static std::string getVersion();
```

Method can be used without **TemplateLibrary** class instance:

```
std::cout << "TemplateLibrary version: " << cr::templ::TemplateLibrary::getVersion() <<
std::endl;
```

Console output:

```
TemplateLibrary class version: 1.0.0
```

setParam method

setParam(...) method sets new parameters value. Every implementation of library with parameters have to provide thread-safe **setParam(...)** method call. This means that the **setParam(...)** method can be safely called from any thread. Method declaration:

```
bool setParam(TemplateLibraryParam id, float value);
```

Parameter	Description
id	Parameter ID according to TemplateLibraryParam enum.
value	Parameter value. Value depends on parameter ID.

Returns: TRUE if the parameter was set or FALSE if not.

getParam method

getParam(...) method returns parameter value. Every implementation of library with parameters have to provide thread-safe **getParam(...)** method call. This means that the **getParam(...)** method can be safely called from any thread. Method declaration:

```
float getParam(TemplateLibraryParam id);
```

Parameter	Description
id	Parameter ID according to TemplateLibraryParam enum.

Returns: parameter value or **-1** if the parameters doesn't exist.

getParams method

getParams(...) method is designed to obtain params structure. Every implementation of library with parameters have to provide thread-safe **getParams(...)** method call. This means that the **getParams(...)** method can be safely called from any thread. Method declaration:

```
void getParams(TemplateLibraryParams& params);
```

Parameter	Description
params	Reference to TemplateLibraryParams object to store params.

doSomething method

doSomething(...) does something. Method declaration:

```
bool doSomething(int& value);
```

Parameter	Description
value	Reference to output value.

Returns: TRUE if the library did something or FALSE if not.

executeCommand method

executeCommand(...) method executes library command. Every implementation of library with parameters have to provide thread-safe **executeCommand(...)** method call. This means that the **executeCommand(...)** method can be safely called from any thread. Method declaration:

```
bool executeCommand(TemplateLibraryCommand id);
```

Parameter	Description
id	Command ID according to TemplateLibraryCommand enum.

Returns: TRUE if the command executed or FALSE if not.

Data structures

TemplateLibraryCommand enum

Enum declaration:

```
enum class TemplateLibraryCommand
{
    /// First command.
    FIRST_COMMAND = 1,
    /// Second command.
    SECOND_COMMAND,
    /// Third command.
    THIRD_COMMAND
};
```

Table 2 - Commands description.

Command	Description
FIRST_COMMAND	First command.
SECOND_COMMAND	Second command.
THIRD_COMMAND	Third command.

TemplateLibraryParam enum

Enum declaration:

```
enum class TemplateLibraryParam
{
    /// First param. Here describe nuances and param value vaid range.
    FIRST_PARAM = 1,
    /// Second param. Here describe nuances and param value vaid range.
    SECOND_PARAM,
    /// Third param. Here describe nuances and param value vaid range.
    THIRD_PARAM
};
```

Table 3 - Params description.

Parameter	Access	Description
FIRST_PARAM	read / write	First param. Valid values **0** or **1**: **0** - set to set firstParam of TempleLibraryParams class to **false**. **1** - set to set firstParam of TempleLibraryParams class to **true**.
SECOND_PARAM	read / write	Second param. Valid values from **-100** to **100**.
THIRD_PARAM	read / write	Third param. Valid values from **-100** to **100**.

TemplateLibraryParams class description

Class declaration

TemplateLibraryParams class is used to provide example params structure. Also **TemplateLibraryParams** provides possibility to write/read params from JSON files (**JSON_READABLE** macro) and provides methods to encode and decode params. **TemplateLibraryParams** interface class declared in **TemplateLibrary.h** file. Class declaration:

```
class TemplateLibraryParams
{
public:

    /// First param. Here describe what status does this flag define.
    bool firstParam{ false };
    /// Second param. Here describe nuances and param value vaid range.
    int secondParam{ 0 };
    /// Third param. Here describe nuances and param value vaid range.
    float thirdParam{ 0.0f };

    /// Macro from ConfigReader to make params readable/writable from JSON.
    JSON_READABLE(TemplateLibraryParams, firstParam, secondParam, thirdParam)

    /// operator =
    TemplateLibraryParams& operator= (const TemplateLibraryParams& src);

    /// Encode (serialize) params.
    bool encode(uint8_t* data, int bufferSize, int& size,
                TemplateLibraryParamsMask* mask = nullptr);

    /// Decode (deserialize) params.
    bool decode(uint8_t* data, int dataSize);
};
```

Table 4 - TemplateLibraryParams class fields description is related to TemplateLibraryParam enum description.

Field	type	Description
firstParam	bool	First template param.
secondParam	int	Second template param.
thirdParam	float	Third template param.

None: *TemplateParams class fields listed in Table 4* ***must*** *reflect params set/get by methods setParam(...) and getParam(...).*

Serialize template params

TemplateLibraryParams class provides method **encode(...)** to serialize template params. Serialization of template params necessary in case when you have to send template params via communication channels. Method provides options to exclude particular parameters from serialization. To do this method inserts binary mask (1 byte) where each bit represents particular parameter and **decode(...)** method recognizes it. Method declaration:

```
bool encode(uint8_t* data, int bufferSize, int& size, TemplateLibraryParamsMask* mask = nullptr);
```

Parameter	Value
data	Pointer to data buffer. Buffer size must be >= 237 bytes.
bufferSize	Data buffer size. Buffer size must be >= 237 bytes.
size	Size of encoded data.
mask	Parameters mask - pointer to **TemplateLibraryParamsMask** structure. **TemplateLibraryParamsMask** (declared in TemplateLibrary.h file) determines flags for each field (parameter) declared in TemplateLibraryParams class. If the user wants to exclude any parameters from serialization, he can put a pointer to the mask. If the user wants to exclude a particular parameter from serialization, he should set the corresponding flag in the **TemplateLibraryParamsMask** structure.

Returns: TRUE if params encoded (serialized) or FALSE if not.

TemplateLibraryParamsMask structure declaration:

```
struct TemplateLibraryParamsMask
{
    bool firstParam{ true };
    bool secondParam{ true };
    bool thirdParam{ true };
};
```

Example without parameters mask:

```
// Prepare random params.
cr::templ::TemplateLibraryParams params1;
params1.firstParam = true;
params1.secondParam = rand() % 255;
params1.thirdParam = static_cast<float>(rand() % 255);

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
params1.encode(buffer, bufferSize, size);
```

Example with parameters mask:

```
// Prepare random params.
cr::templ::TemplateLibraryParams params1;
params1.firstParam = true;
params1.secondParam = rand() % 255;
params1.thirdParam = static_cast<float>(rand() % 255);

// Prepare mask.
cr::templ::TemplateLibraryParamsMask mask;
mask.firstParam = false;
mask.secondParam = true; // Include only one param fr encoding.
mask.thirdParam = false;

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
params1.encode(buffer, bufferSize, size, &mask);
```

Deserialize template params

TemplateLibraryParams class provides method **decode(...)** to deserialize params. Deserialization of template params necessary in case when you need to receive params via communication channels. Method automatically recognizes which parameters were serialized by **encode(...)** method. Method declaration:

```
bool decode(uint8_t* data, int dataSize);
```

Parameter	Value
data	Pointer to data buffer with serialized params.
dataSize	Size of command data.

Returns: TRUE if params decoded (deserialized) or FALSE if not.

Example:

```
// Prepare random params.
cr::templ::TemplateLibraryParams params1;
params1.firstParam = true;
params1.secondParam = rand() % 255;
params1.thirdParam = static_cast<float>(rand() % 255);

// Encode (serialize) params.
int bufferSize = 128;
uint8_t buffer[128];
int size = 0;
params1.encode(buffer, bufferSize, size);

// Decode (deserialize) params.
cr::templ::TemplateLibraryParams params2;
params2.decode(buffer, size);
```

Read params from JSON file and write to JSON file

TemplateLibrary depends on open source **ConfigReader** library which provides method to read params from JSON file and to write params to JSON file. Example of writing and reading params to JSON file:

```
// Write params to file.
cr::utils::ConfigReader inConfig;
cr::templ::TemplateLibraryParams in;
inConfig.set(in, "templateParams");
inConfig.writeToFile("TestTemplateParams.json");

// Read params from file.
cr::utils::ConfigReader outConfig;
if(!outConfig.readFromFile("TestTemplateParams.json"))
{
    cout << "Can't open config file" << endl;
    return false;
}
```

TestTemplateParams.json will look like:

```
{
    "templateParams":
    {
        "firstParam": true,
        "secondParam": 1,
        "thirdParam": 0.5f
    }
}
```

Build and connect to your project

Typical commands to build **TemplateLibrary**:

```
git clone https://github.com/ConstantRobotics-Ltd/TemplateLibrary.git
cd TemplateLibrary
git submodule update --init --recursive
mkdir build
cd build
cmake ..
make
```

If you want connect **TemplateLibrary** to your CMake project as source code you can make follow. For example, if your repository has structure:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
```

You can add repository **TemplateLibrary** as submodule by commands:

```
cd <your respository folder>
git submodule add https://github.com/ConstantRobotics-Ltd/TemplateLibrary.git
3rdparty/TemplateLibrary
git submodule update --init --recursive
```

In you repository folder will be created folder **3rdparty/TemplateLibrary** which contains files of **TemplateLibrary** repository with subrepository **ConfigReader** and **ConfigReader**. New structure of your repository:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    TemplateLibrary
```

Create CMakeLists.txt file in **3rdparty** folder. CMakeLists.txt should contain:

```
cmake_minimum_required(VERSION 3.13)

#################################################################################
## 3RD-PARTY
## dependencies for the project
#################################################################################
project(3rdparty LANGUAGES CXX)

#################################################################################
## SETTINGS
## basic 3rd-party settings before use
#################################################################################
# To inherit the top-level architecture when the project is used as a submodule.
SET(PARENT ${PARENT}_YOUR_PROJECT_3RDPARTY)
# Disable self-overwriting of parameters inside included subdirectories.
SET(${PARENT}_SUBMODULE_CACHE_OVERWRITE OFF CACHE BOOL "" FORCE)

#################################################################################
## CONFIGURATION
## 3rd-party submodules configuration
#################################################################################
SET(${PARENT}_SUBMODULE_TEMPLATE_LIBRARY                ON  CACHE BOOL "" FORCE)
if (${PARENT}_SUBMODULE_TEMPLATE_LIBRARY)
    SET(${PARENT}_TEMPLATE_LIBRARY                      ON  CACHE BOOL "" FORCE)
    SET(${PARENT}_TEMPLATE_LIBRARY_TEST                 OFF CACHE BOOL "" FORCE)
    SET(${PARENT}_TEMPLATE_LIBRARY_EXAMPLE              OFF CACHE BOOL "" FORCE)
```

```
    SET(${PARENT}_TEMPLATE_LIBRARY_DEMO                 OFF CACHE BOOL "" FORCE)
endif()

################################################################################
## INCLUDING SUBDIRECTORIES
## Adding subdirectories according to the 3rd-party configuration
################################################################################
if (${PARENT}_SUBMODULE_TEMPLATE_LIBRARY)
    add_subdirectory(TemplateLibrary)
endif()
```

File **3rdparty/CMakeLists.txt** adds folder **TemplateLibrary** to your project and excludes test application and example (TemplateLibrary class test applications and example of custom TemplateLibrary class implementation) from compiling. Your repository new structure will be:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    CMakeLists.txt
    TemplateLibrary
```

Next you need include folder 3rdparty in main **CMakeLists.txt** file of your repository. Add string at the end of your main **CMakeLists.txt**:

```
add_subdirectory(3rdparty)
```

Next you have to include **TemplateLibrary** library in your **src/CMakeLists.txt** file:

```
target_link_libraries(${PROJECT_NAME} TemplateLibrary)
```

Done!

Example

Simple example shows how to use library.

```
#include <iostream>
#include "TemplateLibrary.h"

int main(void)
{
    // Create library object.
    cr::templ::TemplateLibrary lib;

    // Set params.
    lib.setParam(cr::templ::TemplateLibraryParam::FIRST_PARAM, 1.0f);
    lib.setParam(cr::templ::TemplateLibraryParam::SECOND_PARAM, 2.0f);
    lib.setParam(cr::templ::TemplateLibraryParam::THIRD_PARAM, 2.0f);
```

```
    // Display params.
    std::cout << lib.getParam(cr::templ::TemplateLibraryParam::FIRST_PARAM) << std::endl;
    std::cout << lib.getParam(cr::templ::TemplateLibraryParam::SECOND_PARAM) <<
std::endl;
    std::cout << lib.getParam(cr::templ::TemplateLibraryParam::THIRD_PARAM) << std::endl;

    // Execute commands.
    lib.executeCommand(cr::templ::TemplateLibraryCommand::FIRST_COMMAND);
    lib.executeCommand(cr::templ::TemplateLibraryCommand::SECOND_COMMAND);
    lib.executeCommand(cr::templ::TemplateLibraryCommand::THIRD_COMMAND);

    // Get params.
    cr::templ::TemplateLibraryParams params;
    lib.getParams(params);

    // Display params.
    std::cout << std::string(params.firstParam ? "true" : "false") << std::endl;
    std::cout << params.secondParam << std::endl;
    std::cout << params.thirdParam << std::endl;

    // Do something.
    std::cout << "Do something:" << std::endl;
    for (int i = 0; i < 10; ++i)
    {
        int value = 0;
        lib.doSomething(value);
        std::cout << "Value: " << value << std::endl;
    }

    return 0;
}
```