Guide to vp-cpp-template

Revision: **1** | Version: **159**

Document Status: **Public**

Revision History & Document Info

Rev	Date	Author(s)	ACT	Description
1	28 Oct 2022	VP	A A A A A A A A	About Design of the vp-cpp-template Adding CTypes example lib vkpLibs ► vkp::opmlCPP CI using GitLab local runners (Ubuntu) CI using GitHub local runners (Ubuntu) Creating and testing a new Module - Library OpenCV in vp-cpp-template

**__ACT__: Action(s) regarding the referred sections in the corresponding revisions, compared to their previous revision.*

- ***A**: <u>Added</u> - A new section has been added.*
- ***U**: <u>Updated</u> - A section has been updated (**Minor** modifications (M) and extended (E))*
  - ***E**: <u>Extended</u> - A section has only been extended with new information.*
  - ***M**: <u>Modified</u> - Important content from a section has been deprecated, and possible replaced.*
    - ***ME**: <u>Modified & Extended</u> - **Major** modifications (M) and extended (E).*
- ***D**: <u>Deleted</u> - A section has been removed from the document.*
  - ***O**: <u>Outdated</u> - A section is invalidated/outdated but still remains in the document for historical reasons or because it may contain some useful information (Use "~~Strike Trough~~" in section's Title).*

Abstract

Information document regarding the vp-cpp-template.

Author(s)

Initials	Name
VP	Vasileios Pothos

Table of Contents

ver: 159 -Modified: Oct 28, 2022, 22:54

Table of Contents

Issues Index

Figures Index

Tables Index

About

The **vp-cpp-template** can be cloned from:
https://github.com/terablade2001/vp-cpp-template

This document is referring to the branch and version **[master](0.050)**.
For previous versions look the commits history of the project.

1. What is vp-cpp-template?

It's a C++ & Python framework which includes my most important libraries (**CECS** / **CCEXP** / **MVECTOR** / **vkp_Config** / **PThreadPool** / **vkpLibs**), in a ready-to-be-compiled scheme, under two BuildSystems: **Windows** (*TDM-GCC*[1]) and **Ubuntu**.

You can use this ready-to-work-on framework with the combined aforementioned libraries, in order to setup and build quick different C++ / Python projects, for different platforms. These libraries can help you achieve different things, from integrated advanced error control, easy binary data importing/exporting for debuging and integration issues, easy access to configuration files where multiple parameters can be set during development/testing, easy access to thread-pooling mechanisms, CSV files read/write, e.t.c. Regarding the different platforms the **vp-cpp-template** project provides a **BuildSystems/** folder where developers can add specific **CMake** setups for different targeted platforms. By default the project provides **Windows** (TDM-GCC) and **Ubuntu** support.

The **vp-cpp-template** project also provides some technologies / know-how (*i.e. contains examples of APIs where the generated C/C++ dynamic files can be accessed via Python*) for the users. The file structure is designed to support easy implementation and testing of libraries, which mean it's possible to use the **vp-cpp-template** project to generate and test different libraries, which you can later use elsewhere as dynamic libraries.

The included libraries can be compiled directly in the project, providing some technologies, like integrated Error Control System, pthreads thread pool, checking and reading of configuration files easily, projects common build versioning between C++ and Python parts, data sharing via dynamic format, etc. The libraries are referred below.

- *CECS: C/C++ Error Control System*
- *PThreadPool: pthreads Threading Pool in C++*
- *vkpLibs: Utilities*
  - **vkpBuildVersioner** [C++, Python]: System for automated versioning in C++ and

1 https://jmeubank.github.io/tdm-gcc/download/

  Python.
  - **vkpConfigReader** [C++]: Utility to load multiple variables from configuration files directly to the wanted types.
  - **vkpProgressBar** [C++]: A class to create custom progress bars in the stdout, to monitor progress graphically.
  - **vkpCircularBuffer** [C++, Requires: CECS]: A very simple template-class for circular buffers where their index is treated alike infinite arrays.
  - **vkpTimer** [C++]: Classes for easily measuring and handling processing times between two points of code.
  - **vkpOpmlCpp** [C++]: Export data as strings to MindMap's opml format.
  - **vkpCSVHandler** [C++]: Load, handle and store CSV files in C++.
- *CCEXP: Utility for easy user-defined structured data recording/importing/sharing in C++ / MatLab / Python3*
- *MVECTOR: A modified partial <vector> library which also tracks the size of the allocated memory*

More information regarding the quick setup and build of the **vp-cpp-template** project, along with TODO list and Versioning can be found at the *README.md* file. This document aims to provide information regarding different specific aspects of the **vp-cpp-template** project.

Design of the vp-cpp-template

1. Folders structure

The structure of the folders of the **vp-cpp-template** project in version **(0.050)** is the following.

```
.
├── build // a temporal build where all intermediate build files via CMake are added.
├── BuildSystems // the folder where user can have multiple CMake / Batch commands for different platforms and OSs.
│   ├── my-ubuntu64 // Default provided build system for ubuntu. Can be used for guidance. | it contains files to build the [src/] files.
│   │   └── Libs // Additional libraries / module. Each one corresponds to a .so and can be build independently to work with the [src/] files.
│   │       ├── ExampleCAPI // Example library / module
│   │       ├── ExampleCTypes // Example library / module
│   │       └── vp-cpp-template-extlibs // Integrated vp-cpp-template libraries.
│   └── my-win64 // Default provided build system for windows (TDM-GCC). Can be used for guidance.
│       └── Libs // Additional libraries / module. Each one corresponds to a .dll and can be build independently to work with the [src/] files.
│           ├── ExampleCAPI
│           ├── ExampleCTypes
│           └── vp-cpp-template-extlibs
├── BuildVersion // Contains the BuildVersion.hpp file, which track the version of the project. Other (older) utilities are presented.
├── C++ // The folder where the C++ development is occuring.
│   ├── Common // Folder with common functions. These can be included in the 'vp-cpp-template-extlibs' build or elsewhere.
│   ├── Libs // Folder where the different Modules / Libs are developed. These can produce independent dynamic files (i.e, .dll, .so)
│   │   ├── ExampleCAPI // Example library / module
│   │   │   ├── API-Headers // (Optional) Contains the shared .hpp API headers (usually functions in C style, in `extern "C" {}` clauses)
│   │   │   ├── include // (Optional) Contains other internal .hpp headers
│   │   │   └── src // Contains the C++ codes of the module/library
│   │   ├── ExampleCTypes
│   │   │   ├── API-Headers
│   │   │   └── src
│   │   └── vp-cpp-template-extlibs
│   │       ├── API-Headers
│   │       └── src
│   └── src // Folder [src/] is for creating an executable (i.e. realease/testing app) using the developed libraries/modules of the [Libs/] folder
│       └── ModuleTests // This folder holds all C++ files that implement Unit/Functionl/Behavioral tests for the Modules/Libraries.
│           └── include // Contains the header filers of the ModuleTests codes.
├── ExtLibs // Folder containing external (default + user/custom) libraries that are compiled to the `vp-cpp-template-extlibs` module.
│   ├── CCEXP
│   ├── CECS
│   ├── MVECTOR
│   ├── PThreadPool
│   └── vkpLibs
├── python // Folder for testing python scripts / projects. Have access to all the C++ dynamic modules/libs and the app of the project.
│   └── BuildVersion
└── workdir // Folder for testing the C++ dynamic modules/libs and the app. Can be unlinked from Git and use Dvc to track in case of big-data.
    └── moduleTests // Containing the configuration files and binary data for all the Unit/Functionl/Behavioral tests.
```

The idea behind this file structure is that there is a unified common structure where the user can access via a git-clone command and implement for every new project he starts, *in-situ*:

- ✔ C++ libraries with C/C++ APIs in the ***C++/Libs/*** folder
- ✔ C/C++ applications in the ***C++/src/*** folder
- ✔ Unit/Functional/Behavioral tests in the ***C++/src/ModuleTests/*** folder

The user can also have in the ***ExtLibs/*** folder except from the **vp-cpp-template** provided libraries, also the selected libraries of his choice (*i.e. by forking the project and adding libraries according to his needs*).

2. Main() application and processes.

The **vp-cpp-template** project consists of a template, not only regarding it's files structure but also as an application. To this point, the ***main()*** function of the application, is designed to require as command-line input a text-based configuration file. This configuration file should always start with a tag `ProcessType:`. Based on this tag value the ***main()*** function selects different user-created functions – processes – to execute. By default in **vp-cpp-template** version (0.050) the following 4 processes are supported:

- **ModuleTesting**: Automatic text-file configurable multithreaded module testing of all provided test codes.
- **Example**: The default example process where different tools of the **vp-cpp-template** library are used as example. Code can be used as reference to see how they work.
- **TestExampleCAPI**: Access/Testing of the provided ***Libs/ExampleCAPI/*** library, as an example on how can someone create a library with C-API and access it.
- **TestVkpCSVHandler**: Testing the **vkpCSVHandler** tool for reading/writing data to CSV files.

User can modify the following part, to create his own processes, tests, applications based on the `ProcessType:` parameter in the given configuration file.

```
    if (0==processType.compare("ModuleTesting")) {
      _ERRT(0!=ModuleTesting(argc, argv),"Function \"ModuleTesting()\" failed!")
    } else if (0==processType.compare("Example")) {
      _ERRT(0!=Example(argc, argv),"Failed to run \"Example()\" function!")
    } else if (0==processType.compare("TestExampleCAPI")) {
      _ERRT(0!=TestExampleCAPI(argc, argv),"Failed to run \"TestExampleCAPI\" function!")
    } else if (0==processType.compare("TestVkpCSVHandler")) {
      _ERRT(0!=TestVkpCSVHandler(argc, argv),"Failed to run \"TestVkpCSVHandler\" function!")
    } else {
      _ERRSTR(1,{
        ss << "Unknown process type: [" << processType << "]" << endl;
        ss << "Valid Case-Sensitive process types are: "<<endl;
        ss << " - [ModuleTesting]" << endl;
        ss << " - [Example]" << endl;
        ss << " - [TestExampleCAPI]" << endl;
        ss << " - [TestVkpCSVHandler]" << endl;
      })
      _ERRT(1,"Abort due to unknown process type.")
    }
```

3. Automatic Modules Testing

The above approach can be used to create multiple tests regarding the libraries and the application that user may want to develop.

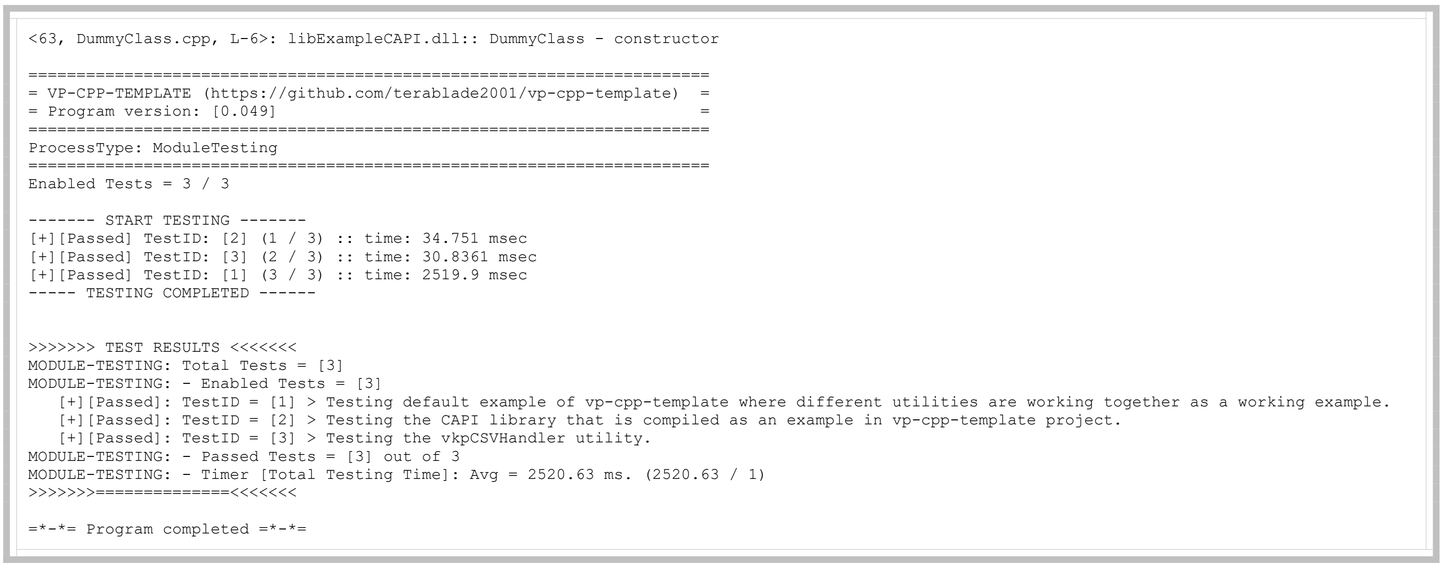
To this end, by using the **vp-cpp-template** library itself, in version (0.049) a process named `ProcessType: ModuleTesting` was introduced, with the corresponding process implemented in the ***C++/src/*ModuleTesting.cpp** file. For the **Ubuntu** setup the configuration file is:

confModuleTesting-Ubuntu.cfg

```
ProcessType: ModuleTesting

DbgVerboseLevel: 9

executableFile: ./vp-cpp-template
testSuccessCompletionString: =*-*= Program completed =*-*=

inParallelTests: 2
unitTestCSVFile: moduleTestsList.csv
displayPassedTestResults: false
displayFailedTestResults: false
logPassedOutputFile: moduleTestsLog_Passed.log
logFailedOutputFile: moduleTestsLog_Failed.log
```

As the generated application from **vp-cpp-template** can execute different processes based on the directives of the provided configuration files, this process is using pipes and threads to run multiple times the same application with different configuration files, thus running the different implemented processes / module tests. This approach eventually gives the opportunity to always have an application that will run and test all the selected module tests that the user wants (based on a CSV file, i.e. **moduleTestsList.csv**).

An example of the execution results where the process **ModuleTesting** is executing in 2 threads per time the three processes **Example**, **TestExampleCAPI** and **TestVkpCSVHandler** using the same compiled app, is shown below. *Passed* and *Failed* results are also stored by default to two different log files as they are defined in the configuration file.



```
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor

======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.049]                                           =
======================================================================
ProcessType: ModuleTesting
======================================================================
Enabled Tests = 3 / 3

------- START TESTING -------
[+][Passed] TestID: [2] (1 / 3) :: time: 34.751 msec
[+][Passed] TestID: [3] (2 / 3) :: time: 30.8361 msec
[+][Passed] TestID: [1] (3 / 3) :: time: 2519.9 msec
----- TESTING COMPLETED ------


>>>>>>> TEST RESULTS <<<<<<<
MODULE-TESTING: Total Tests = [3]
MODULE-TESTING: - Enabled Tests = [3]
   [+][Passed]: TestID = [1] > Testing default example of vp-cpp-template where different utilities are working together as a working example.
   [+][Passed]: TestID = [2] > Testing the CAPI library that is compiled as an example in vp-cpp-template project.
   [+][Passed]: TestID = [3] > Testing the vkpCSVHandler utility.
MODULE-TESTING: - Passed Tests = [3] out of 3
MODULE-TESTING: - Timer [Total Testing Time]: Avg = 2520.63 ms. (2520.63 / 1)
>>>>>>>=============<<<<<<<

=*-*= Program completed =*-*=
```

Adding CTypes example lib

CTypes[2] is a very useful tool that helps to run C/C++ compiled code through python.

The **vp-cpp-template** from the version **(0.037)**, does support such an example, which runs on Unix/Linux.

Here are the steps on how to test it. First clone and detach the project from the repository.

```
git clone https://github.com/terablade2001/vp-cpp-template
cd vp-cpp-template/BuildSystems/my-ubuntu64
./detachProjectFromGit.sh
```

Build the ExtLibs dynamic library (*required for **vp-cpp-template** projects*)

```
cd vp-cpp-template/BuildSystems/my-ubuntu64/Libs/vp-cpp-template-extlibs
./cmake-build.sh
./rmake-build.sh
```

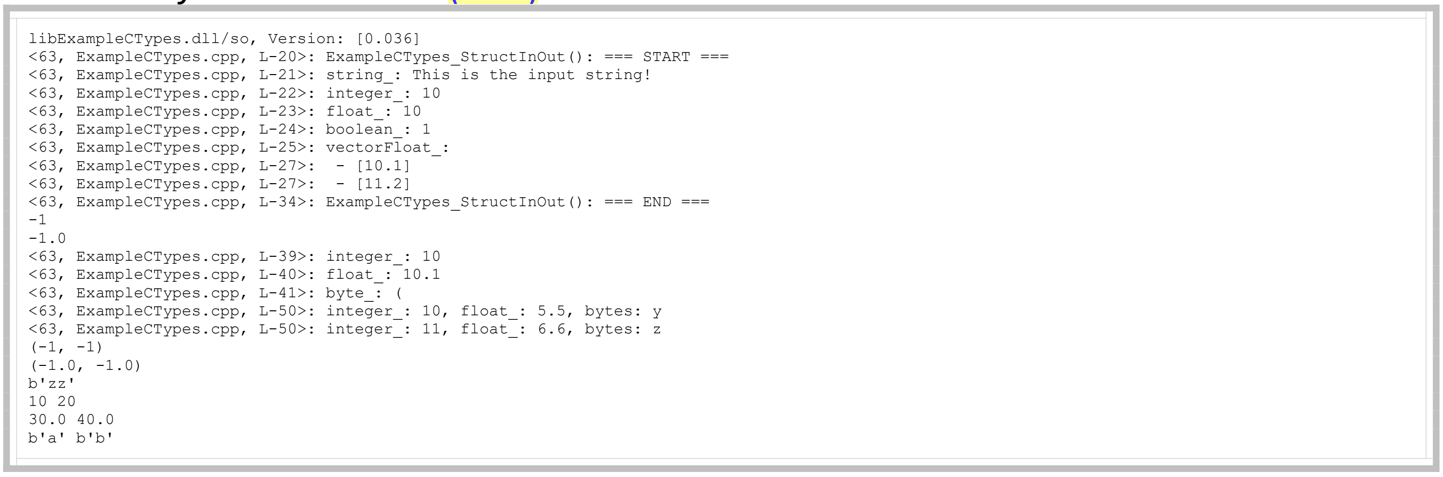
Build the dynamic library with the C-API for testing the python's CTypes.

```
cd ../ExampleCTypes
./cmake-build.sh
./rmake-build.sh
```

Now you can execute and test python with the previous C/C++ dynamic library.

```
cd vp-cpp-template/python
python3 ExampleCTypes.py
```

This will yield for version **(0.036)**:



```
libExampleCTypes.dll/so, Version: [0.036]
<63, ExampleCTypes.cpp, L-20>: ExampleCTypes_StructInOut(): === START ===
<63, ExampleCTypes.cpp, L-21>: string_: This is the input string!
<63, ExampleCTypes.cpp, L-22>: integer_: 10
<63, ExampleCTypes.cpp, L-23>: float_: 10
<63, ExampleCTypes.cpp, L-24>: boolean_: 1
<63, ExampleCTypes.cpp, L-25>: vectorFloat_:
<63, ExampleCTypes.cpp, L-27>:  - [10.1]
<63, ExampleCTypes.cpp, L-27>:  - [11.2]
<63, ExampleCTypes.cpp, L-34>: ExampleCTypes_StructInOut(): === END ===
-1
-1.0
<63, ExampleCTypes.cpp, L-39>: integer_: 10
<63, ExampleCTypes.cpp, L-40>: float_: 10.1
<63, ExampleCTypes.cpp, L-41>: byte_: (
<63, ExampleCTypes.cpp, L-50>: integer_: 10, float_: 5.5, bytes: y
<63, ExampleCTypes.cpp, L-50>: integer_: 11, float_: 6.6, bytes: z
(-1, -1)
(-1.0, -1.0)
b'zz'
10 20
30.0 40.0
b'a' b'b'
```

This example helps to understand:

- How to pass a structure with different data from python to C/C++.
- How to pass scalar values to C/C++
- How to pass arrays of different types allocated in python to C/C++ and get/read the modified results.
- How to get arrays of different types, allocated in the C/C++.

2 https://docs.python.org/3/library/ctypes.html

vkpLibs ► vkp::opmlCPP

The version (0.208) of the submodule **vkpLibs**[3] which is linked to the **vp-cpp-template** (0.037) contains a new class named as ***vkp***::***opmlCPP***. This class can be used for extracting nested information of code in a form of a **MindMap**[4]. Then using proper software it's possible to analyze the produced mind map and it's data interactively. For example, the following code:

```
vkp::opmlCPP opml("Debug","Debug.opml");
opml.add("Branch A");
opml.push("Branch B");
for (int i = 0; i < 5; i++) {
  opml.push(opml.s<<"Branch C-"<<i);
    for (int j = i; j < i+3; j++) {
      opml.add(opml.s<<"Value j = "<<j);
    }
  opml.pop();
}
opml.pop();
opml.add("Branch D");
```

produces a file **Debug.opml**, which when loaded, for instance, to the **SimpleMinds Pro**[5] software produce an interactive Mind-Map as shown in **Figure 1**. This can be useful in some cases where significant information can be kept hierarchical in multiple levels, and interactivity may be required to select properly the wanted data to further analyse them, in more depth, manually.

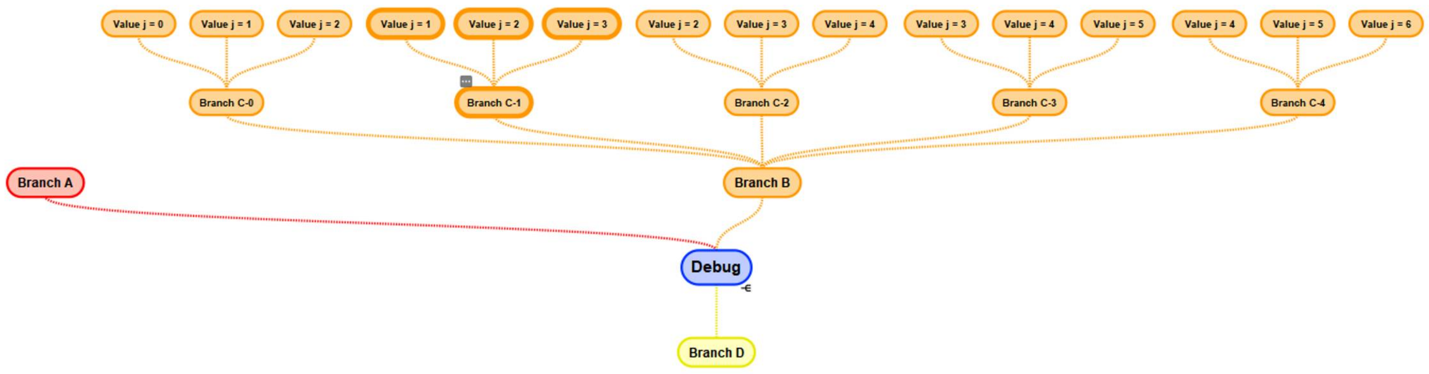


Figure 1: Example of debug information to interactive MindMap via vkp::opmlCPP.

3 https://github.com/terablade2001/vkpLibs
4 https://en.wikipedia.org/wiki/Mind_map
5 https://simplemind.eu/

CI using GitLab local runners (Ubuntu)

In this section information is provided on how to setup local GitLab runners in **Ubuntu** for Continuous Integration (CI) with **vp-cpp-template** version (0.051). This information consists of an *example / know-how* for the users that are interested to learning about CI and directly play/develop with it, using the **vp-cpp-template** project.

1. One Time setup

The steps that are described in this chapter should be done once per project and/or system in order to install proper software for testing.

1.1. Install GitLab runners on the system

Firstly the user has to install the GitLab runners on his **Ubuntu** system. For this reason the user need to have a GitLab[6] account. In order to install GitLab runners user can also find information at the following link: *https://docs.gitlab.com/runner/install/*.

The installation approach I followed is provided below, for an **Ubuntu** system:

```
dpkg -print-architecture
```

```
amd64
```

```
dpkg -i gitlab-runner_amd64.deb
```

```
...
```

```
gitlab-runner -v
```

```
Version:      15.4.0
Git revision: 43b2dc3d
Git branch:   15-4-stable
GO version:   go1.17.9
Built:        2022-09-20T22:38:36+0000
OS/Arch:      linux/amd64
```

6 *https://about.gitlab.com/*

1.2. Create a new project in GitLab, link runners, prepare docker.

At this point the user has to use the **vp-cpp-template** in order to instantiate a new working project in GitLab. The procedure is as follows.

In the GitLab webpage, create a new blank project. Select thus not to initialize it with README file or anything else. It should be completely blank.

When the project is created copy it's SHH cloning address i.e.:
git@gitlab.com:<username>/gitlab-testing.git

Now from the project's settings find out the field of "Specific runners" under this selection procedure: **Settings → CI/CD → Runners [Expand] → Specific runners**. There you can see and copy for later two things:

- The URL to register your runners. Typically is "*https://gitlab.com/*"
- A registration token that you have also to copy.

1.2.1. Connect the runners to the new project

From the **Ubuntu** terminal now use the command:

```
sudo gitlab-runner register
```

```
Enter the GitLab instance URL (for example, https://gitlab.com/):
```

Add the URL copied before. i.e. `https://gitlab.com/`

```
Enter the registration token:
```

Add the registration token copied before.

```
Enter a description for the runner:
```

Add a description for the runner, i.e. "vp-cpp-template testing"

```
Enter tags for the runner (comma-separated):
```

Add some tags for the runner. **For this example use** "`localrunner,ubuntu`".

```
Enter optional maintenance note for the runner:
```

Skip it (enter).

```
Registering runner... succeeded                    runner=************
Enter an executor: custom, docker, shell, ssh, kubernetes, docker-ssh, parallels, virtualbox, docker+machine, docker-ssh+machine:
```

For the executor selection select the "`shell`" executor. Type `shell` and enter.

```
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
```

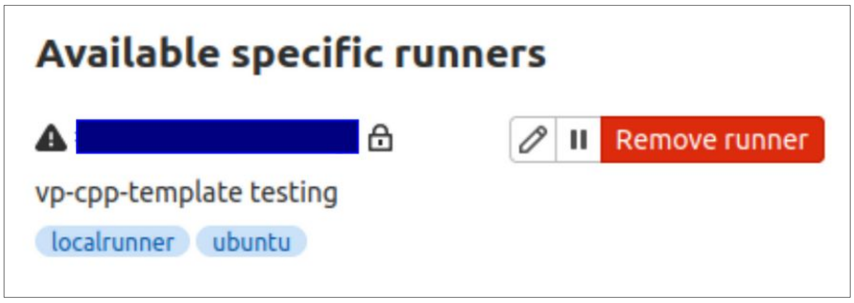
Restart the runner using:

```
sudo gitlab-runner restart
```

Now from the project's settings in GitLab website find again the field of "Specific runners" under this selection procedure: **Settings → CI/CD → Runners [Expand] → Specific**

runners.

If the procedure has been completed successfully then user will be able to see the runner available, having the description given above and the provided tags. The specific tags "`localrunner`" and "`ubuntu`" are used to select this runner via the default **vp-cpp-template** configuration.



Available specific runners

[redacted]

vp-cpp-template testing

1.2.2. Install and prepare docker images

In order to be able to use CI dockers are going to be included. For this reason the user has to install docker first - if not already installed – to his system. The procedure to install the docker software for **Ubuntu** is described in the following link: https://docs.docker.com/engine/install/ubuntu/

Here is one approach that can be also followed:

```
sudo apt-get update
sudo apt install docker.io
sudo systemctl status docker
sudo systemctl enable --now docker
sudo systemctl status docker
sudo usermod -aG docker $USER
```

Restart the system and then check the version:

```
docker --version
```

```
Docker version 20.10.12, build 20.10.12-0ubuntu4
```

Then add gitlab-runner to the docker's group:

```
sudo usermod -aG docker gitlab-runner
sudo service docker restart
```

1.2.3. Use vp-cpp-template as base for your new project.

Clone the **vp-cpp-template** project somewhere in your disk, and setup the template with the following commands:

```
git clone https://github.com/terablade2001/vp-cpp-template.git
cd vp-cpp-template/BuildSystems/my-ubuntu64/
./detachProjectFromGit.sh
```

Note: The **detachProjectFromGit.sh** script gathers all the requirements of the **vp-cpp-template** from other repositories and creates a new commit, while at the same time it removes the *git remote* link to the origin github repository. Thus after running it, the user have all the required files ready to be build, without any active *git remote*.

Now link the project with the new empty GitLab repository using:

```
git remote add origin git@gitlab.com:<username>/gitlab-testing.git
```

This **vp-cpp-template** cloning, detatching and gitlab attaching (adding git remote to gitlab repository) should be done for every new **vp-cpp-template** project the user may create.

1.2.4. Build the C++ base development docker image

Next step is to prepare the base C++ development base docker image which will be used as base for generating multiple other docker images later for testing. It is supposed at this point that there are no images available locally, thus the following command should return no images as is shown below.

```
sudo docker images
```

```
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

Use the command:

```
cd vp-cpp-template
sudo docker build --rm -f Dockerfile_cppdevbase -t cppdevbase .
```

This is expected to install an Ubuntu 22.04 image and then install required packages, as described in the **Dockerfile_cppdevbase** shown also below:

```
FROM ubuntu:22.04
RUN apt update
RUN apt install -y build-essential
RUN apt install -y cmake
RUN apt install -y python3
RUN apt -y install python3-pip
RUN python3 -m pip install numpy
```

After a successful completion new docker images will have been created, as the user can confirm using the following command:

```
sudo docker images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
cppdevbase   latest    cd9e881ba981   3 minutes ago   625MB
ubuntu       22.04     2dc39ba059dc   4 weeks ago     77.8MB
```

This docker images creation step is considered to have happened once, regardless the number of **vp-cpp-template** projects the user may create.

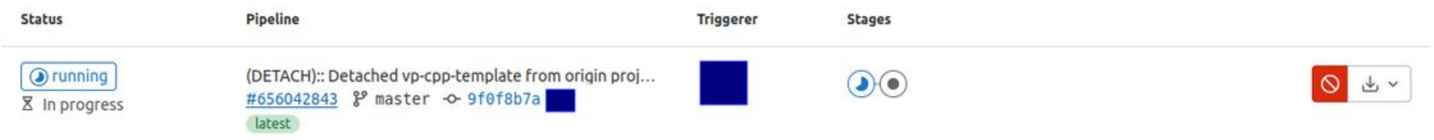
2. Push project to GitLab and confirm CI works.

The above steps of the previous Chapter 1 are expected to have happened once in the user's system.

Having setup the SHH keys in order the user to be able to access his account from **Ubuntu**'s shell, he can now push the project to GitLab.

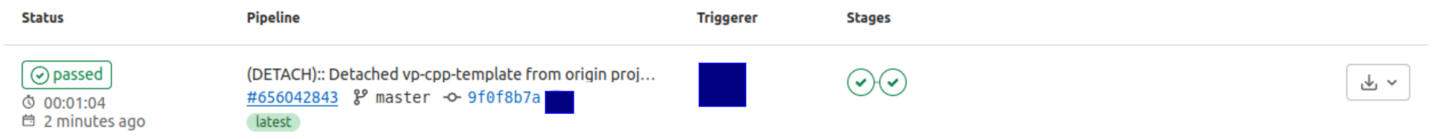
```
cd vp-cpp-template
git push origin master -u
```

From the GitLab project's webpage the user can go to the **`CI/CD [→ Pipelines]`** option where he will see a pipeline with two stages to exist. Given time, both stages will start running consecutively...



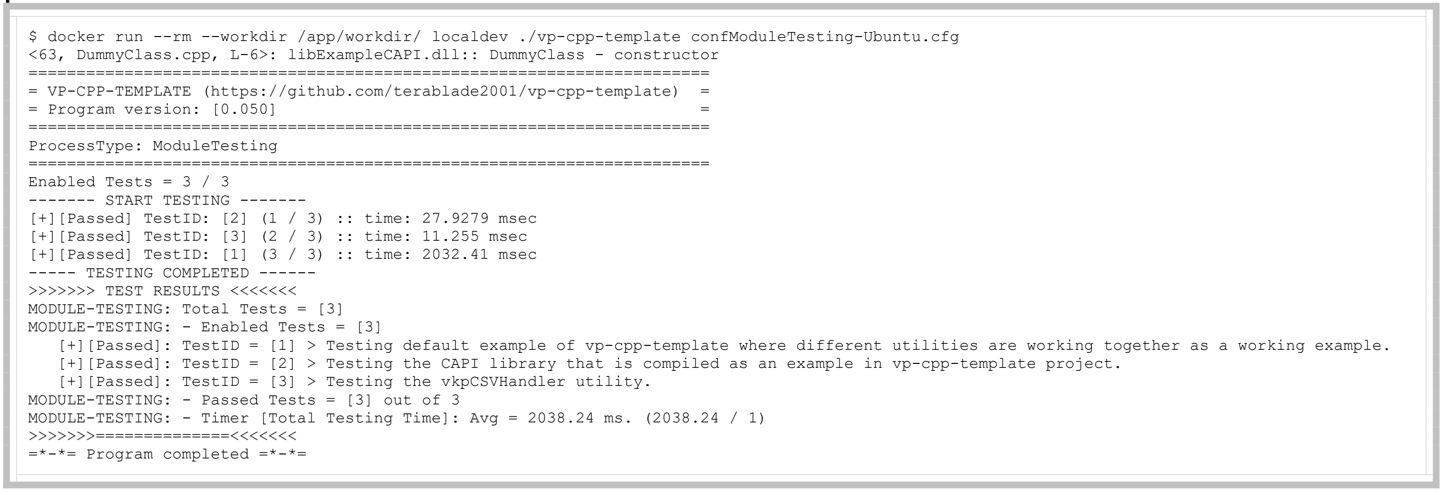
Status	Pipeline	Triggerer	Stages
running In progress	(DETACH):: Detached vp-cpp-template from origin proj... #656042843 master 9f0f8b7a [redacted] latest	[redacted]	

... and in the end both are expected to pass:



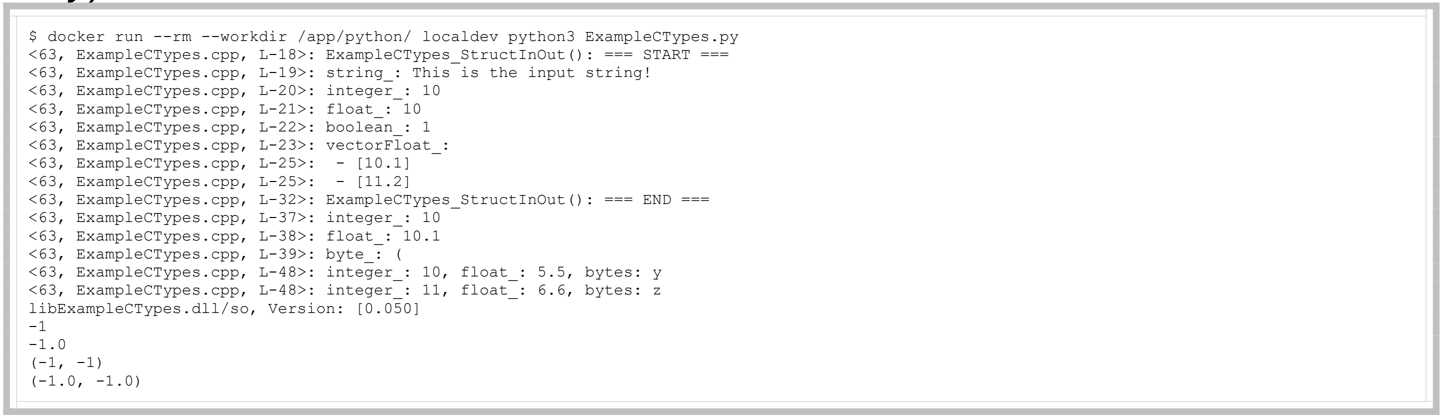
Status	Pipeline	Triggerer	Stages
passed 00:01:04 2 minutes ago	(DETACH):: Detached vp-cpp-template from origin proj... #656042843 master 9f0f8b7a [redacted] latest	[redacted]	✓ ✓

Checking the Test phase (stage #2) the user will see that all three Modules have passed:



```
$ docker run --rm --workdir /app/workdir/ localdev ./vp-cpp-template confModuleTesting-Ubuntu.cfg
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor
=======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.050]                                            =
=======================================================================
ProcessType: ModuleTesting
=======================================================================
Enabled Tests = 3 / 3
------- START TESTING -------
[+][Passed] TestID: [2] (1 / 3) :: time: 27.9279 msec
[+][Passed] TestID: [3] (2 / 3) :: time: 11.255 msec
[+][Passed] TestID: [1] (3 / 3) :: time: 2032.41 msec
----- TESTING COMPLETED ------
>>>>>>> TEST RESULTS <<<<<<<
MODULE-TESTING: Total Tests = [3]
MODULE-TESTING: - Enabled Tests = [3]
   [+][Passed]: TestID = [1] > Testing default example of vp-cpp-template where different utilities are working together as a working example.
   [+][Passed]: TestID = [2] > Testing the CAPI library that is compiled as an example in vp-cpp-template project.
   [+][Passed]: TestID = [3] > Testing the vkpCSVHandler utility.
MODULE-TESTING: - Passed Tests = [3] out of 3
MODULE-TESTING: - Timer [Total Testing Time]: Avg = 2038.24 ms. (2038.24 / 1)
>>>>>>>==============<<<<<<<
=*-*= Program completed =*-*=
```

He will also see the python test results as expected (*outside of the Module Testing utility*).



```
$ docker run --rm --workdir /app/python/ localdev python3 ExampleCTypes.py
<63, ExampleCTypes.cpp, L-18>: ExampleCTypes_StructInOut(): === START ===
<63, ExampleCTypes.cpp, L-19>: string_: This is the input string!
<63, ExampleCTypes.cpp, L-20>: integer_: 10
<63, ExampleCTypes.cpp, L-21>: float_: 10
<63, ExampleCTypes.cpp, L-22>: boolean_: 1
<63, ExampleCTypes.cpp, L-23>: vectorFloat_:
<63, ExampleCTypes.cpp, L-25>:  - [10.1]
<63, ExampleCTypes.cpp, L-25>:  - [11.2]
<63, ExampleCTypes.cpp, L-32>: ExampleCTypes_StructInOut(): === END ===
<63, ExampleCTypes.cpp, L-37>: integer_: 10
<63, ExampleCTypes.cpp, L-38>: float_: 10.1
<63, ExampleCTypes.cpp, L-39>: byte_: (
<63, ExampleCTypes.cpp, L-48>: integer_: 10, float_: 5.5, bytes: y
<63, ExampleCTypes.cpp, L-48>: integer_: 11, float_: 6.6, bytes: z
libExampleCTypes.dll/so, Version: [0.050]
-1
-1.0
(-1, -1)
(-1.0, -1.0)
```

```
b'zz'
10 20
30.0 40.0
b'a' b'b'
```

CI using GitHub local runners (Ubuntu)

In this section information is provided on how to setup local GitHub runners in **Ubuntu** for Continuous Integration (CI) with **vp-cpp-template** version (0.057). This information consists of an *example / know-how* for the users that are interested to learning about CI and directly play/develop with it, using the **vp-cpp-template** project.

For this step it is required to follow the docker instructions of sections **§Install and prepare docker images** and **§Build the C++ base development docker image**. To this point the docker images required for local testing will be available to your local machine.

1. Setup local runners for GitHub

Create a new GitHub project (private) and from there go: **Settings → Actions → Runners →New self-hosted runner**. From there select your setup i.e. **Linux** setup, and **x64** Architecture. GitHub will provide you with the command line commands, like:

```
# Create a folder
$ mkdir actions-runner && cd actions-runner
# Download the latest runner package
$ curl -o actions-runner-linux-x64-2.298.2.tar.gz -L https://github.com/actions/runner/releases/download/v2.298.2/actions-runner-linux-x64-
2.298.2.tar.gz
# Optional: Validate the hash
$ echo "0bfd792196ce0ec6f1c65d2a9ad00215b2926ef2c416b8d97615265194477117  actions-runner-linux-x64-2.298.2.tar.gz" | shasum -a 256 -c
# Extract the installer
$ tar xzf ./actions-runner-linux-x64-2.298.2.tar.gz
```

```
# Create the runner and start the configuration experience
$ ./config.sh --url https://github.com/terablade2001/testing --token <tokenID>
# Last step, run it!
$ ./run.sh
```

Keep this commands for later. Now, for the GitHub runners we are manually going to generate a new **Ubuntu** account, having "*sudo docker*" permissions and install the runners from there. The commands are as follows:

```
sudo adduser github-runner
sudo usermod -aG docker github-runner
sudo usermod -aG sudo github-runner
su github-runner
cd ~
pwd
```

```
/home/github-runner
```

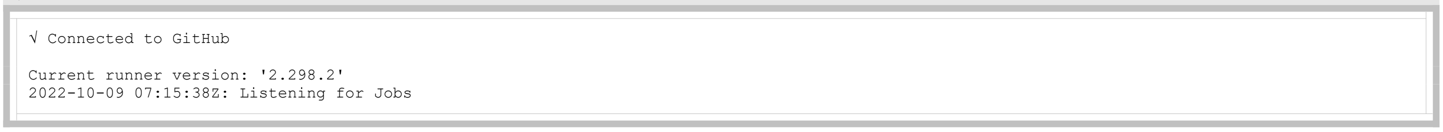
Now use the provided commands from the GitHub above:

```
mkdir actions-runner && cd actions-runner

curl -o actions-runner-linux-x64-2.298.2.tar.gz -L
https://github.com/actions/runner/releases/download/v2.298.2/actions-runner-linux-x64-2.298.2.tar.gz

echo "0bfd792196ce0ec6f1c65d2a9ad00215b2926ef2c416b8d97615265194477117  actions-runner-linux-x64-2.298.2.tar.gz"
| shasum -a 256 -c

tar xzf ./actions-runner-linux-x64-2.298.2.tar.gz

./config.sh --url https://github.com/terablade2001/testing --token <tokenID>
```

- ✔ `Enter the name of the runner group to add this runner to: [press Enter for Default]:` **press Enter**
- ✔ `Enter the name of runner: [press Enter for <name>]:` `localrunner-ubuntu` **enter**
- ✔ `This runner will have the following labels: 'self-hosted', 'Linux', 'X64'`
  `Enter any additional labels (ex. label-1,label-2): [press Enter to skip]`
  `localrunner,ubuntu` **enter**
- ✔ `Enter name of work folder: [press Enter for _work]` **enter**

The following step is to enable the runners thus to listen for new jobs. This has to be done manually (or can be set as a service / automatically).

```
./run.sh
```



```
√ Connected to GitHub

Current runner version: '2.298.2'
2022-10-09 07:15:38Z: Listening for Jobs
```

Now we are ready to use the **vp-cpp-template**.

2. Setup, build and test vp-cpp-template

Open a new terminal with your default account (*not the github-runner account*) and clone the **vp-cpp-template** project and detach it to create a new base project to work on:

```
git clone http://github.com/terablade2001/vp-cpp-template
cd vp-cpp-template/BuildSystems/my-ubuntu64/
./detachProjectFromGit.sh
```

Then push this project to your repository where you have add the runners.

```
git remote add origin git@github.com:<username>/testing.git
git push origin master -u
```

In your GitHub's repository, at the menu **Actions** you may see your workflow running, and after a while it should have been completed successfully!

Creating and testing a new Module - Library

This section provides a step by step example, on how the **vp-cpp-template** can be used to create and test a new C++ module – dynamic library incorporating the GitLab CI/CD technology as it has been introduced **above**. This example is for **Ubuntu** systems thus it's implied that you will have create a new project, and connect with it local runners with the tags `localrunner` and `ubuntu` (ref. §**1.2.1**). The example is valid for the version **(0.053)** of the **vp-cpp-template** library.

Note: In order to avoid creating every time a new GitLab project, just to test **vp-cpp-template**, you can do the procedure described at section §**CI using GitLab local runners (Ubuntu)** once. Then set this repository's master branch not to be protected anymore from the repository's **Settings → Repository → Protected branched** field. Afterwards, each time you start a new test you can use a forced pull to overwrite the previous repository in your project. This approach is going to be used in this example.

First prepare your new project by cloning the **vp-cpp-template** repository, detaching it from GitHub, connecting it to your GitLab's project and apply a CI/CD test, with the following commands:

```
git clone https://github.com/terablade2001/vp-cpp-template.git
cd vp-cpp-template/BuildSystems/my-ubuntu64/
./detachProjectFromGit.sh
git remote add origin git@gitlab.com:<username>/gitlab-testing.git
git push origin +master -u
```

Notice in the above commands, that I use forced push (*`+master`*) on the repository which I had create in section §**CI using GitLab local runners (Ubuntu)**. After a while, from the GitLab website a successful CI pass will have been completed.

The five above commands have created a new project to work on within my GitLab's repository. It now has no connection with the **vp-cpp-template** GitHub's website. Thus now the generated project is your own to develop on. In this example we are going to add a new Module/Dynamic Library based on this project. Let's go!

1. Setup the files for the new module/library

In order to add a new module/library, which we are going to call "**myNewModule**", we have to add files in three different parts of the base code. These are:

- ***BuildSystems/***: There we have to setup the build system for the new module/library. Also adjust the testing build system to incorporate the new module.
- ***C++/Libs/***: Add new folder ***myNewModule/***, in which the source codes of the new module / library will be placed.
- ***C++/src/ModuleTests/***: Add our module-testing source files (unit tests, etc) here.
- ***workdir/***: Add the new module tests in the **moduleTestsList.csv** list and provide the configuration file to run this test in the ***moduleTests/*** folder.

1.1. Setup the new build system files

Enter the folder ***BuildSystems/my-ubuntu64/Libs/*** and copy ***ExampleCAPI/*** folder to a new folder ***myNewModule/***.

```
cd BuildSystems/my-ubuntu64/Libs/
cp -r ExampleCAPI myNewModule
cd myNewModule
```

1.1.1. Update the myNewModule/ build files.

Now modify the three files inside the ***myNewModule/*** folder:

Edit the **cmake-build.sh** file and replace one line as shown below:

```
- export ProjectName="ExampleCAPI"
+ export ProjectName="myNewModule"
```

Edit the **rmake-build.sh** file and do the same modification:

```
- export ProjectName="ExampleCAPI"
+ export ProjectName="myNewModule"
```

Provide execution access to these new files via Git:

```
cd BuildSystems/my-ubuntu64/Libs/myNewModule
git add -- cmake-build.sh rmake-build.sh
git update-index --chmod=+x cmake-build.sh
git update-index --chmod=+x rmake-build.sh
```

Edit the **CMakeLists.txt** file with the following changes:

```
- set(PROJECTNAME "ExampleCAPI")
+ set(PROJECTNAME "myNewModule")

- "${BASE_PROJECT_FOLDER}/C++/Libs/ExampleCAPI/src/*.cpp"
+ "${BASE_PROJECT_FOLDER}/C++/Libs/myNewModule/src/*.cpp"
```

1.1.2. Setup application’s build files

Go to the folder ***BuildSystems/my-ubuntu64/*** and edit the **CMakeLists.txt** file as:

```
   add_library(example_c_api MODULE IMPORTED)
   set_target_properties(example_c_api PROPERTIES IMPORTED_LOCATION "${BASE_PROJECT_FOLDER}/workdir/libExampleCAPI.so")
+  add_library(myNewModule MODULE IMPORTED)
+  set_target_properties(myNewModule PROPERTIES IMPORTED_LOCATION "${BASE_PROJECT_FOLDER}/workdir/libmyNewModule.so")

   ${BASE_PROJECT_FOLDER}/C++/Libs/ExampleCAPI/API-Headers
+  ${BASE_PROJECT_FOLDER}/C++/Libs/myNewModule/API-Headers
   ${BASE_PROJECT_FOLDER}/C++/src/include

-  target_link_libraries(${PROJECTNAME} example_c_api vpCppTemplateExtLibs ${LINK_OPTIONS})
+  target_link_libraries(${PROJECTNAME} example_c_api myNewModule vpCppTemplateExtLibs ${LINK_OPTIONS})
```

This means that the compiled application (*which is actually implementing the testing application*) will also link with the new module’s dynamic library **libmyNewModule.so** and the app’s C++ code will have access to this library only via the header files provided at the ***C++/Libs/myNewModule/API-Headers/*** folder. These header files in the very end are expected to be shared for integration of the new library to other systems.

1.2. Include the build to GitLab's CI commands

Edit the file **Dockerfile_localDev** with the following changes:

```
   FROM cppdevbase:latest
   WORKDIR /app
   ADD . /app
   WORKDIR /app/BuildSystems/my-ubuntu64/Libs/vp-cpp-template-extlibs/
   RUN ./cmake-build.sh
   RUN ./rmake-build.sh
   WORKDIR /app/BuildSystems/my-ubuntu64/Libs/ExampleCAPI/
   RUN ./cmake-build.sh
   RUN ./rmake-build.sh
   WORKDIR /app/BuildSystems/my-ubuntu64/Libs/ExampleCTypes/
   RUN ./cmake-build.sh
   RUN ./rmake-build.sh
+  WORKDIR /app/BuildSystems/my-ubuntu64/Libs/myNewModule/
+  RUN ./cmake-build.sh
+  RUN ./rmake-build.sh
   WORKDIR /app/BuildSystems/my-ubuntu64/
   RUN ./cmake-build.sh
   RUN ./rmake-build.sh
   WORKDIR /app/workdir/
```

1.3. Create the new module’s source files

Go at ***C++/Libs/*** and copy ***ExampleCAPI/*** folder to ***myNewModule/*** folder. Rename the internal files appropriately a shown below.

```
cd C++/Libs
cp -r ExampleCAPI myNewModule
cd myNewModule/API-Headers
mv ExampleCAPI.hpp myNewModuleCAPI.hpp
cd ..
mv src/ExampleCAPI.cpp src/myNewModuleCAPI.cpp
```

As we don’t need the class ***DummyClass*** which is provided in the **ExampleCAPI** example library, just delete the corresponding references.

```
rm src/DummyClass.cpp
rm include/DummyClass.hpp
```

1.3.1. Edit the library's C-API header file

The header file for the new module is the ***C++/Libs/myNewModule/API-Headers/*****myNewModuleCAPI.hpp**. The changes we have to do are shown below. Actually we can replace all "**ExampleCAPI**" text references with "**myNewModule**" text. Then delete the custom functions of this example and add our new function ***myNewModule_ addPositiveNumbers()***. It should be noted that the names of the functions can be whatever you like, but I do use this convention, with the module's name first placed in the functions, as far as it concerns the shared C-API header.

```
   #pragma once
   #include <stdint.h>

   extern "C" {

   // --- User defined C-API functions ---
-  int ExampleCAPI_Initialize();
-  int ExampleCAPI_Process();
-  int ExampleCAPI_Shutdown();
+  int myNewModule_addPositiveNumbers(float a, float b, float* sumPtr);

   // --- Standard C-API functions ---
-  int ExampleCAPI_version(void** versionPtr); // Allocated in C, char**
-  int ExampleCAPI_error(void** errorPtr); // Allocated in C, char**
-  void ExampleCAPI_clearErrors();
-  void* ExampleCAPI_cecs();
-  int ExampleCAPI_setcecs(void* errorPtr);
+  int myNewModule_version(void** versionPtr); // Allocated in C, char**
+  int myNewModule_error(void** errorPtr); // Allocated in C, char**
+  void myNewModule_clearErrors();
+  void* myNewModule_cecs();
+  int myNewModule_setcecs(void* errorPtr);

}; // extern C
```

1.3.2. Add the source code of the library

The C++ source code for the new module can be added at the file ***C++/Libs/myNewModule/src/*****myNewModuleCAPI.cpp**. The changes that must be done are shown below.

```
   #include <Common.hpp>
-  #include "../API-Headers/ExampleCAPI.hpp" // CAPI functions for .dll use
+  #include "../API-Headers/myNewModuleCAPI.hpp"
-  #include "../include/DummyClass.hpp" // .dll internal objects

-  CECS_MODULE("ExampleCAPI")
+  CECS_MODULE("myNewModule")

   using namespace std;
   using namespace vkp;

   static vkpBuildVersioner BV1(1, VERSION_NUMBER);

-  static DummyClass dummyClass;

   extern "C" {

   // --- User defined C-API functions ---
-  int ExampleCAPI_Initialize() {
-    try {
-      dbg_(63,"ExampleCAPI - Initialize")
-    } catch (std::exception& e) { _ERRI(1,"Exception:\n[%s]") }
-    return 0;
-  }
-  int ExampleCAPI_Process() {
-    try {
-      dbg_(63,"ExampleCAPI - Process")
-      _ERRI(1,"Produced Error!")
-    } catch (std::exception& e) { _ERRI(1,"Exception:\n[%s]") }
-    return 0;
-  }
-  int ExampleCAPI_Shutdown() {
-    try {
-      dbg_(63,"ExampleCAPI - Shutdown")
-    } catch (std::exception& e) { _ERRI(1,"Exception:\n[%s]") }
-     return 0;
-   }
+   int myNewModule_addPositiveNumbers(float a, float b, float* sumPtr) {
+     _ERRINF(1,"On myNewModule_addPositiveNumbers()..")
+     _ERRI(sumPtr==nullptr,"Input sumPtr is nullptr!")
+     _ERRI(a < 0,"Input a (=%f) < 0. Must be a positive number or zero.",a)
+     _ERRI(b < 0,"Input b (=%f) < 0. Must be a positive number or zero.",b)
+     *sumPtr = a + b;
+     _ECSCLS(1)
+     return 0;
+   }

    // --- Standard C-API functions ---
-   int ExampleCAPI_version(void** versionPtr){
+   int myNewModule_version(void** versionPtr){
      _ERRI(nullptr == versionPtr,"NULL pointer provided!")
      *versionPtr = const_cast<char*>(BV1.version.c_str());
      return 0;
    }

-   int ExampleCAPI_error(void** errorPtr){
+   int myNewModule_error(void** errorPtr){
      if (errorPtr == NULL) return -1;
      *errorPtr = const_cast<char*>(__ECSOBJ__.str());
      if (_NERR_ != 0) return -1;
      return 0;
    }

-   void ExampleCAPI_clearErrors() {
+   void myNewModule_clearErrors() {
      _ECSCLS_
    }

-   void* ExampleCAPI_cecs() {
+   void* myNewModule_cecs() {
      return __ECSOBJ__.cecs();
    }

-   int ExampleCAPI_setcecs(void* errorPtr) {
+   int myNewModule_setcecs(void* errorPtr) {
      __ECSOBJ__.ConnectTo(errorPtr);
      return 0;
    }

    }; // extern "C"
```

The above changes include:

- Remove example's ***DummyClass*** references.
- Include the **myNewModule**'s header file
- Rename the CECS module (`CECS_MODULE("myNewModule")`)
- Remove the three user defined functions of the example code.
- Add the code for the new ***myNewModule_addPositiveNumbers()*** function.
- Rename all function names starting with **ExampleCAPI_** to **myNewModule_**

Our new function is provided below with some explanations.

```
1  int myNewModule_addPositiveNumbers(float a, float b, float* sumPtr) {
2    _ERRINF(1,"On myNewModule_addPositiveNumbers()..")
3    _ERRI(sumPtr==nullptr,"Input sumPtr is nullptr!")
4    _ERRI(a < 0,"Input a (=%f) < 0. Must be a positive number or zero.",a)
5    _ERRI(b < 0,"Input b (=%f) < 0. Must be a positive number or zero.",b)
6    *sumPtr = a + b;
7    _ECSCLS(1)
8    return 0;
9  }
```

Line 2: Set's an information log in the CECS error tracking mechanism, which will be provided if an error occurs. It's optional. This single log statement is cleared at line 7.

Lines 3-5: The ***_ERRI()*** macros will make checks in the first argument. If the checks are true then the following up messages will be logged and the function will return with a negative number.

Line 5: The summing operation.

Line 7: If the execution of the code reaches this point, then one log of the last logged errors will be cleared. Thus it's expected to clear the log set from Line 2. It's optional if Line 2 is also used.

Line 8: If everything is ok, the function will return **0** value.

At this point our **myNewModule** library has been implemented.
Next step is to test everything is OK when using it!

1.4. Generate a new test code file for the library.

Copy the file ***C++/src/ModuleTests/*TestExampleCAPI.hpp** to a new file ***C++/src/ModuleTests/*TestMyNewModule.cpp**.

```
cd C++/src/ModuleTests
cp TestExampleCAPI.cpp TestMyNewModule.cpp
```

Now edit it the **TestMyNewModule.cpp** thus to contain the following source code.

```
#include <Common.hpp>
#include <myNewModuleCAPI.hpp>

CECS_MODULE("Test_myNewModule")

using namespace std;
using namespace vkp;
using namespace vkpConfigReader;

static vector<float> vTestsValues;
class TestmyNewModuleConfigData : public vkpConfigReader::_baseDataLoader {
  public:
  int numberOfTests;
  std::string testsValues;
  std::vector<std::string>& getCheckParamList() {
    static std::vector<std::string> CheckParamList = {
      "numberOfTests", "testsValues"
    };
    return CheckParamList;
  }
  int loadDataSection(cfg_type& cfgData) {
    vkpConfigReaderLOADPARAM(numberOfTests)
    vkpConfigReaderLOADPARAM(testsValues)
    _ERRI(0!=cfg_convertToVector<float>(testsValues, vTestsValues),"Failed to convert testsValues: [%s] to array of values.",testsValues.c_str())
    return 0;
  }
};
static TestmyNewModuleConfigData confData;


int TestmyNewModule(int argc, char** argv) {
  char* moduleErrorStr;
  _ERRI(0!=myNewModule_error((void**)&moduleErrorStr),"myNewModule has recorded errors:\n---\n%s\n---\n",moduleErrorStr)
  myNewModule_setcecs(__ECSOBJ__.cecs());

  _ERRI(0!=confData.loadConfigFile(argv[1]),"Failed to load config file [%s]",argv[1])
  _ERRI((size_t)confData.numberOfTests != vTestsValues.size()/3,"Number of Tests (%i) are not matching the number of test values (%zu/3).",confData.numberOfTests, vTestsValues.size())

  // Get version of the
  char* moduleVersion;
  _ERRI(0!=myNewModule_version((void**)&moduleVersion),"Failed to get module's version!")
  std::cout << "= Library [myNewModule.so]: Version: " <<moduleVersion<<std::endl;

  try {
    for (int testIdx = 0; testIdx < (const int)vTestsValues.size(); testIdx+=3) {
      const float ka = vTestsValues[testIdx];
      const float kb = vTestsValues[testIdx+1];
      const float kexpectedSum = vTestsValues[testIdx+2];
      float sumResult;
      _ERRI(0!=myNewModule_addPositiveNumbers(ka, kb, &sumResult), "Failed to calculate the sum of positive numbers")
      _ERRI(sumResult != kexpectedSum,"Calculated result (=%f) != expected result (=%f)",sumResult, kexpectedSum)
    }
  } catch(std::exception &e) { _ERRI(1,"Exception captured:\n%s",e.what()) }

  return 0;
}
```

Let's explain it.

Line 1: Include all relative headers in order to use the **vp-cpp-template**.
Line 2: Include the C-API headers for our new module.
Line 4: Set the name for the error control system (CECS[7]) of **vp-cpp-template**, regarding this module.
Lines 10: Declare a vector which will store our test's values. It will be discussed later.
Lines 11-28: Create the implementation class of a ***_baseDataLoader*** for the specific test. This class loads data of different basic types directly from configuration text-files. Then it can be used as a class for providing parameters to our program.

Lines 13-14: We define two types of data. An integer and a string. These are the external parameters of this test application / process.

Lines 16-18: We provide the strings that are required to exist in the configuration file as tags. For instance, the configuration file must contain the "`numberOfTests:`" and "`testsValues:`" labels. The names must be identical to the variables used in lines 13-14.

Lines 22-23: Using the macro ***vkpConfigReaderLOADPARAM()*** the specific parameters of the class are loaded in proper type from the configuration file's corresponding tags.

Line 24: Converting the string value of `testsValue` to a vector of floating point numbers. The string should be formatted with values separated via commas, like: `testsValues: 1,1,2,3,3,6,21,21,42`. This will produce a vector with the 9 floating point numbers, stored at the vector of Line 10.

Line 28: `confData` will be the static object that will load the configuration file and provide us access to the external (*configuration file*) parameters of the test.
Line 31: The function ***TestmyNewModule()*** will be called from ***main()*** with the same arguments, if this process is selected by ***main()***, based on the application's input configuration file.
Lines 32-33: This is a check if there are errors recorded in the library, during the construction phase (*i.e. constructor calls etc*). The default C-API function ***myNewModule_error()*** is used for checking errors of this library.
Line 34: If no errors have occurred during the construction of the library, the function ***myNewModule_setcecs()*** redirects our **myNewModule** CECS system to use the application's CECS system (*`__ECSOBJ__` object*) for error reporting. This way the error reporting systems among the two modules, the application and the library, are integrated to one single error-report.
Line 36: Our configuration object `confData` loads the provided configuration file, to acquire the test's parameters.
Line 37: The test parameter `confData.numberOfTests` must equal to `vTestsValues.size()/3`. That is for every test the vector `vTestsValues` should contain 3 values: The \`a\` value, the \`b\` value and the expected sum value of the \`a+b\` operation. Thus for 3 tests, it should contain 9 values. If that's not true we get error and the test-program aborts.
Lines 40-42: Example of how to acquire and print our new Library's version.
Lines 45-48: For each test we acquire the 3 test values.

7 *https://github.com/terablade2001/CECS*

Line 50: For each test we insert the \`a\` and \`b\` test values to our function ***myNewModule_addPositiveNumbers()*** and get the result. At the same time with the macro ***_ERRI()*** we check if errors have been recorded after this function call, and if yes we record a new error and return.

Line 51: We check if the result of the tested function ***myNewModule_addPositiveNumbers()*** is the same as the one which we are expecting from the configuration file parameters. If not we log an error and return a negative number representing failure of the function call.

Line 55: If all checks (using CECS's *_ERRI()* macros) have been completed successfully then return success code **0**.

That's it!

1.5. Add the new test to the main() call

Edit the ***C++/src/*****Main.cpp** file with the following changes, in order to incorporate the **above** code as a testing process in the executable application:

```
   CECS_MAIN_MODULE("Main","CECS::Project")

   int ModuleTesting(int argc, char** argv);
+  int TestmyNewModule(int argc, char** argv);
...
...
...
   if (0==processType.compare("ModuleTesting")) {
     _ERRT(0!=ModuleTesting(argc, argv),"Function \"ModuleTesting()\" failed!")
   } else if (0==processType.compare("Example")) {
     _ERRT(0!=Example(argc, argv),"Failed to run \"Example()\" function!")
   } else if (0==processType.compare("TestExampleCAPI")) {
     _ERRT(0!=TestExampleCAPI(argc, argv),"Failed to run \"TestExampleCAPI\" function!")
   } else if (0==processType.compare("TestVkpCSVHandler")) {
     _ERRT(0!=TestVkpCSVHandler(argc, argv),"Failed to run \"TestVkpCSVHandler\" function!")
+  } else if (0==processType.compare("TestmyNewModule")) {
+    _ERRT(0!=TestmyNewModule(argc, argv),"Failed to run \"TestmyNewModule\" function!")
   } else {
     _ERRSTR(1,{
       ss << "Unknown process type: [" << processType << "]" << endl;
       ss << "Valid Case-Sensitive process types are: "<<endl;
       ss << " - [ModuleTesting]" << endl;
       ss << " - [Example]" << endl;
       ss << " - [TestExampleCAPI]" << endl;
       ss << " - [TestVkpCSVHandler]" << endl;
+      ss << " - [TestmyNewModule]" << endl;
     })
     _ERRT(1,"Abort due to unknown process type.")
   }
```

2. Set the testing files in folder workdir/

In order to test the new module-library two things are now required.

- Modify the CSV file ***workdir/*****moduleTestsList.csv** to include one more test:

```
   "Test ID" " Enabled" "Result Status" "Configuration File" Description
   1 1 0 moduleTests/confExample.cfg "Testing default example of vp-cpp-template where different utilities are working together as a working
example."
   2 1 0 moduleTests/confTestExampleCAPI.cfg "Testing the CAPI library that is compiled as an example in vp-cpp-template project."
   3 1 1 moduleTests/confTestVkpCSVHandler.cfg "Testing the vkpCSVHandler utility."
+  4 1 1 moduleTests/confTestmyNewModule.cfg "Testing the new module 'myNewModule'"
```

This change actually means that we add a 4th enabled test, which uses the configuration file at ***workdir/moduleTests/*****confTestmyNewModule.cfg** and has also it's description. This test is designed to not produce any errors ("Result Status == 1"). This can be regarded as a "positive pass" test, where the test passes because it yield no errors as it is expected. ***Note:*** *The first two tests are designed to produce errors ("Result Status == 0") thus in order the tests to Pass they must produce errors. In this case, the first two tests can be regarded as "negative pass" tests, because they pass with errors as expected.*

- Provide the configuration file in order to run the test as implemented in **§1.4** above. Create a file ***workdir/moduleTests/*****confTestmyNewModule.cfg** with the following data:

```
+  ProcessType: TestmyNewModule
+
+  # Three tests
+  numberOfTests: 3
+
+  # Three triplets of [a, b, expectedSum]...
+  testsValues: 1,1,2,3,3,6,21,21,42
```

3. Test with GitLab's CI

It's proposed to update the build version number from ***BuildVersion/*****BuildVersion.hpp** file, by just editing it.

To review the changes use:

```
git status
```

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   BuildSystems/my-ubuntu64/Libs/myNewModule/cmake-build.sh
        new file:   BuildSystems/my-ubuntu64/Libs/myNewModule/rmake-build.sh

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   BuildSystems/my-ubuntu64/CMakeLists.txt
        modified:   BuildVersion/BuildVersion.hpp
        modified:   C++/src/Main.cpp
        modified:   Dockerfile_localDev
        modified:   workdir/moduleTestsList.csv

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        BuildSystems/my-ubuntu64/Libs/myNewModule/CMakeLists.txt
        C++/Libs/myNewModule/
        C++/src/ModuleTests/TestMyNewModule.cpp
        workdir/moduleTests/confTestmyNewModule.cfg
```

Then commit all the changes and push to GitLab (*forced push for this example*). For instance:

```
git add --all
git commit -m "(0.053.01): Adding myNewModule"
git push origin +master -u
```

Now, wait for the GitLab CI to complete the build and testing phase. If all have been correctly implemented a similar to the following result should be available in the **passed** test phase. According to this result our new Module/library, it's implemented function and it's testing code have all been passed successfully!

```
$ docker rm moduleTesting || true && docker run --name moduleTesting --workdir /app/workdir/ localdev ./vp-cpp-template confModuleTesting-
Ubuntu.cfg
moduleTesting
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor
======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.053]                                            =
======================================================================
ProcessType: ModuleTesting
======================================================================
Enabled Tests = 4 / 4
------- START TESTING -------
[+][Passed] TestID: [2] (1 / 4) :: time: 19.419 msec
[+][Passed] TestID: [3] (2 / 4) :: time: 19.182 msec
[+][Passed] TestID: [4] (3 / 4) :: time: 14.6849 msec
[+][Passed] TestID: [1] (4 / 4) :: time: 2031.11 msec
----- TESTING COMPLETED ------
>>>>>>> TEST RESULTS <<<<<<<
MODULE-TESTING: Total Tests = [4]
MODULE-TESTING: - Enabled Tests = [4]
   [+][Passed]: TestID = [1] > Testing default example of vp-cpp-template where different utilities are working together as a working example.
   [+][Passed]: TestID = [2] > Testing the CAPI library that is compiled as an example in vp-cpp-template project.
   [+][Passed]: TestID = [3] > Testing the vkpCSVHandler utility.
   [+][Passed]: TestID = [4] > Testing the new module 'myNewModule'
MODULE-TESTING: - Passed Tests = [4] out of 4
MODULE-TESTING: - Timer [Total Testing Time]: Avg = 2031.65 ms. (2031.65 / 1)
>>>>>>>=============<<<<<<<
=*-*= Program completed =*-*=
```

Great!

3.1. Simulate an erroneous behavior for CI

In this case we want a "negative pass" test. We want to make a test for our function, which should produce expected errors. This can achieved if we provide a negative number to one of it's input arguments. Without changing the code we can directly simulate the test case using a new configuration file and updated the CSV list with a new test:

➢ Modify the CSV file ***workdir/*****moduleTestsList.csv** to include one more test.

```
   "Test ID" " Enabled" "Result Status" "Configuration File" Description
   1 1 0 moduleTests/confExample.cfg "Testing default example of vp-cpp-template where different utilities are working together as a working
example."
   2 1 0 moduleTests/confTestExampleCAPI.cfg "Testing the CAPI library that is compiled as an example in vp-cpp-template project."
   3 1 1 moduleTests/confTestVkpCSVHandler.cfg "Testing the vkpCSVHandler utility."
   4 1 1 moduleTests/confTestmyNewModule.cfg "Testing the new module 'myNewModule'"
+  5 1 0 moduleTests/confTestmyNewModule-2.cfg "Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!"
```

➢ Add the configuration file in order to run the test as implemented in §1.4 above. Create a file ***workdir/moduleTests/*****confTestmyNewModule-2.cfg** with the following data:

```
cd workdir/moduleTests
cp confTestmyNewModule.cfg confTestmyNewModule-2.cfg
gedit confTestmyNewModule-2.cfg
```

```
   ProcessType: TestmyNewModule

   # Three tests
   numberOfTests: 3

   # Three triplets of [a, b, expectedSum]...
-  testsValues: 1,1,2,3,3,6,21,21,42
+  testsValues: 1,1,2,3,3,6,-21,21,42
```

```
git add --all
git commit -m "(0.053.02): Adding myNewModule and testing with negative argument."
git push origin +master -u
```

The GitLab CI will pass this run too. From the log of the Test phase we can see that all 5 tests have now passed.

```
$ docker rm moduleTesting || true && docker run --name moduleTesting --workdir /app/workdir/ localdev ./vp-cpp-template confModuleTesting-
Ubuntu.cfg
moduleTesting
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor
======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.053]                                           =
======================================================================
ProcessType: ModuleTesting
======================================================================
Enabled Tests = 5 / 5
------- START TESTING -------
[+][Passed] TestID: [2] (1 / 5) :: time: 33.1511 msec
[+][Passed] TestID: [3] (2 / 5) :: time: 18.224 msec
[+][Passed] TestID: [4] (3 / 5) :: time: 18.3921 msec
[+][Passed] TestID: [5] (4 / 5) :: time: 14.0991 msec
[+][Passed] TestID: [1] (5 / 5) :: time: 2040.6 msec
----- TESTING COMPLETED ------
>>>>>>> TEST RESULTS <<<<<<<
MODULE-TESTING: Total Tests = [5]
MODULE-TESTING: - Enabled Tests = [5]
   [+][Passed]: TestID = [1] > Testing default example of vp-cpp-template where different utilities are working together as a working example.
   [+][Passed]: TestID = [2] > Testing the CAPI library that is compiled as an example in vp-cpp-template project.
   [+][Passed]: TestID = [3] > Testing the vkpCSVHandler utility.
   [+][Passed]: TestID = [4] > Testing the new module 'myNewModule'
   [+][Passed]: TestID = [5] > Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!
MODULE-TESTING: - Passed Tests = [5] out of 5
MODULE-TESTING: - Timer [Total Testing Time]: Avg = 2040.72 ms. (2040.72 / 1)
>>>>>>>==============<<<<<<<
=*-*= Program completed =*-*=
```

3.1.1. Review the tests results

In order to check the generated results use the following commands in your local **Ubuntu** system. This way you can access the latest running docker container which was used for the last testing:

```
sudo docker commit moduleTesting temp
sudo docker run -it --entrypoint sh temp
cd /app/workdir
cat moduleTestsLog_Passed.log
cat moduleTestsLog_Failed.log
exit
sudo docker container prune
sudo docker rmi temp
```

Below are the corresponding CECS recorded results from the 5th test, which was designed to fail in order to pass the modules testing.

```
==============================================================
[+][Passed]: TestID = [5] > Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!
   ----------------------------------------------------------------------------------------------------------------------------
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor

======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.053]                                           =
======================================================================
ProcessType: TestmyNewModule
======================================================================
= Library [myNewModule.so]: Version: [0.053]

(*) Exception occurred:
-------------------------------------------------------
::    CECS (C/C++ Error Control System) v[0.131]     ::
::        www.github.com/terablade2001/CECS          ::
-------------------------------------------------------
======= (CECS::Project):: [4] Record(s) of ALL Types recorded! =======
= [ERRINF ]> #myNewModule: 15 |> On myNewModule_addPositiveNumbers()..
= [ERROR  ]> #myNewModule: 17 |> Input a (=-21.000000) < 0. Must be a positive number or zero.
= [ERROR  ]> #Test_myNewModule: 50 |> Failed to calculate the sum of positive numbers
= [ERROR  ]> #Main: 73 |> Failed to run "TestmyNewModule" function!
=======================================================
```

And for comparison below are the results from the test No 4.

```
==============================================================
[+][Passed]: TestID = [4] > Testing the new module 'myNewModule'
   ----------------------------------------------------------------------------------------------------------------------------
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor
```

```
======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.053]                                            =
======================================================================
ProcessType: TestmyNewModule
======================================================================
= Library [myNewModule.so]: Version: [0.053]
=*-*= Program completed =*-*=
```

Everything is as expected!

Test 4 passed because it had no errors, and test 5 passed because it had expected errors!

3.1.2. Enforce wrong behavior to Fail the CI

In this final test we are going to provide wrong test data. As far as our function works as expected, the automated CI should fail! Edit the ***workdir/*****moduleTestsList.csv** thus the last test to be considered as a test that should not produce errors:

```
-  5 1 0 moduleTests/confTestmyNewModule-2.cfg "Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!"
+  5 1 1 moduleTests/confTestmyNewModule-2.cfg "Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!"
```

```
git add --all
git commit -m "(0.053.03): Adding myNewModule and testing with negative argument but accepted result"
git push origin +master -u
```

In this case the GitLab CI/CD pipeline run will fail, and the failed Test phase log contains the following:

```
$ docker rm moduleTesting || true && docker run --name moduleTesting --workdir /app/workdir/ localdev ./vp-cpp-template confModuleTesting-
Ubuntu.cfg
Error: No such container: moduleTesting
<63, DummyClass.cpp, L-6>: libExampleCAPI.dll:: DummyClass - constructor
======================================================================
= VP-CPP-TEMPLATE (https://github.com/terablade2001/vp-cpp-template)  =
= Program version: [0.053]                                            =
======================================================================
ProcessType: ModuleTesting
======================================================================
Enabled Tests = 5 / 5
------- START TESTING -------
[+][Passed] TestID: [2] (1 / 5) :: time: 22.778 msec
[+][Passed] TestID: [3] (2 / 5) :: time: 11.996 msec
[+][Passed] TestID: [4] (3 / 5) :: time: 8.42309 msec
[-][Failed] TestID: [5] (4 / 5) :: time: 7.96199 msec
[+][Passed] TestID: [1] (5 / 5) :: time: 2021.59 msec
----- TESTING COMPLETED ------
>>>>>>> TEST RESULTS <<<<<<<
MODULE-TESTING: Total Tests = [5]
MODULE-TESTING: - Enabled Tests = [5]
   [+][Passed]: TestID = [1] > Testing default example of vp-cpp-template where different utilities are working together as a working example.
   [+][Passed]: TestID = [2] > Testing the CAPI library that is compiled as an example in vp-cpp-template project.
   [+][Passed]: TestID = [3] > Testing the vkpCSVHandler utility.
   [+][Passed]: TestID = [4] > Testing the new module 'myNewModule'
   [-][Failed]: TestID = [5] > Testing the new module 'myNewModule' with one negative argument. Expecting a failure to Pass!
MODULE-TESTING: - Passed Tests = [4] out of 5
MODULE-TESTING: - Failed Tests = [1] out of 5
MODULE-TESTING: - Timer [Total Testing Time]: Avg = 2021.76 ms. (2021.76 / 1)
>>>>>>>=============<<<<<<<
(*) Exception occurred:
-------------------------------------------------------
::    CECS (C/C++ Error Control System) v[0.131]     ::
::        www.github.com/terablade2001/CECS          ::
-------------------------------------------------------
======= (CECS::Project):: [2] Record(s) of ALL Types recorded! =======
= [ERROR  ]> #ModuleTesting: 406 |> Tests Failed --> 1 / 5
= [ERROR  ]> #Main: 65 |> Function "ModuleTesting()" failed!
====================================
```

OpenCV in vp-cpp-template

1. Clone and Build OpenCV (Ubuntu)

Supposing we want to Build the OpenCV 4.1.0 then follow the procedure:

```
git clone https://github.com/opencv/opencv
git clone https://github.com/opencv/opencv_contrib
cd opencv
git checkout 4.1.0
```

Then create and modify accordingly to your needs the **build.sh** file provided below. Replace the **<openCV-Cloned-Path>** tag in the code below, with the path of the repositories. Place this file at the ***opencv/*** folder and give it execution rights with **chmod** command. Execute it.

```
#!/bin/sh
clear
rm -rf build
mkdir build
cd build

cmake -G "Unix Makefiles" \
  -DBUILD_SHARED_LIBS=ON \
  -DCMAKE_INSTALL_PREFIX=/usr/local/opencv-4.1.0 \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_CONFIGURATION_TYPES=Release \
  -DOPENCV_GENERATE_PKGCONFIG=ON \
  -DCMAKE_CXX_STANDARD=11 \
  -DCMAKE_CXX_STANDARD_REQUIRED=TRUE \
  -DGLIBCXX_USE_CXX11_ABI=0 \
  -DBUILD_DOCS=OFF \
  -DBUILD_PERF_TESTS=OFF \
  -DBUILD_TESTS=OFF \
  -DBUILD_FAT_JAVA_LIB=OFF \
  -DBUILD_CUDA_STUBS=OFF \
  -DBUILD_JAVA=OFF \
  -DENABLE_PROFILING=OFF \
  -DENABLE_NEON=OFF \
  -DENABLE_VFPV3=OFF \
  -DBUILD_ZLIB=OFF \
  -DBUILD_JASPER=OFF \
  -DBUILD_WEBP=OFF \
  -DBUILD_TBB=OFF \
  -DBUILD_IPP_IW=OFF \
  -DBUILD_ITT=OFF \
  -DBUILD_OPENEXR=OFF \
  -DENABLE_PYLINT=OFF \
  -DENABLE_FLAKE8=OFF \
  -DBUILD_opencv_python=OFF \
  -DWITH_PROTOBUF=OFF \
  -DOPENCV_EXTRA_MODULES_PATH=<openCV-Cloned-Path>/opencv_contrib/modules/ \
  -DBUILD_opencv_apps=OFF \
  -DBUILD_opencv_aruco=OFF \
  -DBUILD_opencv_bgsegm=OFF \
  -DBUILD_opencv_bioinspired=OFF \
  -DBUILD_opencv_calib3d=ON \
  -DBUILD_opencv_ccalib=OFF \
  -DBUILD_opencv_core=ON \
  -DBUILD_opencv_datasets=OFF \
  -DBUILD_opencv_dnn=OFF \
  -DBUILD_opencv_dnn_objdetect=OFF \
  -DBUILD_opencv_dnn_superres=OFF \
  -DBUILD_opencv_dpn=OFF \
  -DBUILD_opencv_face=OFF \
  -DBUILD_opencv_features2d=ON \
  -DBUILD_opencv_flann=ON \
  -DBUILD_opencv_fuzzy=OFF \
  -DBUILD_opencv_gapi=ON \
  -DBUILD_opencv_hfs=OFF \
  -DBUILD_opencv_highgui=ON \
  -DBUILD_opencv_img_hash=OFF \
  -DBUILD_opencv_imgcodecs=ON \
  -DBUILD_opencv_imgproc=ON \
  -DBUILD_opencv_intensity_transform=OFF \
  -DBUILD_opencv_java_bindings_generator=OFF \
  -DBUILD_opencv_js=OFF \
  -DBUILD_opencv_line_descriptor=OFF \
  -DBUILD_opencv_mcc=OFF \
  -DBUILD_opencv_ml=OFF \
  -DBUILD_opencv_objdetect=OFF \
  -DBUILD_opencv_optflow=ON \
  -DBUILD_opencv_phase_unwrapping=OFF \
  -DBUILD_opencv_photo=ON \
  -DBUILD_opencv_plot=OFF \
  -DBUILD_opencv_python3=OFF \
  -DBUILD_opencv_python_bindings_generator=OFF \
  -DBUILD_opencv_quality=OFF \
  -DBUILD_opencv_rapid=OFF \
  -DBUILD_opencv_reg=OFF \
  -DBUILD_opencv_rgbd=OFF \
  -DBUILD_opencv_saliency=OFF \
  -DBUILD_opencv_shape=OFF \
  -DBUILD_opencv_stereo=OFF \
  -DBUILD_opencv_stitching=OFF \
  -DBUILD_opencv_structured_light=OFF \
  -DBUILD_opencv_superres=OFF \
  -DBUILD_opencv_surface_matching=OFF \
  -DBUILD_opencv_text=OFF \
  -DBUILD_opencv_tracking=OFF \
  -DBUILD_opencv_ts=OFF \
  -DBUILD_opencv_video=ON \
  -DBUILD_opencv_videoio=ON \
  -DBUILD_opencv_videostab=OFF \
  -DBUILD_opencv_world=ON \
  -DBUILD_opencv_xfeatures2d=OFF \
  -DBUILD_opencv_ximgproc=ON \
  -DBUILD_opencv_xobjdetect=OFF \
  -DBUILD_opencv_xphoto=OFF \
  ..
```

After the execution use

```
cd build
make -j4
sudo make install
```

Notice that in the install command there is a line providing a path. Keep this path.

```
-- Installing: /usr/local/opencv-4.1.0/lib/pkgconfig/opencv4.pc
```

1.1. Link OpenCV to vp-cpp-template (Ubuntu)

Edit the ***BuildSystems/*****BuildConfigurations.cmake** file and do three modifications:

- Enable the code `set(IS_USING_OPENCV TRUE)`.
- Modify the code `set(ENV{PKG_CONFIG_PATH} "/usr/local/opencv-4.1.0/lib/pkgconfig/:$ENV{PKG_CONFIG_PATH}")` with the path that you have kept in the previous step **above**.
- Modify the code `pkg_check_modules(OPENCV4_PKG REQUIRED IMPORTED_TARGET opencv4>=4.1.0)` using the appropriate name and version (i.e. `opencv4` and `4.1.0`). The name is the name of the **opencv4.pc** file, and the version can be seen by viewing the contents of this **opencv4.pc** file.

Now all the source codes can have access to the OpenCV as far as they include the <*Common.hpp*> header (or the standard <*opencv2/opencv.hpp*> header). If OpenCV is linked in this way to the **vp-cpp-template** project then the definition **VP_CPP_TEMPLATE__USING_OPENCV** will have been set. An example on how to use this definition is provided in the file ***C++/src/ModuleTests/*****TestOpenCV.cpp**:

```
int Test_OpenCV(int argc, char** argv) {
  _ERRI(0!=confCLI.readCommandLine(argc, argv),"Failed to read command line!")
  #ifdef VP_CPP_TEMPLATE__USING_OPENCV
    info_(1,"* Using OpenCV ["<<CV_MAJOR_VERSION<<"."<<CV_MINOR_VERSION<<"]")
    try {
      cv::Mat img = cv::Mat::zeros(cv::Size(32,32),CV_8UC1);
      for (int y=0; y < 32; y++) {
        auto p = img.ptr<uint8_t>(y);
        for (int x = 0; x < 32; x++) { p[x] = x*8; }
      }
      cv::imwrite("testOpenCV_result.bmp",img);
    } catch (std::exception& e) { _ERRI(1,"Exception:\n[%s]",e.what()) }
  #else
    info_(1,"* OpenCV is disabled for this build.")
  #endif
  return 0;
}
```

2. Clone and Build OpenCV (Windows)

For the windows build a similar procedure can be used. To follow the same style as Ubuntu for the cmake files, you should first install the pgk-config[8] in Windows on mingw.

Supposing we want to Build the OpenCV 4.1.0 then follow the procedure:

```
git clone https://github.com/opencv/opencv
git clone https://github.com/opencv/opencv_contrib
cd opencv
git checkout 4.1.0
```

Then modify and use as an template-example the following **build.bat** file.

```
cls
del /F /S /Q build
mkdir build
cd build

cmake -G "MinGW Makefiles" ^
 -DBUILD_SHARED_LIBS=ON ^
 -DCMAKE_MAKE_PROGRAM=C:/TDM-GCC-64/bin/mingw32-make.exe ^
 -DCMAKE_INSTALL_PREFIX=<openCV-Cloned-Path>/opencv/install ^
 -DCMAKE_BUILD_TYPE=Release ^
 -DCMAKE_CONFIGURATION_TYPES=Release ^
 -DOPENCV_GENERATE_PKGCONFIG=ON ^
 -DCMAKE_CXX_STANDARD=11 ^
 -DCMAKE_CXX_STANDARD_REQUIRED=TRUE ^
 -DGLIBCXX_USE_CXX11_ABI=0 ^
 -DBUILD_DOCS=OFF ^
 -DBUILD_PERF_TESTS=OFF ^
 -DBUILD_TESTS=OFF ^
 -DBUILD_FAT_JAVA_LIB=OFF ^
 -DBUILD_CUDA_STUBS=OFF ^
 -DBUILD_JAVA=OFF ^
 -DENABLE_PROFILING=OFF ^
 -DENABLE_NEON=OFF ^
 -DENABLE_VFPV3=OFF ^
 -DBUILD_ZLIB=OFF ^
 -DBUILD_JASPER=OFF ^
 -DBUILD_WEBP=OFF ^
 -DBUILD_TBB=OFF ^
 -DBUILD_IPP_IW=OFF ^
 -DBUILD_ITT=OFF ^
 -DBUILD_OPENEXR=OFF ^
 -DENABLE_PYLINT=OFF ^
 -DENABLE_FLAKE8=OFF ^
 -DBUILD_opencv_python=OFF ^
 -DWITH_PROTOBUF=OFF ^
 -DWITH_VTK=OFF ^
 -DWITH_OPENCL_D3D11_NV=OFF ^
 -DOPENCV_EXTRA_MODULES_PATH=<openCV-Cloned-Path>/opencv_contrib/modules ^
 -DBUILD_opencv_apps=OFF ^
 -DBUILD_opencv_aruco=ON ^
 -DBUILD_opencv_bgsegm=OFF ^
 -DBUILD_opencv_bioinspired=OFF ^
 -DBUILD_opencv_calib3d=ON ^
 -DBUILD_opencv_ccalib=OFF ^
 -DBUILD_opencv_core=ON ^
 -DBUILD_opencv_datasets=OFF ^
 -DBUILD_opencv_dnn=OFF ^
 -DBUILD_opencv_dnn_objdetect=OFF ^
 -DBUILD_opencv_dnn_superres=OFF ^
 -DBUILD_opencv_dpn=OFF ^
 -DBUILD_opencv_face=OFF ^
 -DBUILD_opencv_features2d=ON ^
 -DBUILD_opencv_flann=ON ^
 -DBUILD_opencv_fuzzy=OFF ^
 -DBUILD_opencv_gapi=ON ^
 -DBUILD_opencv_hfs=OFF ^
 -DBUILD_opencv_highgui=ON ^
 -DBUILD_opencv_img_hash=OFF ^
 -DBUILD_opencv_imgcodecs=ON ^
 -DBUILD_opencv_imgproc=ON ^
 -DBUILD_opencv_intensity_transform=OFF ^
 -DBUILD_opencv_java_bindings_generator=OFF ^
 -DBUILD_opencv_js=OFF ^
 -DBUILD_opencv_line_descriptor=ON ^
 -DBUILD_opencv_mcc=OFF ^
 -DBUILD_opencv_ml=OFF ^
 -DBUILD_opencv_objdetect=OFF ^
 -DBUILD_opencv_optflow=ON ^
 -DBUILD_opencv_phase_unwrapping=OFF ^
```

8 https://stackoverflow.com/questions/1710922/how-to-install-pkg-config-in-windows

```
-DBUILD_opencv_photo=ON ^
-DBUILD_opencv_plot=OFF ^
-DBUILD_opencv_python3=OFF ^
-DBUILD_opencv_python_bindings_generator=OFF ^
-DBUILD_opencv_quality=OFF ^
-DBUILD_opencv_rapid=OFF ^
-DBUILD_opencv_reg=OFF ^
-DBUILD_opencv_rgbd=OFF ^
-DBUILD_opencv_saliency=OFF ^
-DBUILD_opencv_shape=OFF ^
-DBUILD_opencv_stereo=OFF ^
-DBUILD_opencv_stitching=OFF ^
-DBUILD_opencv_structured_light=OFF ^
-DBUILD_opencv_superres=OFF ^
-DBUILD_opencv_surface_matching=OFF ^
-DBUILD_opencv_text=OFF ^
-DBUILD_opencv_tracking=OFF ^
-DBUILD_opencv_ts=OFF ^
-DBUILD_opencv_video=ON ^
-DBUILD_opencv_videoio=ON ^
-DBUILD_opencv_videostab=OFF ^
-DBUILD_opencv_world=ON ^
-DBUILD_opencv_xfeatures2d=OFF ^
-DBUILD_opencv_ximgproc=ON ^
-DBUILD_opencv_xobjdetect=OFF ^
-DBUILD_opencv_xphoto=OFF ..
```

After the execution use

```
cd build
mingw32-make -j8
mingw32-make install
```

In the ***opencv\build\unix-install\*** folder the file **opencv4.pc** will be available.

2.1. Link OpenCV to vp-cpp-template (Windows)

Edit the ***BuildSystems/*****BuildConfigurations.cmake** file and do four modifications:

- Enable the code `set(IS_USING_OPENCV TRUE)`.
- Modify the code `set(ENV{PKG_CONFIG_PATH} "<your .pc file path>")` with the path that you have kept in the previous step, where the **opencv4.pc** file exist. The path should be in windows style thus use "\\" to avoid using the escape sequence character "\".
- Modify the code `pkg_check_modules(OPENCV4_PKG REQUIRED IMPORTED_TARGET opencv4>=4.1.0)` using the appropriate name and version (i.e. `opencv4` and `4.1.0`). The name is the name of the **opencv4.pc** file, and the version can be seen by viewing the contents of this **opencv4.pc** file.
- You should manually copy opencv's generated .dll files (i.e. **opencv_ffmpeg410_64.dll** and **libopencv_world410.dll**) to the ***workdir/*** folder.

Now all the source codes can have access to the OpenCV as far as they include the <*Common.hpp*> header (or the standard <*opencv2/opencv.hpp*> header). If OpenCV is linked in this way to the **vp-cpp-template** project then the definition VP_CPP_TEMPLATE__USING_OPENCV will have been set. As an example of use check the ***C++\src\ModuleTests\*****TestOpenCV.cpp** file.