Guide to vp-cpp-template

This document is part of the repository:
https://github.com/terablade2001/vp-cpp-template

Revision: **1** | Version: **73**

Document Status: **Public**

* LibreOffice Template based on: https://github.com/terablade2001/LibreOffice-Templates/blob/master/writer/CSoftDevReport-v42I.ott.
Modification on the template have been applied, especially regarding the used fonts.

Revision History & Document Info

Rev	Date	Author(s)	ACT	Description
1	01 Oct 2022	VP	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)

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

- ***A**: Added - A new section has been added.*
- ***U**: Updated - A section has been updated (**Minor** modifications (M) and extended (E))*
  - ***E**: Extended - A section has only been extended with new information.*
  - ***M**: Modified - Important content from a section has been deprecated, and possible replaced.*
    - ***ME**: Modified & Extended - **Major** modifications (M) and extended (E).*
- ***D**: Deleted - A section has been removed from the document.*
  - ***O**: Outdated - A section is invalidated/outdated but still remains in the document for historical reasons or because it may contain some useful information (Use "~~Strike Trough~~" in section's Title).*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: 73 -Modified: Oct 1, 2022, 23:00

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*

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

- **vkpBuildVersioner** [C++, Python]: System for automated versioning in C++ and 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.

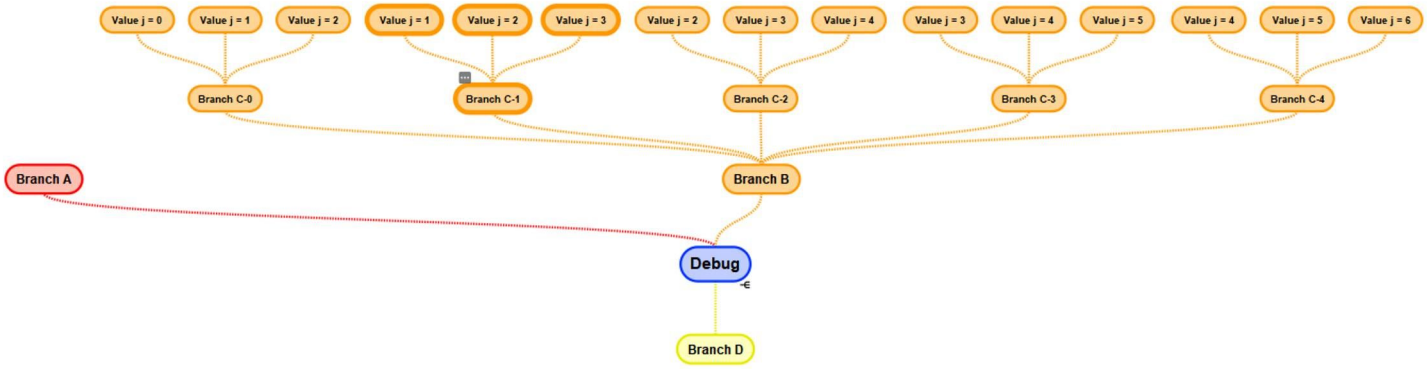


Figure 1: Example of debug information to interactive MindMap via vkp::opmlCPP.

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"
```

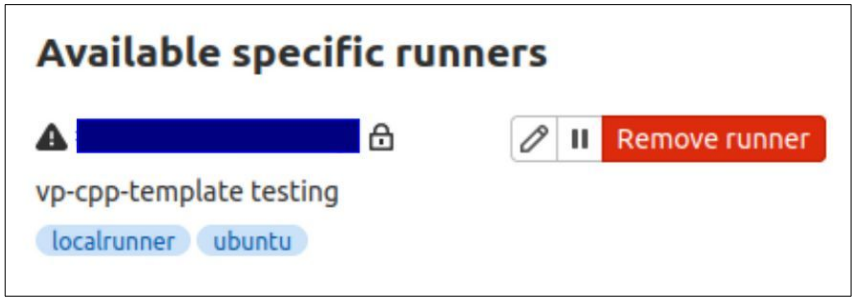
Restart the runner using:

```
sudo gitlab-runner restart
```

Now from the project's settings in GitLab website find again the field of "Specific run-

ners" 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.



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.

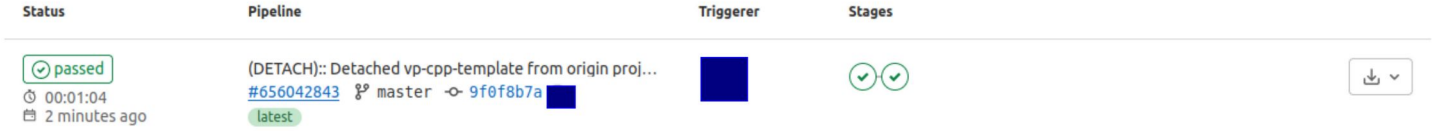
The above steps of the previous Chapter 1 are expected to have happened once in the user's system.

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...

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



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'
```