default

February 7, 2024

```
<hr />
<h1 class="text-center">
    <span class="text-primary">Master Class 2023 / Checkpoint 2</span>
</h1>
<hr />
```

```
- Summary -
```

What does the project contain?

- Initial Setup
- Introduction
- Project Goal
- Part 1: Write the Robot Info Node
- Part 2: Write the GUI node
- Optional: Add a 'reset distance' service
- Optional: Extend the CVUI library

```
- End of Summary -
```

```
<h2 class="text-center">
    <span class="text-primary"></span>
     
    <span class="">Initial Setup</span>
</h2>
```

To download the code, run the following commands:

Execute in Shell

[ ]:
```
cd ~/catkin_ws/src
```

Download the repository:

Execute in Shell

[ ]:
```
git clone https://bitbucket.org/theconstructcore/advanced_cpp_auxiliary_pkgs.git
```

Finally, make sure to compile after you've downloaded the code:

Execute in Shell

[ ]:
```
cd ~/catkin_ws && catkin_make
```

[ ]:
```
source devel/setup.bash
```

0.1 Start the simulation

Execute in Shell

[ ]:
```
roslaunch mir_gazebo mir_maze_world.launch
```

Important: you must unpause the simulation physics by clicking on the ”play” button in the Gazebo GUI or execute this service call:

[ ]:
```
rosservice call /gazebo/unpause_physics
```

You should see something like the picture below on your screen:

```
<h2 class="text-center">
    <span class="text-primary"></span>
     
    <span class="">Introduction</span>
</h2>
```

```
- IMPORTANT NOTES -
```

Please follow the instructions carefully. Note that you will often encounter very specific instructions about naming, such as the name of the package, names of files, topic names to use, etc. Please be sure to use exactly the names given to you in the instructions; otherwise, it will be challenging or even imposible for the team to correct the assignments. Failing to do so may result in points being deducted or your tasks being graded as incorrect.

Using Git is mandatory for this project. If you do not use Git, your assignment will not be reviewed, and you will need to redo the project using Git. Make a habit of committing your work daily or at least each time you complete one specific task. Also exercise writing clean meaningful commit messages that briefly describe the changes made in the commit. If you do not follow these instructions, it won't be possible for the team to correct some of the tasks.

Stop. Take a deep breath.

Grab a coffee.

1

Read that again.

Git Setup

Each time you want to use git in this project enviroment, you need to set up a few things before you start:

- Define the user name
- Define your email

This is needed so that git has information about the author of the changes. You do this with the following commands:

and

- IMPORTANT NOTES -

1.0.1 How to proceed with this project

- You can consult the **C++ Advanced** course to review concepts and reutilize code.
- You can consult the internet to search for additional information if needed.
- You are expected to gain a basic understanding of how to integrate code that is new to you, such as the CVUI library, into your project. **Working with new libraries is a part of a software developer's daily life and is a crucial skill to have.** Plan some time for experimenting, trying things out, and work with the code examples provided to you.
- Project tasks build upon one another so you should complete the tasks in the presented order.

```
<h2 class="text-center">
    <span class="text-primary"></span>
     
    <span class="">Story and Project Goal</span>
</h2>
```

You work in a robot lab that has recently acquired new mobile base robots. These robots are used for testing various robotic algorithms. Sometimes, these algorithms fail, and the robots need to be moved using a teleoperation interface. You might already be familiar with the **teleop_twist_keyboard** tool, which can be cumbersome for frequent teleoperation tasks. Your goal in this project is to develop a user-friendly graphical interface that even those with little or no previous experience in robotics can use to interact with the robots remotely. You will create this interface using C++, ROS, and the CVUI library. As with any typical ROS system, the project involves several ROS nodes communicating with each other and the Graphical Interface. As such this GUI is fed with information from a variety of ROS nodes that provide the data that is shown to the user. Similarly the user interface publishes information or calls ROS services through the ROS network. While some of these additional ROS nodes have already been provided, you will also be required to write one additional ROS node yourself, utilizing the knowledge and skills you have acquired throughout this course. Writing this node will enable you to gain further experience in writing C++ with ROS and help you to solidify your understanding of important C++ concepts.

The example image below showcases how the final product could appear. However, it is important to note that the visual design of your own user interface may differ. Although the creation of a graphical interface is crucial for this project, its appearance will not be graded. Instead, the main emphasis is on the underlying implementation of all features using C++ and ensuring their correct functioning via the graphical interface.

1.0.2 Design considerations

Let's have a closer look at the most important ROS nodes that the system requires, starting by the ROS node for the GUI:

- The GUI application functions as a ROS node, using subscribers, publishers and services to communicate with other nodes, and then displays data on the user's screen with the help of a graphical user interface library. For this project you will use OpenCV and CVUI, a very simple UI library built on top of OpenCV's drawing primitives. You don't need to have any previous experince using OpenCV or CVUI, as we provide plenty usage examples to get you familiarized with the library and help you understand how to put it to work.

As mentioned the GUI node subscribes to messages from other nodes but can also publish messages and call services. Here is a description of the other ROS nodes involved:

- **Robot Info Node**: An important portion of the functionality of this application resides in the **robot info** node. This nodes provides relevant information about the robot, such as the robot name and the current status of its systems. You will have to write this node as part of this project.
- **Distance Tracker Service node**: this node subscribes to a `nav_msgs/Odometry` topic, calculates the distance traveled by the robot using the odometry data, and provides a service server of type `std_srvs/Trigger` to send a message with the distance traveled on request. The service name is `/get_distance`.

```
<h1 class="text-center">
    <span class="text-primary">Part 1</span>
     
    <span class="">Write the Robot Info Node</span>
</h1>
```

During the course, you created a node called **RobotManager** that keeps track of all a robot information details, including the robot status, monitoring CPU and RAM usage and other details. Now, for this project part, you will create a comparable node that includes similar features you previously worked on, along with some additional capabilities that you will need to develop on your own.

- Your program must have its functionality implemented into into smaller, self-contained classes. Each class can have its own methods and properties, and be responsible for a specific task or functionality.

Before you start create a new ROS package called **robot_info** inside ~/**catkin_ws/src**:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src
```

[ ]:
```
catkin_create_pkg robot_info roscpp robotinfo_msgs
```

Create a new local Git repository in the package's root directory:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src/robot_info
```

[ ]:
```
git init
```

Remember that using Git is mandatory for this project.

1.1 Specifications

Step 1: Inheritance

- You must create one **robot_info_class.cpp** file for the **RobotInfo** class described below and one **robot_info_main.cpp** file for the main function.
- Then you have to write a class named **RobotInfo** that keep general data related to a robot. In particular this class must store the following information (as strings):
  - **robot_description**: The robot description, for instance the brand or make.
  - **serial_number**: The robot's serial number
  - **ip_address**: The robot's IP Address
  - **firmware_version**: The robot's Firmware version
- This class must also implement a **ROS publisher** that will use a custom message type to send the above information fields via the `robot_info` topic through the ROS network.
- You can create your own message type or use the supplied message located inside the **robot-info_msgs** package which is part of the **/advanced_cpp_auxiliary_pkgs** repository.
- **RobotInfo** also has a virtual method called `publish_data()` which publishes the data kept in the member variables robot_description, serial_number, ip_address and firmware_version.
- Initialize an **RobotInfo** object inside **robot_info_main.cpp** and write the neccesary code for this file to serve as starting point for the execution of this ROS node.
- You must build an executable called **robot_info_node** from the two source files above.
- Create a new source file called **agv_robot_info_class.cpp**. Inside it, using inheritance, create a new class called **AGVRobotInfo** based on the existing class **RobotInfo**. The derived class has only one member variables called `maximum_payload` which is the mayimum weight the robot can transport and only applies to autonomous ground vehicles (AGV's). Write the derived class **AGVRobotInfo** in such as way that it overrides the `publish_data()` method from the base class. Upon calling this method, the publisher must be executed to transmit the data over the ROS network. The transmitted data should include the contents of the parent class as well as the data stored in the `maximum_payload` variable.

The base class **RobotInfo** must hold a ROS Publisher to transmit its data to other nodes.

- Using the source file for the derived class build an executable called **agv_robot_info_node**.
- Running the **agv_robot_info_node** executable in conjunction with the `rostopic echo` command should produce the following result:

```
[ ]: data_field_01: "robot_description: Mir100"
     data_field_02: "serial_number: 567A359"
     data_field_03: "ip_address: 169.254.5.180"
     data_field_04: "firmware_version: 3.5.8"
     data_field_05: "maximum_payload: 100 Kg"
     data_field_06:
     data_field_07:
     data_field_08:
     data_field_09:
```

```
data_field_10:
```

Commit your work once you get the expected result. Let me say that again, because it's important: commit your work to your Git repository before you continue beyond this point!

NOW, AFTER COMMITING YOUR WORK, MARK THIS POINT IN YOUR PROJECT CREATING A GIT TAG.

Git tags are like checkpoint in the development history of your project. If you don't add these tags the team may not be able to correct your work, and it probably won't be possible for the team to review and grade this part of your work.

Proceed to create a Git tag called `part1-step1` by using these commands here:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src/robot_info
```

[ ]:
```
git tag part1-step1
```

`- Grading Guide -`

- Checkout the Git tag for part 1 step 1:

  `git checkout part1-step1`

  If this tag does not exist it is not possible to grade this part of the project.

- Using the IDE, open the file **robot_info_class.cpp** located in **~catkin_ws/src/robot_info/src**.
- This file must contain a class named **RobotInfo** with member variables robot_description, serial_number, ip_address and firmware_version. **(1 point)**
- The **RobotInfo** class correctly implements a ROS publisher object and a method called `publish_data()`. **(1 point)**
- Running the **robot_info_node** executable and `rostopic echo` produces the expected result as shown below. **(1 point)**

Execute in Terminal #1

[ ]:
```
cd ~/catkin_ws && catkin_make && source devel/setup.bash
```

[ ]:
```
roscore
```

Execute in Terminal #2

[ ]:
```
rosrun robot_info robot_info_node
```

Execute in Terminal #3

[ ]:
```
rostopic echo robot_info
```

Expected result:

```
data_field_01: "robot_description: Mir100"
data_field_02: "serial_number: 567A359"
data_field_03: "ip_address: 169.254.5.180"
data_field_04: "firmware_version: 3.5.8"
data_field_05:
data_field_06:
data_field_07:
data_field_08:
data_field_09:
data_field_10:
```

- Using the IDE, open the file **agv_robot_info_class.cpp** located in **~catkin_ws/src/robot_info/src**.
- This file must contain a class named **AGVRobotInfo** which derives from **RobotInfo** and has a member variable named `maximum_payload`. **(1 point)**
- The **AGVRobotInfo** class correctly overrides the `publish_data()` method. **(1 point)**
- Running the **agv_robot_info_node** executable and `rostopic echo` produces the expected result as shown below. **(1 point)**

Execute in Terminal #1

```
cd ~/catkin_ws && catkin_make && source devel/setup.bash
```

```
roscore
```

Execute in Terminal #2

```
rosrun robot_info agv_robot_info_node
```

Execute in Terminal #3

```
rostopic echo robot_info
```

Expected result:

```
data_field_01: "robot_description: Mir100"
data_field_02: "serial_number: 567A359"
data_field_03: "ip_address: 169.254.5.180"
data_field_04: "firmware_version: 3.5.8"
data_field_05: "maximum_payload: 100 Kg"
data_field_06:
data_field_07:
data_field_08:
data_field_09:
data_field_10:
```

- End Grading Guide -

Step 2: Composition

- Additionally, the **AGVRobotInfo** class must be composed of a **HydraulicSystemMonitor** object. This object is initialized inside the **AGVRobotInfo** class as a member variable, and its member variables can be accessed from within the **AGVRobotInfo** class's methods.
- The **HydraulicSystemMonitor** class must be defined inside the file **hydraulic_system_monitor.cpp** which must be located inside ~**catkin_ws/src/robot_info/src**.

The **HydraulicSystemMonitor** class must have three member variables of type `std::string` to hold the following data required for condition monitoring (variable names to be used appear bold): - **hydraulic_oil_temperature**: the temperature of the hydraulic oil helps to identify malfunction early and extended the operational time for the oil. - **hydraulic_oil_tank_fill_level**: can help identify any issues before the pump starts to suck in air. - **hydraulic_oil_pressure**: Monitoring the pressure is critical to ensure safe operation.

- You must write one or more methods on that object the can be called to retrieve the attribute values individually or as a group.

You must also modify the `publish_data()` method implementation in the **AGVRobotInfo** class in such a way that it also publishes the **hydraulic oil temperature**, **hydraulic oil tank fill level** and **hydraulic oil pressure** alongside with the data stored in the **RobotInfo** class and **AGVRobotInfo**.

All in all, the message published to the **robot_info** topic must contain following pieces of data: - robot_description - serial_number - ip_address - firmware_version - maximum_payload - hydraulic_oil_temperature - hydraulic_oil_tank_fill_level - hydraulic_oil_pressure

For simplicity reasons all these data fields are strings.

Running the **agv_robot_info_node** executable in conjunction with the `rostopic echo robot_info` command should produce the following result:

[ ]:
```
data_field_01: "robot_description: Mir100"
data_field_02: "serial_number: 567A359"
data_field_03: "ip_address: 169.254.5.180"
data_field_04: "firmware_version: 3.5.8"
data_field_05: "maximum_payload: 100 Kg"
data_field_06: "hydraulic_oil_temperature: 45C"
data_field_07: "hydraulic_oil_tank_fill_level: 100%"
data_field_08: "hydraulic_oil_pressure: 250 bar"
data_field_09:
data_field_10:
```

Commit your work once you get the expected result!

AFTER COMMITING YOUR WORK, MARK THIS POINT IN YOUR PROJECT CREATING A NEW GIT TAG.

Proceed to create a Git tag called `part1-step2` by using these commands here:

Execute in Shell #1

```
[ ]: cd ~/catkin_ws/src/robot_info
```

```
[ ]: git tag part1-step2
```

- Grading Guide -

- Checkout the Git tag for part 1 step 2:

  `git checkout part1-step2`

  If this tag does not exist it is not possible to grade this part of the project.

- Using the IDE, open the file **hydraulic_system_monitor.cpp** located in **~catkin_ws/src/robot_info/src**.
- This file must contain a class named **HydraulicSystemMonitor** and have the following three member variables of type `std::string`: hydraulic_oil_temperature, hydraulic_oil_tank_fill_level and hydraulic_oil_pressure. **(1 point)**
- The **AGVRobotInfo** class has one member variable of type **HydraulicSystemMonitor** and one or more methods that calls the method(s) of the **HydraulicSystemMonitor** object and then assigns the values to data fields in the outgoing ROS message. **(1 point)**
- Running the **agv_robot_info_node** executable and `rostopic echo` produces the expected result as shown below. **(1 point)**

Execute in Terminal #1

```
[ ]: cd ~/catkin_ws && catkin_make && source devel/setup.bash
```

```
[ ]: roscore
```

Execute in Terminal #2

```
[ ]: rosrun robot_info agv_robot_info_node
```

Execute in Terminal #3

```
[ ]: rostopic echo robot_info
```

Expected result:

```
[ ]: data_field_01: "robot_description: Mir100"
data_field_02: "serial_number: 567A359"
data_field_03: "ip_address: 169.254.5.180"
data_field_04: "firmware_version: 3.5.8"
data_field_05: "maximum_payload: 100 Kg"
data_field_06: "hydraulic_oil_temperature: 45C"
data_field_07: "hydraulic_oil_tank_fill_level: 100%"
data_field_08: "hydraulic_oil_pressure: 250 bar"
data_field_09:
```

```
data_field_10:
```

```
- End Grading Guide -
```

```
<h1 class="text-center">
    <span class="text-primary">Part 2</span>
     
    <span class="">Write the GUI node</span>
</h1>
```

You will have to write the GUI node in a separate ROS package called **robot_gui** inside ~/**catkin_ws/src**:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src
```

[ ]:
```
catkin_create_pkg robot_gui roscpp nav_msgs geometry_msgs std_srvs robotinfo_msgs
```

Create a new local Git repository in the package's root directory:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src/robot_gui
```

[ ]:
```
git init
```

Remember that using Git is mandatory for this project.

Before you start to build your GUI node, please familiarize yourself with CVUI by going through the provided examples (located inside the ~**catkin_ws/src/advanced_cpp_auxiliary_pkgs** directory), the examples in the official CVUI github repository, the CVUI documentation and other ressources that you can find online.

When you have become acquainted with the CVUI library, proceed to construct the GUI node in accordance with the requirements below. The image shown here is just for reference. Remember that your GUI design can be different and that the grading will be based on functionality rather than visual aesthetics.

1.2 Specifications

- **General Info Area**: include an area to display the messages published into the `robot_info` topic. If the published message changes, the GUI must update accordingly. Commit your work after having created the GUI with the **general info area**.
- **Teleoperation Buttons**: Include buttons for increasing and decreasing the speed in the x-axis direction and the rotation on the z-axis. These buttons must be fully functional, so please ensure that pressing the buttons and modifying the speed updates the data in the `cmd_vel` topic and the simulated robot behaves accordingly. Commit your work after having created the GUI with the **teleoperation buttons**.

- **Current velocities**: Display the current speed send to the robot via the `cmd_vel`topic as "Linear Velocity" and "Angular Velocity". These buttons must be fully functional, so please ensure that pressing the buttons modifies the speed shown in this part of the GUI. Commit your work after having created the GUI with the **Current velocities**.
- **Robot position (Odometry based)**: Subscribe to `/odom` and display the current x,y,z position to the GUI. The values displayed must be the actual values being published to the /odom topic. Commit your work after having created the GUI that shows the **robot position**.
- **Distance travelled service**: Create a button that calls the `/get_distance` service and displays the response message to the screen. Ensure that pressing this button results in a functional service call via te ROS network and the response from the **distance_tracker_service** node is displayed in the GUI. The button's full functionality is required. Commit your work after having completed this part.

Create commits of your work if you need to do iterative adjustments to any of the GUI areas at any time. Please note that any uncommitted work will not be considered for grading!

AFTER YOUR FINAL COMMIT FOR THIS PART, MARK THIS POINT IN YOUR PROJECT CREATING A NEW GIT TAG.

Proceed to create a Git tag called part2 by using these commands here:

Execute in Shell #1

[ ]:
```
cd ~/catkin_ws/src/robot_gui
```

[ ]:
```
git tag part2
```

- Grading Guide -

- Checkout the Git tag for part 2:

  `git checkout part2`

- Run the **robot_gui_node** executable as shown below:

Execute in Terminal #1

[ ]:
```
cd ~/catkin_ws && catkin_make && source devel/setup.bash
```

[ ]:
```
roscore
```

Start the **robot_info** node and the **distance_tracker_service** node:

Execute in Terminal #2

[ ]:
```
rosrun robot_info agv_robot_info_node
```

Execute in Terminal #3

[ ]:
```
rosrun distance_tracker_service distance_tracker_service
```

Execute in Terminal #4

```
[ ]: rosrun robot_gui robot_gui_node
```

- Running the **robot_gui_node** executable launches a GUI window that can be seen using the graphical tools. **(1 point)**
- The GUI window includes a fully functional **General Info Area**. For additional testing shut down the **robot_info** node and publish a message to the `robot_info` topic via the command line. The message contents should be displayed inside the GUI. **(1 point)**
- The GUI window includes a fully functional **Teleoperation Buttons**. Pressing the buttons should result in movement of the simulated robot. **(1 point)**
- The current linear velocity in the x-axis and angular velocity in the z-axis is displayed. Test it by pressing the **Teleoperation Buttons**. **(1 point)**
- The GUI window displays the current **robot position** based on odometry data. For additional testing shut down the simulation and publish a message to the `odom` topic via the command line. The published values should be displayed inside the GUI. **(1 point)**
- The GUI features a button that calls the `/get_distance` service. The response from the distance_tracker_service node is displayed in the GUI. **(1 point)**

```
- End Grading Guide -
```

```
<h1 class="text-center">
    <span class="text-primary">Optional:</span>
     
    <span class="">Add a 'reset distance' service</span>
</h1>
```

- Update the **distance_tracker_service** node by adding a second service that resets the distance traveled to zero.
- Modify the GUI to call the reset distance service. Update the GUI field that shows the distance accordingly.
- This optional part will not be graded.

```
<h1 class="text-center">
    <span class="text-primary">Optional:</span>
     
    <span class="">Extend the CVUI library</span>
</h1>
```

- Modify the CVUI library to add a functionality that you think is missing. You could add for instance a progress bar, a display for the robot's laser scan data, a battery level indicator, a simulated analog joystick control stick or any other graphical element that you think is usefull.
- Make use of that functionality in your **robot_gui** node.
- This part is optional and will not be graded.