Robot Programming Introduction

Getting Started

Now that you've learned the basics of C++, we can move on to programming on the robot!

A Brief Introduction to ROS

You will be using the **Robot Operating System** (ROS) to develop software for your robot. ROS is an open-source meta-operating system for robots which is used around the world for robotics software development and research.

We've developed software for you to use ROS without needing to explore the complexities of this system, but if you're interested in exploring ROS after the camp, their website provides many tutorials and has a vast community of developers. Feel free to check it out after the camp to learn more about how ROS works!

A Boilerplate ROS Program

To get us started, here's a simple, brief ROS program that is a sort of "empty" program:

```
#include <ros/ros.h>
#include <unistd.h>
#include "texas_robotics_academy/texbot_wrapper.h"

int main(int argc, char **argv) {

    ros::init(argc, argv, "robot");

    TexBot bot;

    while(ros::ok()) {
        //Your code goes here!
```

```
        ros::spinOnce();

    }

    return 0;
}
```

Let's explore what's going on here.

```
#include <ros/ros.h>
#include <unistd.h>
#include "texas_robocamp/texbot_wrapper.h"
```

Hopefully these first lines should seem vaguely familiar - we're once again **including** other libraries to work with our code. In this case, the libraries we are including are the ROS libraries, libraries that provide access to the OS, and the robot libraries we've written to help you control your robot.

```
int main(int argc, char **argv) {
```

This function should look somewhat familiar as well - it's the `main` function! Unlike previous `main` functions you've seen so far, this one has two parameters, argc and argv. These can be used to pass in arguments from your terminal, but since we won't ever be using these arguments, you don't need to worry about them.

```
ros::init(argc, argv, "robot");
```

This function starts up our ROS process. ROS calls these **nodes**, and allows for arguments to be passed in via the command line with `argc` and `argv`. The last argument defines the name of the node - in this case, the name is "robot".

```
TexBot bot;
```

Here, we're creating an instance of the `TexBot` object, and calling it `bot`.

This object is a **class** just like the `BankAccount` class you developed earlier in the programming portion of this camp, and much like the `BankAccount` class, it contains a number of variables and functions, some of which are private and some of which are public. We'll explore these in more detail later on.

```
while(ros::ok()) {
```

This should seem familiar as well - it's another **while loop**. Whereas earlier you saw the condition be a logical expression involving numbers, this loop will execute for as long as the function `ros::ok()` returns `true`. In general, this will always be true until you close your program.

```
ros::spinOnce();
```

This function lets our node update its information. For our purposes, it lets the `TexBot` object update its sensor data so that we can be aware of what's going on in the simulation. We recommend writing all of your code before this line.

It is vital that you leave this line here! Without this function, our node would have no idea what is going on in the simulator.

```
return 0;
```

Here we see our usual return statement from main, which simply returns zero. This line will only be executed when `ros::ok()` is `false`, so don't write any code outside of the `while` loop because it will not be executed until after the ROS node is finished running.

Hello World - On the Robot's LCD

Now that we're developing software on the actual robot, let's go ahead and write a new "Hello World" program. However, instead of just printing "Hello World" on the computer screen, we're going to print it on the robot's LCD screen!

You might notice that there isn't actually an LCD screen on our robot. In the world of simulation, we developers have to make design choices that may differ from what a robot would look like in real life. This is a great example of one of these design choices. In simulation, printing out information on an actual screen on the robot would be difficult to read, since we're viewing the robot from above in Gazebo instead of being able to physically lean down and view what is being printed on the LCD. Because of this, we've built an additional UI (User Interface) where the LCD information will appear.

We will continue to refer to the screen as an LCD and name our function `lcd` so that if there were actual development on the physical robot, we would only have to change the behavior of our robot object instead of the code printing to the LCD.

The lcd Function

```
void lcd#(const std::string &line)
```

This function is actually one of two `lcd#` functions. The robot has two lines of LCD output. Calling `lcd1` or `lcd2` specifies which line you would like for your text to appear on. There is no concept of a "carriage return," "\n," or `endl` that is useful here.

The parameter here `const std::string &line` might look a little confusing; this is because we are passing the argument by **reference**. We won't go too into detail about what passing by reference is - all you need to know is that strings are treated differently in C++.

Here is an example of how you could use these functions:

```
bot.lcd1("How are you today?");
bot.lcd2("I'm doing great thanks");
```

The lcd functions must be called on the TexBot object - you attach the function to the object with a period. For our purposes, our TexBot is always named "bot", so anytime you want to call the lcd functions you will always use `bot.lcd1()` or `bot.lcd2()`.

Compiling and Running a ROS Program

Although you're still writing code in C++, we will be using a different build system to turn our ROS code into code that can be executed by your computer.

You should have already cloned the robocamp_exercises folder into your new **catkin workspace**. Inside that folder we have provided you with the directory structure and cpp files you will write your code in.

Inside the robocamp_exercises directory, you will see six directories named "4_1", "4_2", "4_3", etc. These are **packages**, which each contain a src folder. Inside the src folder are the cpp files for each individual exercise. By default, we've filled each cpp file with the boilerplate ROS program we showed you at the beginning of this page.

Next, we'll want to build our program. To do this, do

catkin build

It is important to run this command after you've made any changes to your program and before you try to run it. You can think of it as a ROS equivalent of the g++ command we used in our C++ exercises.

The Gazebo simulator is very complex, so it might slow down your computer. You'll want to make sure it is not running when you run `catkin build` to make the build go faster.

Now run this command to automatically source your setup.bash:

echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc source ~/.bashrc

In order to actually use the robot, you'll need to run what are called launch files. These are files which group up smaller executable programs so that we don't have to run all of them individually. Using the **roslaunch** command will fire up the gazebo simulation. The syntax for this is:

roslaunch <package_name> <file_name>

For the first few exercises, we'll be using the boxed world that we've built for you.

roslaunch texas_robotics_academy box.launch

Launch files aren't the only way to run a ROS program however; like we said earlier, they're actually launching smaller executables that are called ROS **nodes**. To launch a single node, do:

rosrun <package_name> <file_name>

The syntax for both `roslaunch` and `rosrun` commands are nearly identical! Don't forget that `roslaunch` is for launch files and `rosrun` is for individual nodes

To run **your** code, you'll always do the **rosrun** command, which we also mentioned earlier. Whenever you are running multiple launch files or nodes, you always need one terminal open for each. For this specific example, split your terminal and in the new terminal do:

rosrun 4_1 ex_4_1_1

Exercise 4.1.1

- Write your first Hello World program on the robot!
- Make this program say "Hello World!" on the first line in the LCD, and "Texas Robotics Academy!" on the second line.

Sleep

```
int usleep(useconds_t useconds)
```

One function that you will find helpful this week is `usleep` . This is a function of the unistd library, which gives us access to some of the operating system.

Calling this function will cause your program to wait for `useconds` microseconds.

You'll want to use `usleep` whenever you want to keep a robot in a certain state for some time, or if you want to slow down your program to debug information.

Exercise 4.1.2

In the ex_4_1_2.cpp file we have provided you, copy in your code from Exercise 4.1.1 and modify your program to:

- Say: "Hello World!" on the first line in the LCD, and "Texas Texas Robotics Academy!" on the second line.
- Wait 2 seconds
- Say: "Hook 'em" on the first line in the LCD, and "Horns!" on the second line.

- Then repeat this in a loop, after waiting 2 seconds with "Hook 'em Horns!" displayed.

You can modify this exercise to use whatever cheer you like. The point is to learn how the code works.

Reference:

- https://github.com/Texas-Robotics-Academy/markdown.git