FTC Programming Style Rules

These are the programming style rules used by Swerve Robotics' FTC teams. I have tried to reformat the style list in such a way that it would apply easily to other teams. I can be reached for questions or comment at: eukota@gmail.com.
Cheers!

Darrell Ross
Programming Mentor
Swerve Robotics Club

Introduction

Programming Style is incredibly important and usually ignored when learning to program. Instead of ignoring it, we will lay out an explicit set of rules for all programmers to follow. It is difficult to teach the "why" of these rules without the students first creating a large quagmire of code. To avoid the quagmire and ensuing refactor which results in a set of Style Rules, we will lay them out ahead of time and explain as best we can why they are needed.

In the worst case scenario, should we fail, the programmers will ultimately end up writing their own style rules anyway but only after some unknown length of time during which the code will be nearly indecipherable and only understood by some.

Code Repository

Similarly to how students work together on the physical pieces of the robot, the programming team must work together. The product of the programming team is the code base.

A copy of the code base is stored locally on each student's machine and kept in sync with a server. Teams can run their own server but there are numerous public options available:

- CodePlex.com - private repositories cost money
  - Git
  - Team Foundation Server (only use if you know this one)
  - Mercurial
- GitHub.com - private repositories cost money
  - Git
- BitBucket.com - allows free private repositories
  - Git
  - Mercurial

Any of these three free solutions will work fine for your team. As for the type of repository, I recommend using Mercurial. The primary reason for this is that Mercurial has a friendly GUI while Git is more useful via text terminals. If you feel that your team works smoothly with terminals, then Git might be right for you. If you fear terminals, use Mercurial. It will mostly allow you to avoid terminals.

Eventually, anyone who programs will need to learn to use terminals. In my experience though, students pick up the repository functionality faster using GUIs.

Table of Contents

Style Rules

The style rules range over a few general areas.

- **Programmer Level** style rules describe how programmers should go about thinking about problems with some best practices.
- **Code Level** style rules describe how programmers should write their code including formatting, naming constraints, and best coding practices.
- **Repository Level** style rules describe how programmers should organize their code into files, name their files, and structure their folders.

Programmer Level Style Rules

Programming take meticulous care. Programmers should not be hasty. Following rules helps facilitate the programmer mind set.

Code Level Style Rules

Scope Spacing

Scope refers to the scope of a variable but in this case, I am talking about the indentation level that each additional if/while/for statement adds. Each time that a scope-level statement is used, you MUST indent everything within that statement by a single additional tab character - in RobotC, this equates two spaces (in more sophisticated IDEs, you can save tab-characters and each user can configure their IDE to display tab characters as a certain number of spaces).

Code Example:

```
// first level
task main()
{
	// second level
	waitForStart();
	while(true)
	{
		// third level
		if(firstBoolean)
		{
			// fourth level
		}
		// third level again
	}
	// second level again
}
// first level again
```

Curly Braces

All curly braces should always appear on their own line.

Code Example:

See Scope Spacing code example.

Naming

For the purposes of clarity and conformity, different parts of the code will use different styles so that we can clearly delineate between them at a glance. In general, when you choose variable names, make them descriptive. This helps reduce the need for large comments. Note the difference between the two following code examples.

Bad Code Example:

```
// This function returns x divided by y
bool Q(int x, int y)
{
	return (x / y);
}
```

Good Code Example:

```
bool Quotient(int dividend, int divisor)
{
	return (dividend/divisor);
}
```

Variables

We will use what is often referred to as camel-case for most variables. This means you use no spaces or underscores between words in the name of a variable. Instead, you capitalize each word.

Code Examples:

```
bool myBooleanValue;
double superDuperLongVariableName;
```

Functions

Functions will be identical to variable naming except that the first word will be capitalized. With the exception of the main task, all tasks will follow this regime too.

Code Examples:

```
void InitializeRobot()
task DriveWheels()
```

Global Variables

All #defines and global constants must use all caps and use underscores between words. This helps programmers easily identify the difference in their code between a local variable and a global constant which is very important. Global constants are usually accessed at the top level and require changing from time to time.

Code Examples:

```
#define DEAD_ZONE (15)

#ifndef FTC_INIT_ROBOT_C
#define FTC_INIT_ROBOT_C
#endif

const float MINIMUM_CONTROLLER_INPUT = 15.0;
const float CONTROLLER_DEAD_ZONE = MINIMUM_CONTROLLER_INPUT;
```

Magic Numbers

Magic numbers are number values typed directly into the middle of your code and they should *never* be used. If you need a constant value, then you should predefine it as a constant or use a #define.

Magic Number Code Examples:

DO THIS	**NOT THIS**

`const int MAX = 50;`
 	
`int buf[MAX];`
`if(i < MAX)...`
	
	`int buf[50];`
	`if(i < 50)...`

Magic Numbers should instead be defined as constants at the top of the file either through a #define or a constant and then assigned within the function. Note the DEAD_ZONE used in the following section's Code Example for Function Encapsulation and Length.

Function Encapsulation and Length

All control of the robot should use functions. This includes but is not limited to motor control, reading sensors, and initializing the robot. All function definitions will go in files which are not the main file. This leaves the main task in a file by itself. #includes must be used to access all needed functions.

There is no hard-and-fast rule about function length but a safe bet is no longer than one page. If your function is longer than a page, you should seriously consider breaking it up into several functions.

Code Examples:

```
#include "motors.c"
const int DEAD_ZONE = 15;
task main()
{
      InitializeRobot();
      waitForStart();
      StartTask(TankDrive);
      while(true)
      {
            getJoystickSettings(joystick);
            if(abs(joystick.joy2_y1) > DEAD_ZONE)
            {
                  DriveLeftMotors(joystick.joy2_y1);
            }
      }
}
```

Comments

You *must* comment your code. When you leave comments, you are not only explaining it to your peers but to yourself. When you see this code in two years (or even in two months), you want to easily be able to remember what you intended to do.

Code Examples:

This code fragment does something weird but leaves no comment informing a future programmer why.

```
void DriveSpinnerMotor(int power)
{
      if(power>power-1) // should be a comment here explaining this
      {
            motor[spinnerMotor] = power;
      }
      else
      {
            motor[spinnerMotor] = -power;
      }
}
```

Comment Location

Comments about a code block should go above the code block. Comments about a single line should go at the end of the line unless this would make the line too long, in which case the comment should go above the line. If you do a good job of using descriptive variable names, you can reduce your need for detailed comments.

Code Example:

```
// ScaleControllerInput converts controller input values by
// first calculating the the slope-intercept equation based
// on constants.
int ScaleControllerInput(int controllerInput)
```

```
{
      float slope = (MAX_MOTOR-MIN_MOTOR)/(MAX_CONTROLLER-MIN_CONTROLLER);
      float intercept = (MIN_MOTOR-(slope*MIN_CONTROLLER));
      // note output may be reversed
      return (slope*controllerInput + sgn(controllerInput)*intercept);
}
```

#ifndef File Wrapping

When using #includes, it is quickly possible to ended up with recursive includes or errors due to including the same code more than once. To avoid this, all files other than the main compiled files *must* be wrapped in an #ifndef/#endif pair. The name of the the #ifndef *must* match the name of the file but using our #define Style Rule replacing any punctuation with underscores.

Code Example:

In the file motors.c:

```
#ifndef MOTORS_C
#define MOTORS_C
// all motors.c code here //
#endif
```

Repository Level Style Rules

There is no hard-and-fast rule about where to put your code or what to name your folders. Decisions about this should be discussed with your teammates so that you can all agree on where, for example, the code should go that allows you to handle the light sensors. Over time, you will find that some files must be split. This is normal. Just make sure you discuss larger library changes with your team so that nobody gets out of sync.

File Naming

All files and folders *must* be named using only lowercase characters. An effort should be made to use simple single-word naming. Users should be able to easily browse through folders at a glance without having to decipher the text.

Example:
motors.c would be placed in a folder named "motors".
The #include would then be adjusted to read `#include "motors/motors.c"`
Other motors related files would also go in the folder named “motors”.

Folder Structure

The primary downloaded Teleop code should be located at the top level of a directory structure. All includes should be maintained inside folders which describe what the includes handle.

Examples:
Include files full of global constants belong in a folder named “includes”.
Motor interaction functions belong in files in a folder named "motors".
Sensor interaction functions belong in files in a folder named "sensors".
Autonomous code belongs in a folder named "autonomous".

Style Rules Practice Lesson

To see the usefulness of these style rules and for practice, I recommend running the students through an exercise as follows.

Assignment

1. Write a simple tank drive program in RobotC based code.
2. You must start from scratch.
3. Your program must follow all of the style rules laid out in this guide.
4. When complete, let me know.

Followup

When they are done, have the students swap positions in pairs and perform a code review. Each reviewer should note:

1. at least two things the code does well
2. at least one improvement the code could have

Students should be encouraged to provide positive feedback and not to insult each other.

Assignment Enhancements

Here are a couple ideas you could use to enhance this lesson plan.

1. If you already have them working with repositories, you can have each student create their code inside a folder named “practice” in your main repository. So my code would be in \\repository\practice\darrell\. Instead of swapping computers for review, students would pull changes and review them. When we swap for review, the reviewers would commit the changes with good commit notes. Then when we swap back, the original authors will be able to update back to their old functions.

2. Reviewers could be encouraged to intentionally leave one or two bugs in the software which the authors would then need to find using version control diff software.