FIRST Team 1511 Rolling Thunder

Programming sub-team pre-season C++ class

C++ Overview

- Comment your code
- Code is case-sensitive
- Create functions to break up code into more readable pieces.
- Break up the program into separate sections (i.e. classes) for understanding and to allow multiple people to work on the project. For example, separate code that drives the robot from the code that moves a mechanism.

Variable Types

Variables are used to store temporary values

- Integer – a whole number that can be negative or positive

```
int counter;
```

- Floating point – a decimal/fractional number that can be negative or positive

```
float voltage;
```

- Boolean – true or false

```
bool flagTripped;
```

Enumerations

A data type that limits the valid values.

```
typedef enum {AUTO, TELE_OP, DISABLED} RobotMode;
```

The variable can only have these values

The type (e.g. instead of int or float)

```
RobotMode mode;
```

Syntax

Comments – help explain the code to others
(and yourself when you look at it later)

Commands end in a semi-colon
(do NOT forget them)

Braces define a section of code

```
/*
  Thunderbot
*/
#include "WPILib.h"

void OperaterControl() {
   float pos;
   while (IsOperatorControl() && IsEnabled()) {
      // get position of sensor 0
      pos = GetPosition(0);
      Wait(0.005);    // wait for motor update time
   }
}
```

#include

#include means: insert source code from file WPILib.h here

```
/*
  Thunderbot
*/
#include "WPILib.h"

void OperaterControl()
{
   float pos;
   while (IsOperatorControl() && IsEnabled()) {
      // get position of sensor 0
      pos = GetPosition(0);
      Wait(0.005);// wait for a motor update time
   }
}
```

Functions/Methods

```
float GetPosition(int sensor);   // prototype
```

Functions need a prototype. This one takes an integer as input and returns a float. Typically defined in header (i.e. .h) file.

```
void OperaterControl() {
    float pos;
    while (IsOperatorControl() && IsEnabled()) {
        pos = GetPosition(0);
        Wait(0.005);// wait for a motor update time
    }
}
```

This function does not return anything and has no input arguments.

The value zero is passed into the function

```
// Function GetPosition
float GetPosition(int sensor) {
    AnalogInput pot(sensor);
    return(pot.GetVoltage();
}
```

Within the function, the input value is represented by variable "sensor"

if

Statements within the **if** run when the conditional expression evaluates to *true*

```
if (flagTripped) {
   motor.Set(0);
}
```

Motor speed is stopped when the *flagTripped* is *true*

if-else

Statements within the **if** run when the conditional expression evaluates to *true*.

Statements within the **else** run when the conditional expression evaluates to *false*.

```
if (flagTripped) {
    motor.Set(0);
}
else {
    motor.Set(1);
}
```

Motor speed is stopped when the *flagTripped* is *true*
Motor speed is 1 when the *flagTripped* is *false*

while

Statements within the **while** run until the conditional expression evaluates to *false*

```
while (!flagTripped) {
     motor.Set(1);
     flagTripped = IsFlagTripped();
}
```

The motor will run at speed 1 until the variable *flagTripped* is *true*.

for

Statements within the **for** statement run, in order, until the conditional expression evaluates to *false*

```
for (i = 0; i < 5; i++) {
        statement1;
        statement2;
        statement3;
}
```

- The variable 'i' is initialized to zero before the loop is run the first time
- *Statement1* thru *3* are run while 'i' is less than 5 (the evaluation occurs prior to the statements being executed; including the first time)
- Each time after all the statements are run, 'i' is incremented by 1, then it is compared to 5

if/while/for Logical Operations

==	see if two values are **equal**
!=	see if two value are **not equal** to each other
<	see if value on the left is **less than** value on right
>	see if left value is **greater than** right value
<=	see if left is **less than or equal to** right value
>=	see if left is **greater than or equal to** right value

```
if (speed == 0) {
        printf("motor is stopped\n");
}
```

The text is printed when *speed* is zero

if/while/for Logical Operations

&&	compares two expressions and evaluates *true* when the left **and** right side are both *true*
||	compares two expressions and evaluates *true* when either the left **or** the right side are *true*

```
if ((pos > -0.1) && (pos < 0.1)) {
            motor.Set(0);
}
```

The motor will stop when the position is between -0.1 and 0.1

switch

The switch statement similar to **if-else**. The value is evaluated and then compared to each *case* to determine which statements to run. When the value does not match any of the cases, then the **default** statements are run.

```
switch (step) {
    default:
        statement1;
        break;
    case 0:
        statement2;
        break;
    case 1:
        statement3;
        break;
}
```

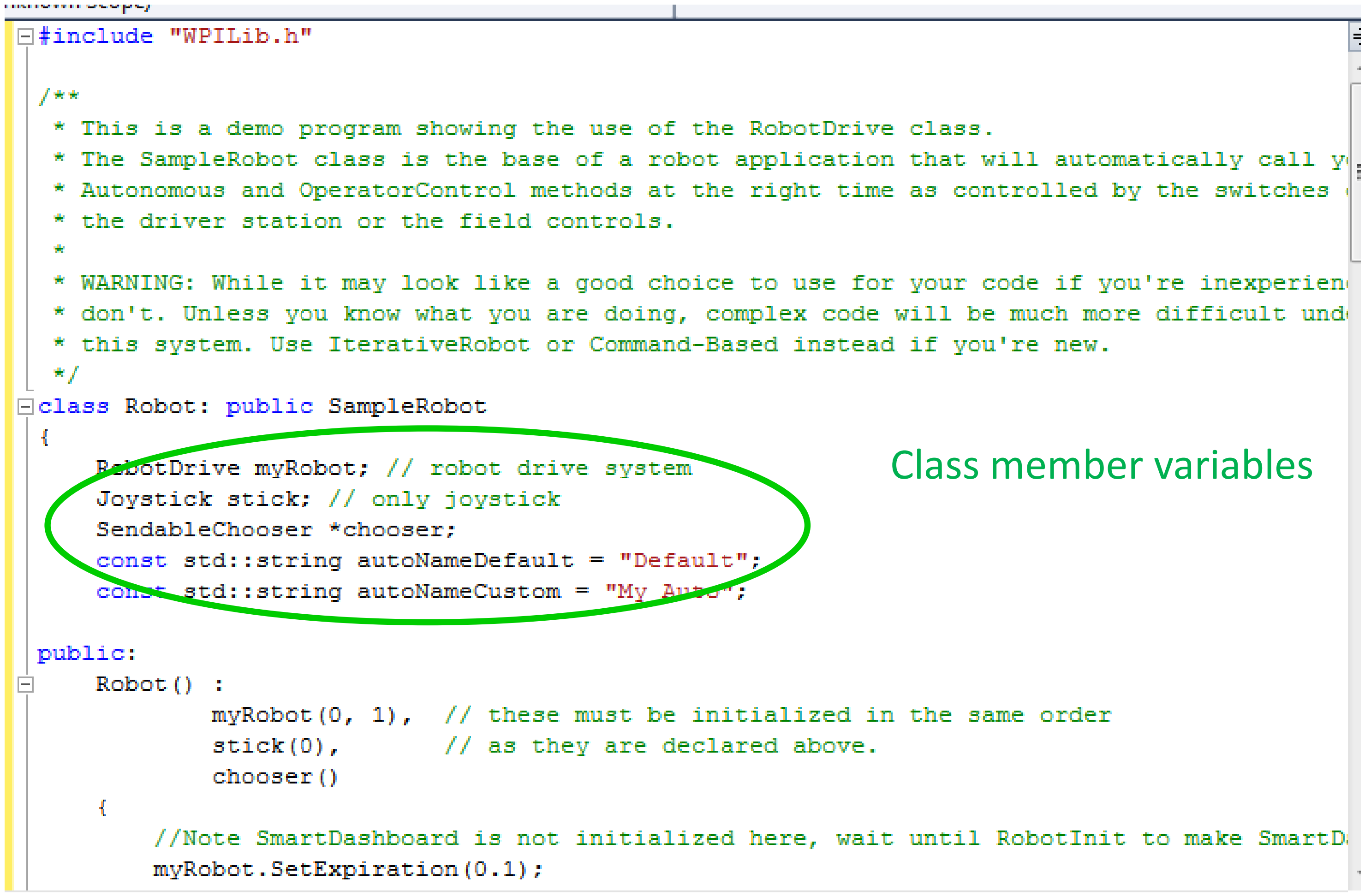
If *step* is 0, then *statement2* is run; otherwise if *step* is 1, then *statement3* is run; otherwise *statement1* is run.

```
if(step == 0) {
    statement2;
}
else if (step == 1) {
    statement3;
}
else {
    statement1;
}
```

Sample Robot Project

- A project is made from header (.h) and source (.cpp) files.
- The header file is where the class is defined (i.e. function prototypes and member variables).
- The source file contains the code for the class' functions.
- Robot.cpp is unique
  - This is the main file that the WPI Library calls into.
  - There is no header file.
  - RoboInit() – where we initialize our specific variables
  - Autonomous() – the code we want to run during the first 15 second of the match goes here
  - OperatorControl() – the code we want to run for the rest of the match

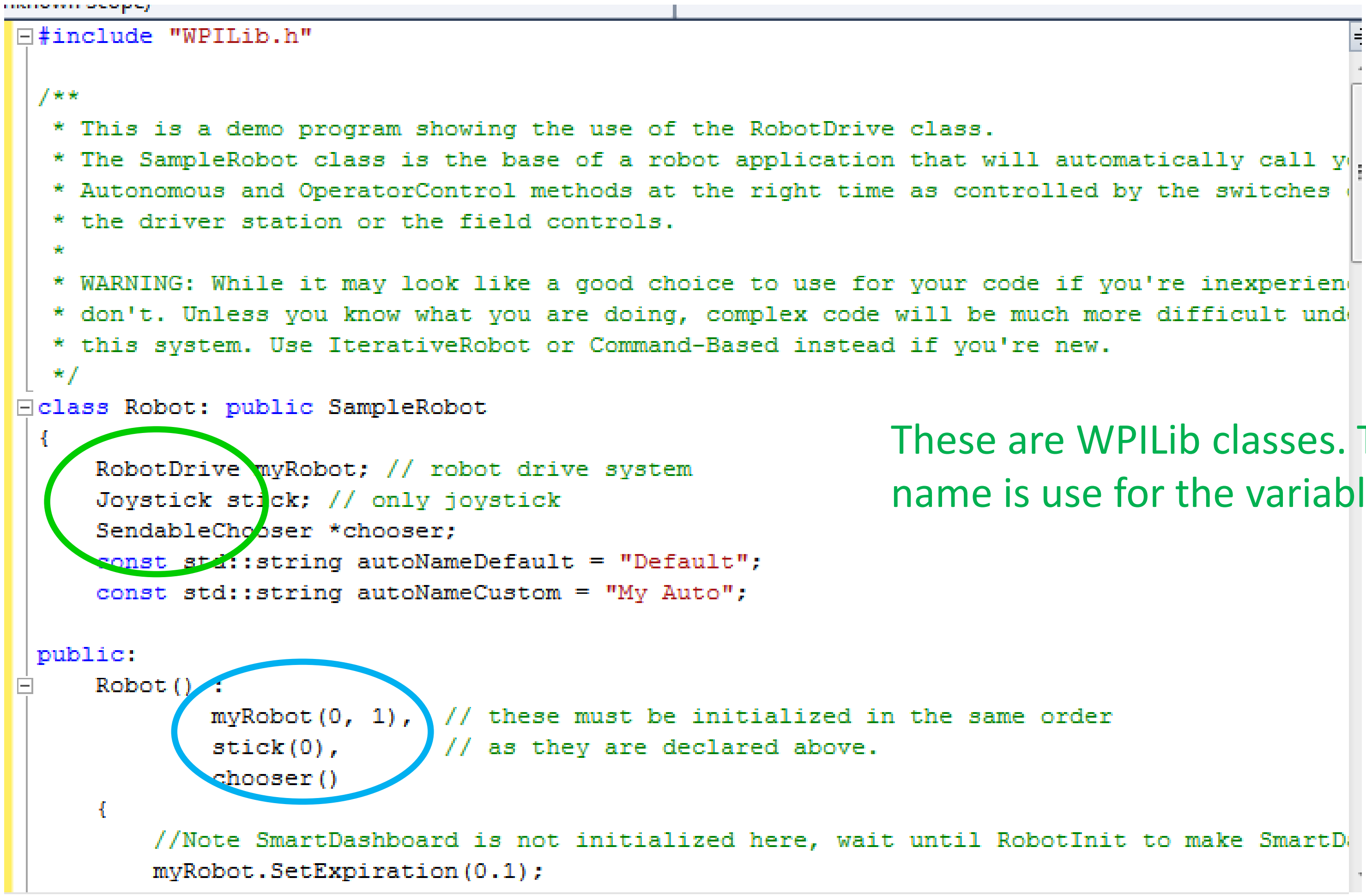
Sample Robot Project



```
#include "WPILib.h"

/**
 * This is a demo program showing the use of the RobotDrive class.
 * The SampleRobot class is the base of a robot application that will automatically call y
 * Autonomous and OperatorControl methods at the right time as controlled by the switches
 * the driver station or the field controls.
 *
 * WARNING: While it may look like a good choice to use for your code if you're inexperien
 * don't. Unless you know what you are doing, complex code will be much more difficult und
 * this system. Use IterativeRobot or Command-Based instead if you're new.
 */
class Robot: public SampleRobot
{
	RobotDrive myRobot; // robot drive system
	Joystick stick; // only joystick
	SendableChooser *chooser;
	const std::string autoNameDefault = "Default";
	const std::string autoNameCustom = "My Auto";

public:
	Robot() :
			myRobot(0, 1),	// these must be initialized in the same order
			stick(0),		// as they are declared above.
			chooser()
	{
		//Note SmartDashboard is not initialized here, wait until RobotInit to make SmartD
		myRobot.SetExpiration(0.1);
```

Class member variables

Sample Robot Project

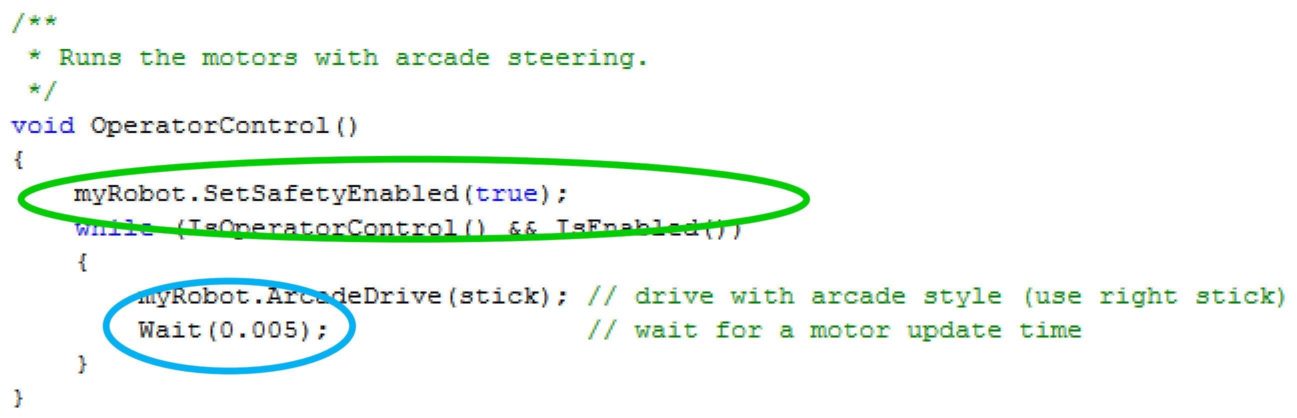


```
#include "WPILib.h"

/**
 * This is a demo program showing the use of the RobotDrive class.
 * The SampleRobot class is the base of a robot application that will automatically call y
 * Autonomous and OperatorControl methods at the right time as controlled by the switches
 * the driver station or the field controls.
 *
 * WARNING: While it may look like a good choice to use for your code if you're inexperien
 * don't. Unless you know what you are doing, complex code will be much more difficult und
 * this system. Use IterativeRobot or Command-Based instead if you're new.
 */
class Robot: public SampleRobot
{
    RobotDrive myRobot; // robot drive system
    Joystick stick; // only joystick
    SendableChooser *chooser;
    const std::string autoNameDefault = "Default";
    const std::string autoNameCustom = "My Auto";

public:
    Robot() :
            myRobot(0, 1),  // these must be initialized in the same order
            stick(0),       // as they are declared above.
            chooser()
    {
        //Note SmartDashboard is not initialized here, wait until RobotInit to make SmartD
        myRobot.SetExpiration(0.1);
```

These are WPILib classes. The class name is use for the variable type.

Initializes the class based variables by calling the class constructors. NOTE: the preceding colon, that they are comma separated, and they use the variable name.

Sample Robot Project

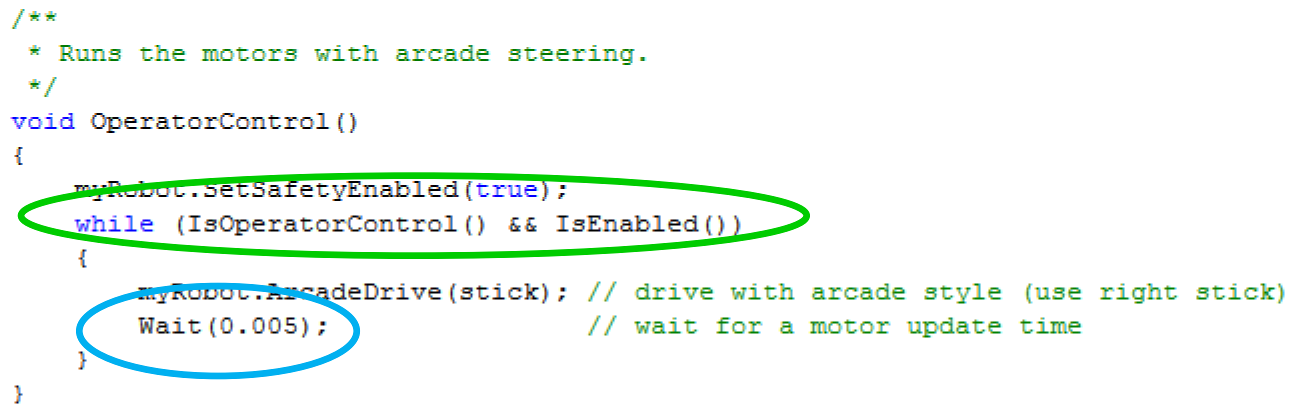


```
/**
 * Runs the motors with arcade steering.
 */
void OperatorControl()
{
    myRobot.SetSafetyEnabled(true);
    while (IsOperatorControl() && IsEnabled())
    {
        myRobot.ArcadeDrive(stick); // drive with arcade style (use right stick)
        Wait(0.005);                // wait for a motor update time
    }
}
```

myRobot is a variable; which is a class. This line shows calling the SetSafetyEnabled() function within class (i.e. variable name, then a dot, then the function name, followed by any parameters).

This shows calling a ‘regular’ method that is not part of any class.

Sample Robot Project



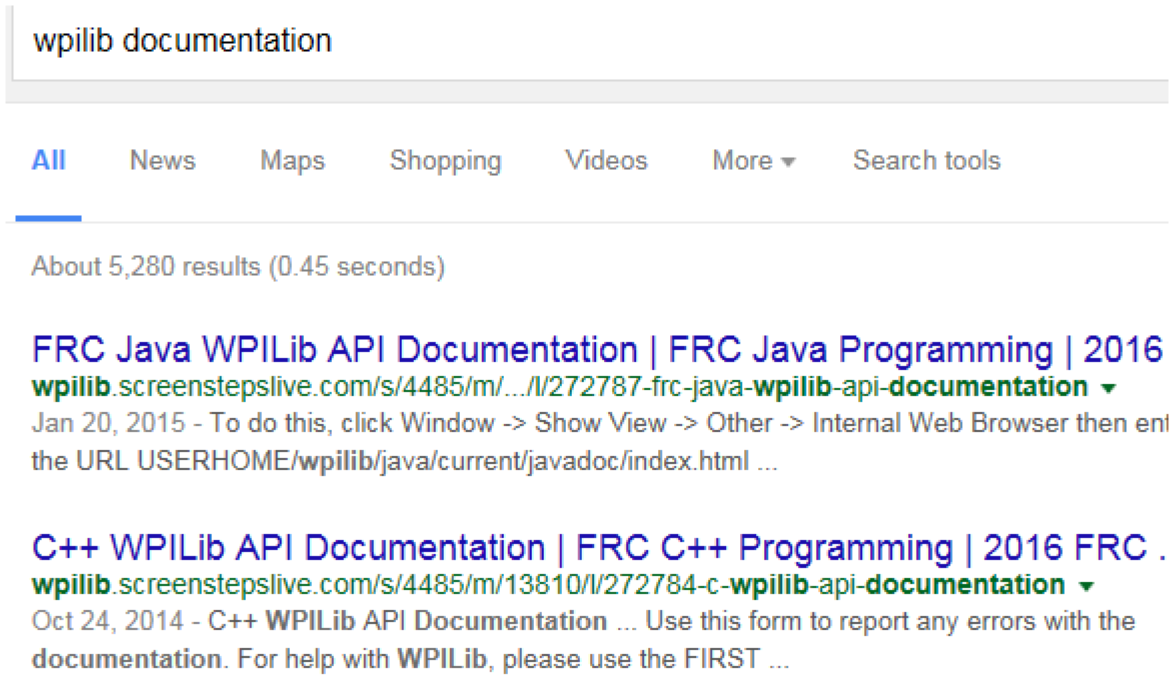
```
/**
 * Runs the motors with arcade steering.
 */
void OperatorControl()
{
    myRobot.SetSafetyEnabled(true);
    while (IsOperatorControl() && IsEnabled())
    {
        myRobot.ArcadeDrive(stick); // drive with arcade style (use right stick)
        Wait(0.005);                // wait for a motor update time
    }
}
```

Stay in this method while in tele-op mode AND while not disabled.

Waits for 5 milliseconds. This allows the WPILib to run its code (e.g. update motors, send data to dashboard, etc.). If this number is too small, then it doesn't give the motors & sensors a chance to update. If we this number it too big, then we will not check sensors often enough (e.g. could miss a sensor being blocked).

WPILib C++

- Search for "wpilib documentation"



wpilib documentation

All News Maps Shopping Videos More Search tools

About 5,280 results (0.45 seconds)

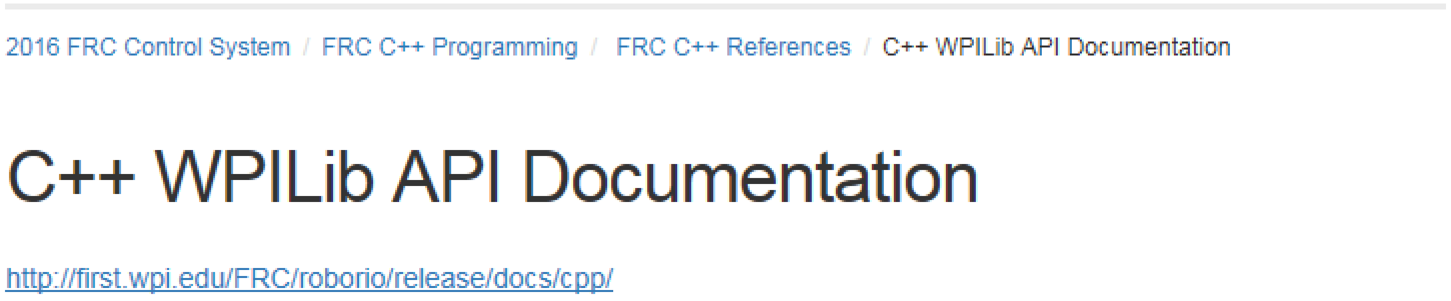
FRC Java WPILib API Documentation | FRC Java Programming | 2016
wpilib.screenstepslive.com/s/4485/m/.../l/272787-frc-java-wpilib-api-documentation
Jan 20, 2015 - To do this, click Window -> Show View -> Other -> Internal Web Browser then ent
the URL USERHOME/wpilib/java/current/javadoc/index.html ...

C++ WPILib API Documentation | FRC C++ Programming | 2016 FRC .
wpilib.screenstepslive.com/s/4485/m/13810/l/272784-c-wpilib-api-documentation
Oct 24, 2014 - C++ WPILib API Documentation ... Use this form to report any errors with the
documentation. For help with WPILib, please use the FIRST ...

- Select the second link (the one for "C++ WPILib")

WPILib C++

- Click the link under “C++ WPILib”

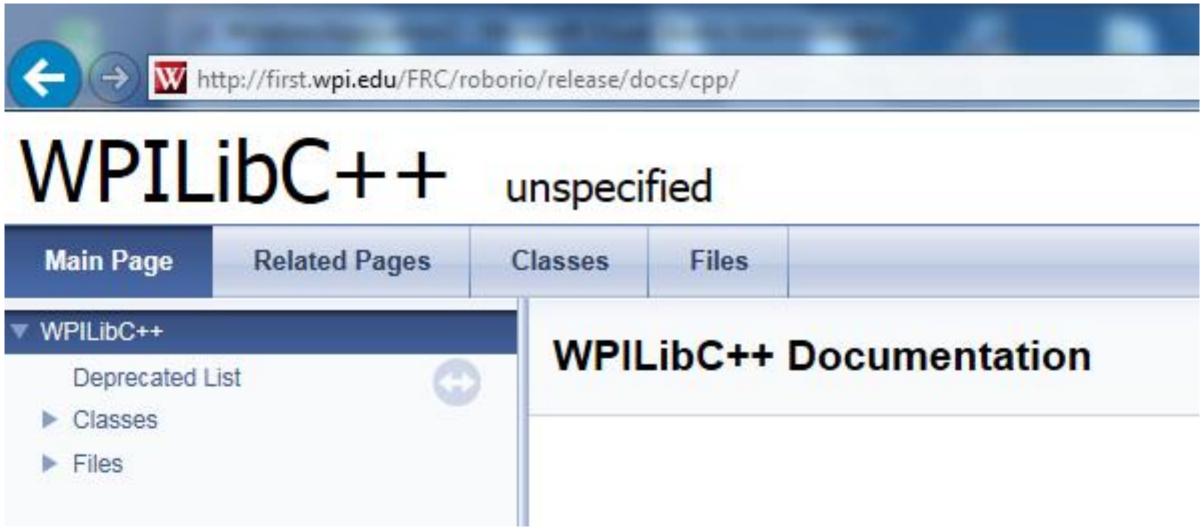


2016 FRC Control System / FRC C++ Programming / FRC C++ References / C++ WPILib API Documentation

C++ WPILib API Documentation

http://first.wpi.edu/FRC/roborio/release/docs/cpp/

- You are now at the documentation



http://first.wpi.edu/FRC/roborio/release/docs/cpp/

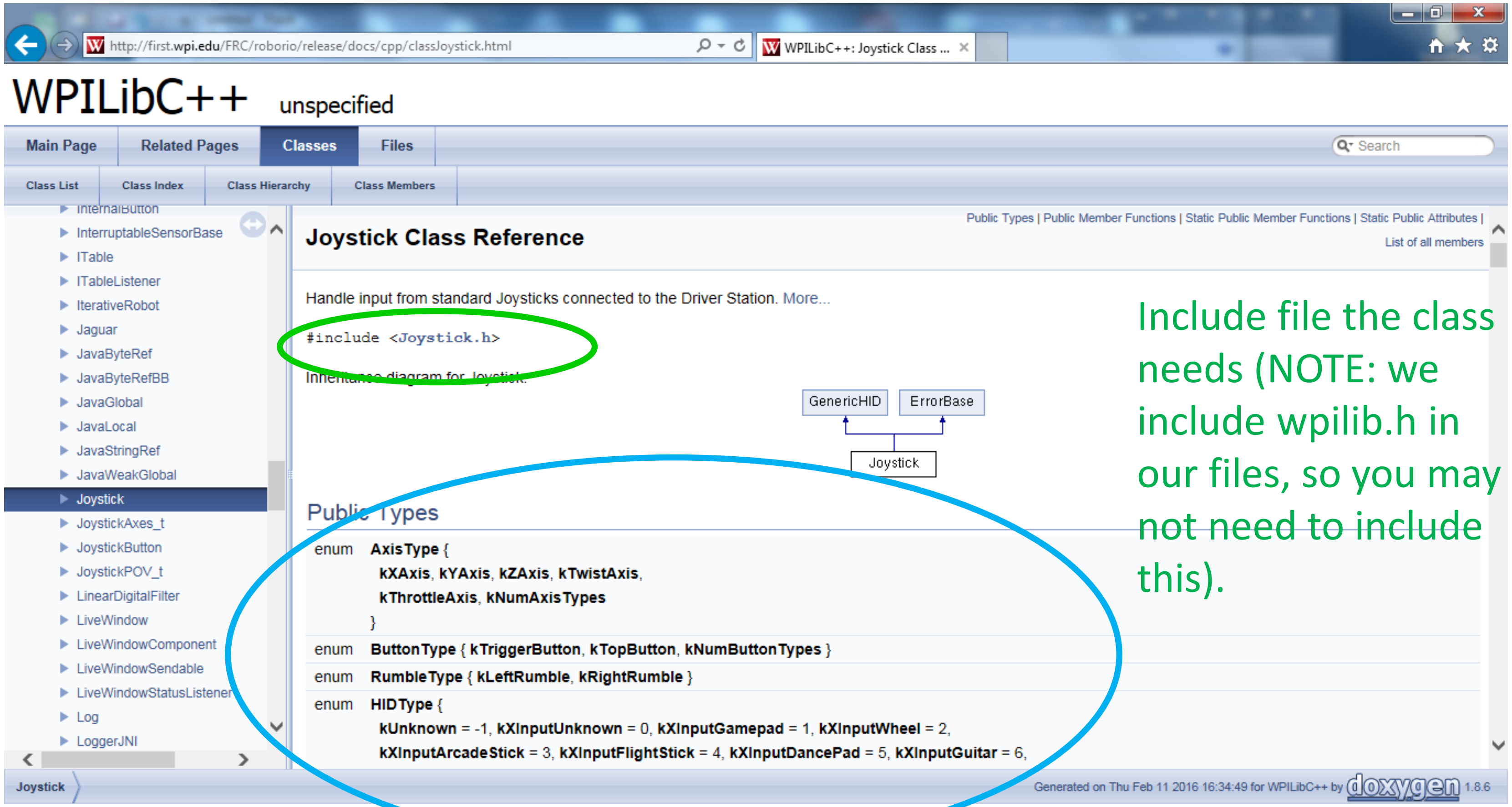
WPILibC++ unspecified

Main Page | Related Pages | Classes | Files

WPILibC++
- Deprecated List
- Classes
- Files

WPILibC++ Documentation

WPILib C++ - Example Class



http://first.wpi.edu/FRC/roborio/release/docs/cpp/classJoystick.html

WPILibC++: Joystick Class ...

WPILibC++ unspecified

Main Page | Related Pages | Classes | Files

Class List | Class Index | Class Hierarchy | Class Members

InternalButton
InterruptableSensorBase
ITable
ITableListener
IterativeRobot
Jaguar
JavaByteRef
JavaByteRefBB
JavaGlobal
JavaLocal
JavaStringRef
JavaWeakGlobal
Joystick
JoystickAxes_t
JoystickButton
JoystickPOV_t
LinearDigitalFilter
LiveWindow
LiveWindowComponent
LiveWindowSendable
LiveWindowStatusListener
Log
LoggerJNI

Public Types | Public Member Functions | Static Public Member Functions | Static Public Attributes | List of all members

Joystick Class Reference

Handle input from standard Joysticks connected to the Driver Station. More...

```
#include <Joystick.h>
```

Inheritance diagram for Joystick:

Public Types

```
enum  AxisType {
        kXAxis, kYAxis, kZAxis, kTwistAxis,
        kThrottleAxis, kNumAxisTypes
      }
enum  ButtonType { kTriggerButton, kTopButton, kNumButtonTypes }
enum  RumbleType { kLeftRumble, kRightRumble }
enum  HIDType {
        kUnknown = -1, kXInputUnknown = 0, kXInputGamepad = 1, kXInputWheel = 2,
        kXInputArcadeStick = 3, kXInputFlightStick = 4, kXInputDancePad = 5, kXInputGuitar = 6,
```

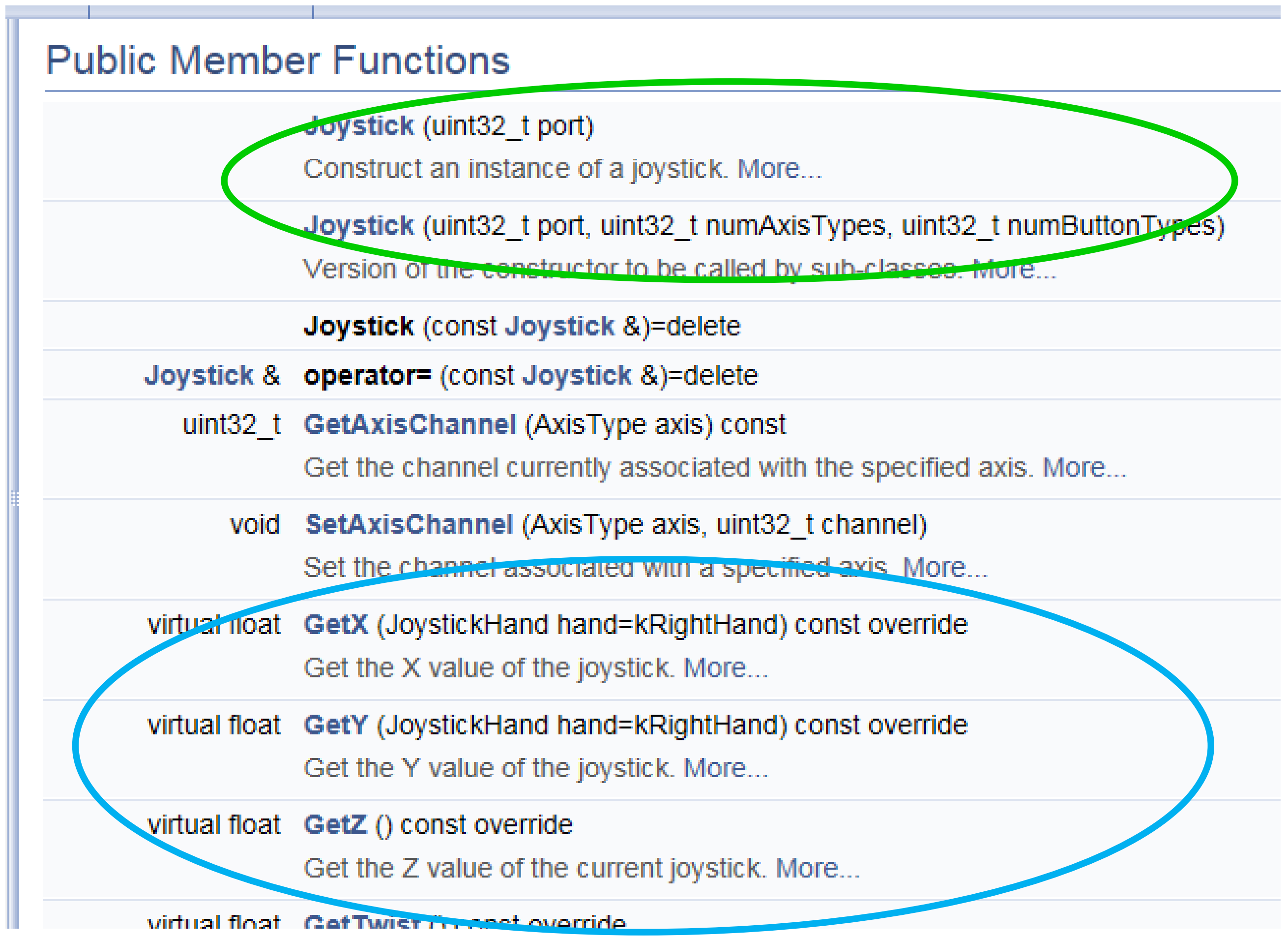
Joystick

Generated on Thu Feb 11 2016 16:34:49 for WPILibC++ by doxygen 1.8.6

Include file the class needs (NOTE: we include wpilib.h in our files, so you may not need to include this).

Any types the class uses (e.g. parameters to methods)

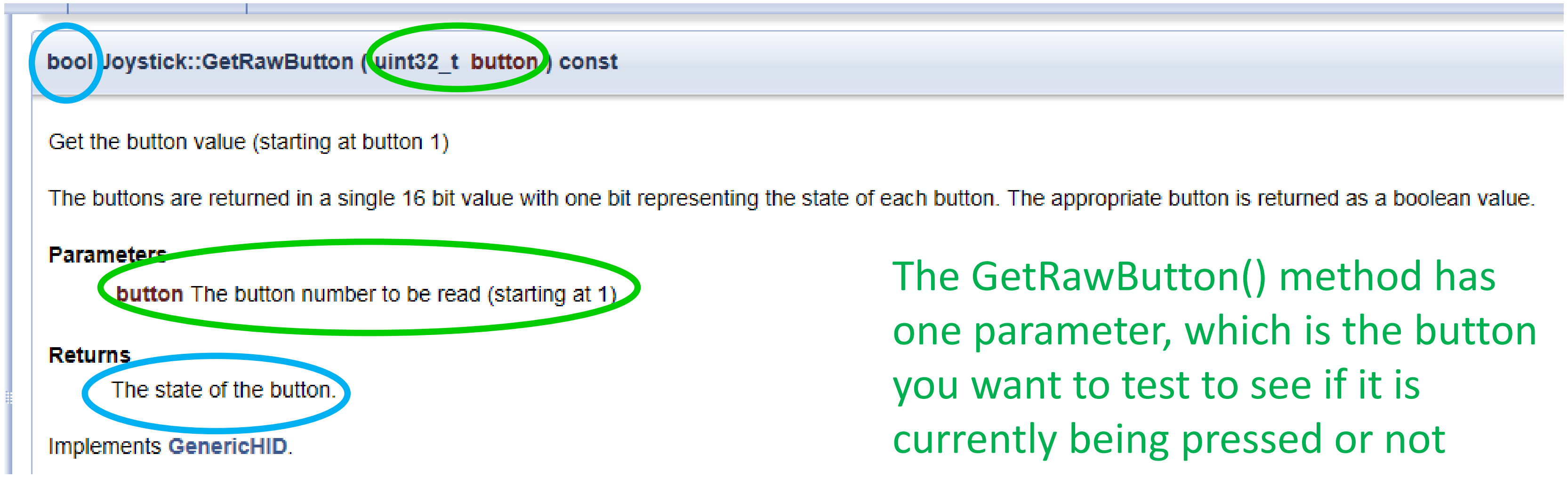
WPILib C++ - Example Class



Public Member Functions

	Joystick (uint32_t port) Construct an instance of a joystick. More...
	Joystick (uint32_t port, uint32_t numAxisTypes, uint32_t numButtonTypes) Version of the constructor to be called by sub-classes. More...
	Joystick (const **Joystick** &)=delete
Joystick &	**operator=** (const **Joystick** &)=delete
uint32_t	**GetAxisChannel** (AxisType axis) const Get the channel currently associated with the specified axis. More...
void	**SetAxisChannel** (AxisType axis, uint32_t channel) Set the channel associated with a specified axis. More...
virtual float	**GetX** (JoystickHand hand=kRightHand) const override Get the X value of the joystick. More...
virtual float	**GetY** (JoystickHand hand=kRightHand) const override Get the Y value of the joystick. More...
virtual float	**GetZ** () const override Get the Z value of the current joystick. More...
virtual float	**GetTwist** () const override

WPILib C++ - Example Class



bool Joystick::GetRawButton (uint32_t button) const

Get the button value (starting at button 1)

The buttons are returned in a single 16 bit value with one bit representing the state of each button. The appropriate button is returned as a boolean value.

Parameters

button The button number to be read (starting at 1)

Returns

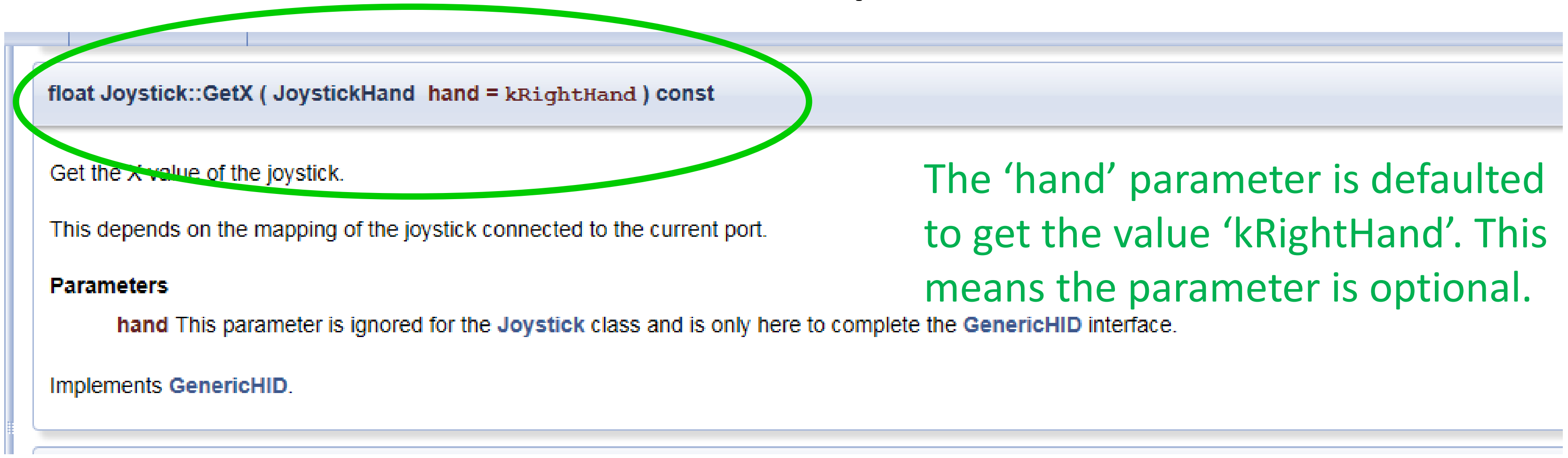
The state of the button.

Implements **GenericHID**.

The GetRawButton() method has one parameter, which is the button you want to test to see if it is currently being pressed or not

Returns a Boolean indicating if button is pressed or not.

WPILib C++ - Example Class



```
float Joystick::GetX ( JoystickHand  hand = kRightHand ) const
```

Get the X value of the joystick.

This depends on the mapping of the joystick connected to the current port.

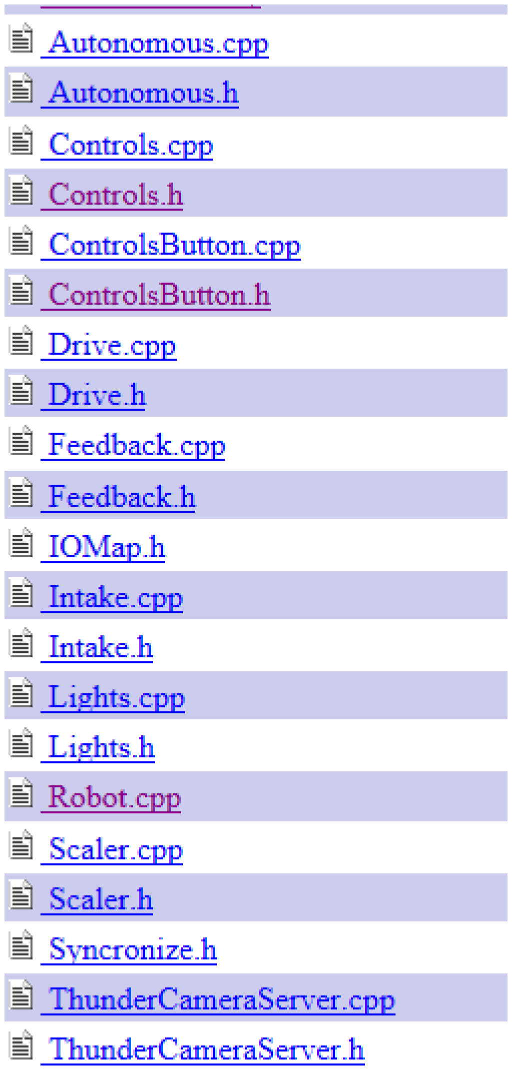
Parameters

hand This parameter is ignored for the **Joystick** class and is only here to complete the **GenericHID** interface.

Implements **GenericHID**.

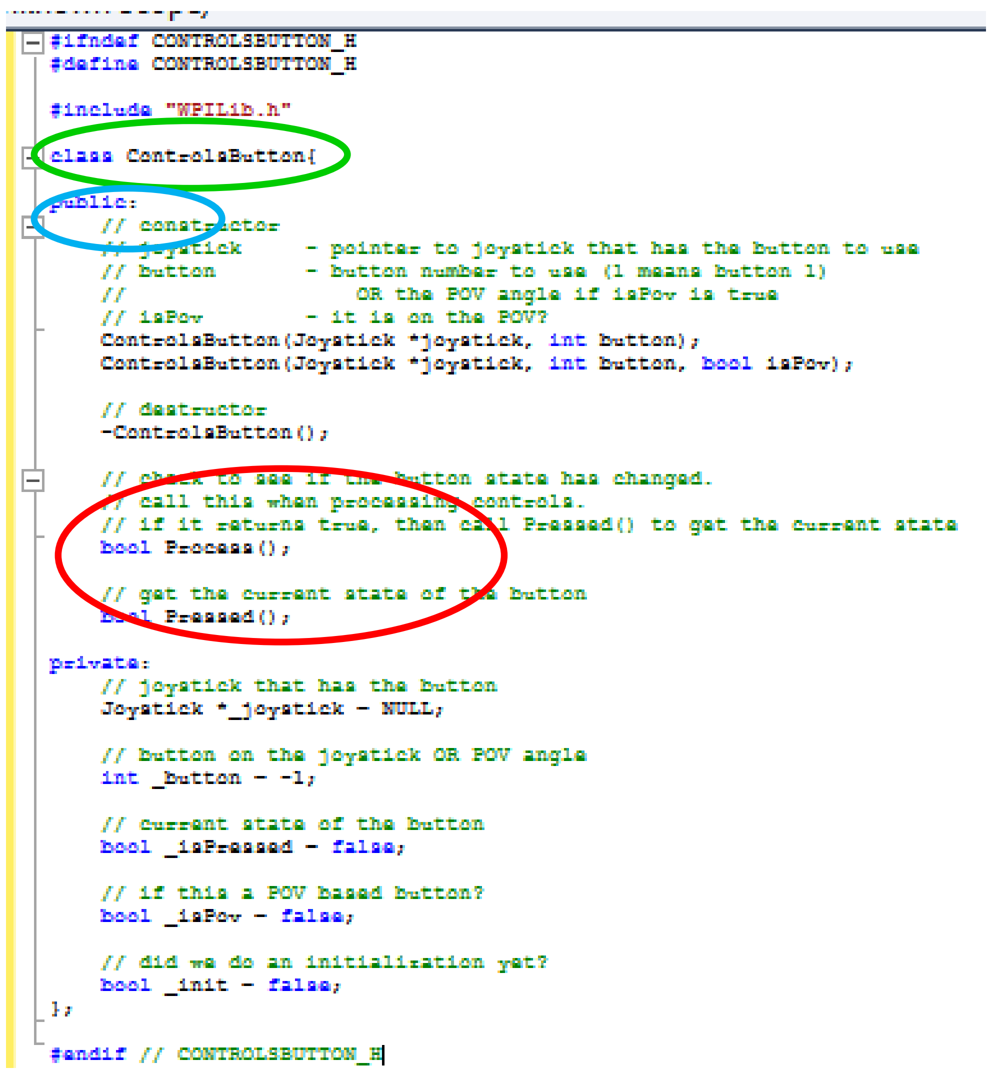
The 'hand' parameter is defaulted to get the value 'kRightHand'. This means the parameter is optional.

Team 1511: 2016 Robot Files



Autonomous.cpp Autonomous.h	auto
Controls.cpp Controls.h	operator controls
ControlsButton.cpp ControlsButton.h	helper class for button
Drive.cpp Drive.h	drivetrain (i.e. wheels)
Feedback.cpp Feedback.h	send info to dashboard
IOMap.h	
Intake.cpp Intake.h	breacher/front bucket
Lights.cpp Lights.h	LED lights (were not used)
Robot.cpp	
Scaler.cpp Scaler.h	arm/lifter
Syncronize.h ThunderCameraServer.cpp ThunderCameraServer.h	custom code to handle 2 camera

Team 1511: 2016 Robot Files - .h



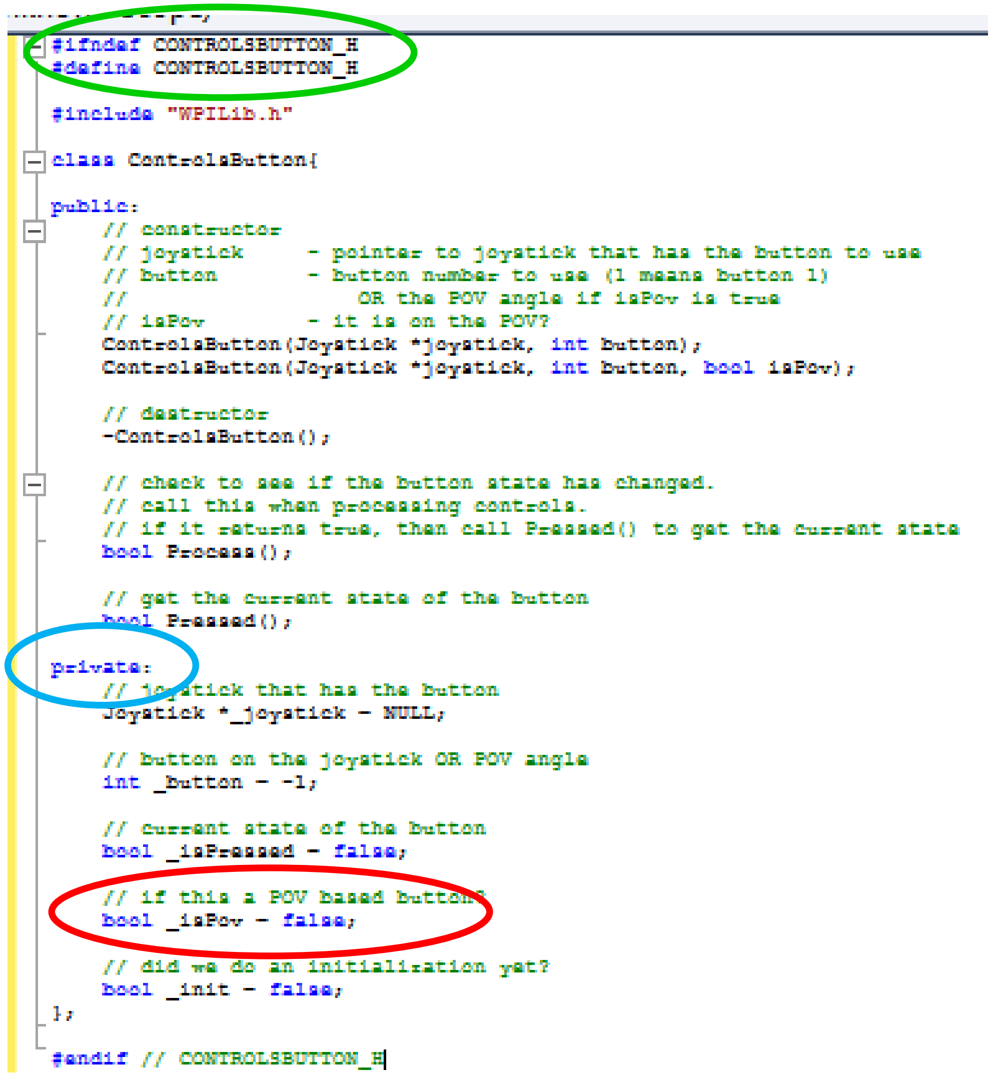
```
#ifndef CONTROLSBUTTON_H
#define CONTROLSBUTTON_H

#include "WPILib.h"

class ControlsButton{

public:
    // constructor
    // joystick      - pointer to joystick that has the button to use
    // button        - button number to use (1 means button 1)
    //                   OR the POV angle if isPov is true
    // isPov         - it is on the POV?
    ControlsButton(Joystick *joystick, int button);
    ControlsButton(Joystick *joystick, int button, bool isPov);

    // destructor
    ~ControlsButton();

    // check to see if the button state has changed.
    // call this when processing controls.
    // if it returns true, then call Pressed() to get the current state
    bool Process();

    // get the current state of the button
    bool Pressed();

private:
    // joystick that has the button
    Joystick *_joystick = NULL;

    // button on the joystick OR POV angle
    int _button = -1;

    // current state of the button
    bool _isPressed = false;

    // if this a POV based button?
    bool _isPov = false;

    // did we do an initialization yet?
    bool _init = false;
};

#endif // CONTROLSBUTTON_H
```

Defines a class named: ControlButton

Indicates that all items in this section can be seen by other files. Typically we put in functions we want others to call. This is the ‘contract’ between other parts of the code and this class. Almost never put variables here.

Member functions for others to call.

Team 1511: 2016 Robot Files - .h



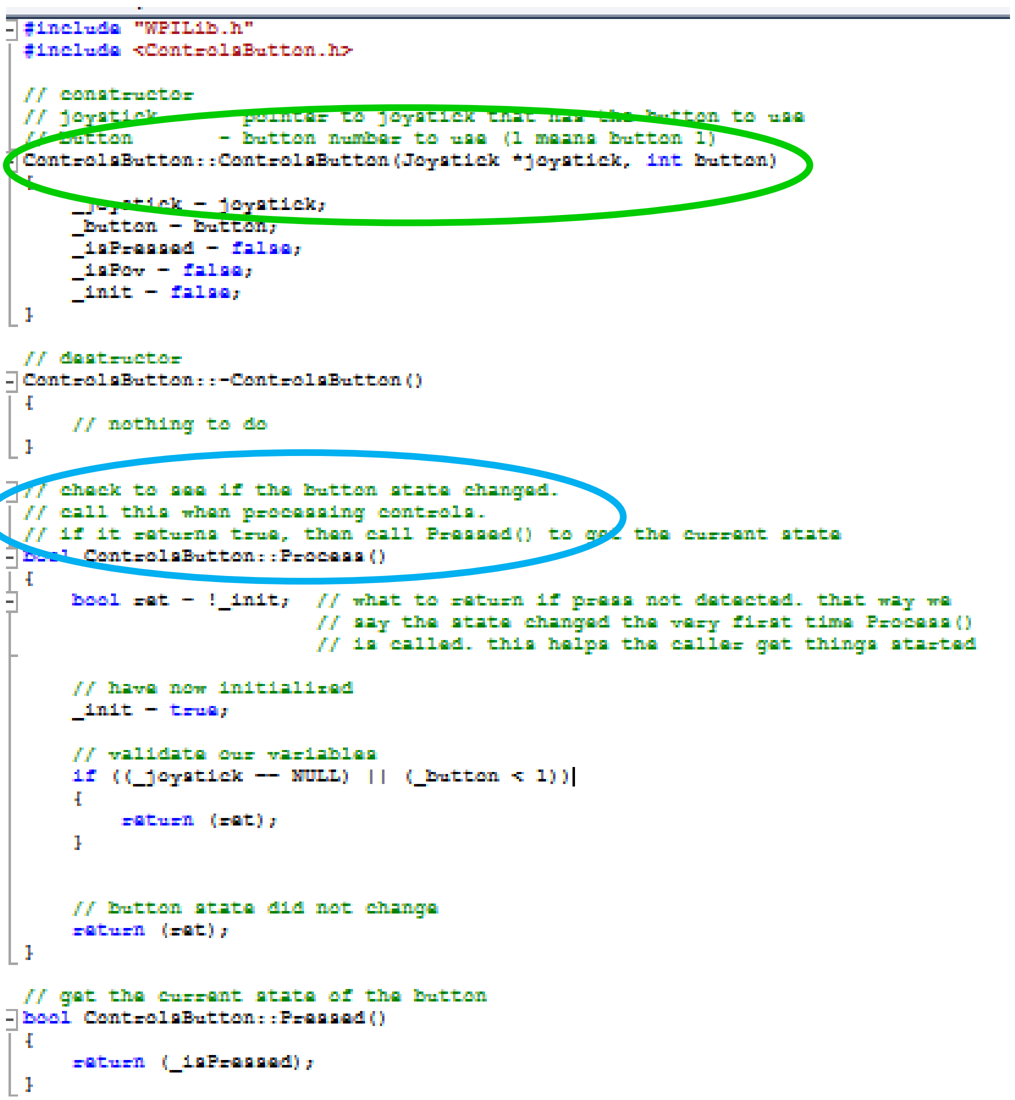
```
#ifndef CONTROLSBUTTON_H
#define CONTROLSBUTTON_H

#include "WPILib.h"

class ControlsButton{

public:
    // constructor
    // joystick     - pointer to joystick that has the button to use
    // button       - button number to use (1 means button 1)
    //                OR the POV angle if isPov is true
    // isPov        - it is on the POV?
    ControlsButton(Joystick *joystick, int button);
    ControlsButton(Joystick *joystick, int button, bool isPov);

    // destructor
    ~ControlsButton();

    // check to see if the button state has changed.
    // call this when processing controls.
    // if it returns true, then call Pressed() to get the current state
    bool Process();

    // get the current state of the button
    bool Pressed();

private:
    // joystick that has the button
    Joystick *_joystick = NULL;

    // button on the joystick OR POV angle
    int _button = -1;

    // current state of the button
    bool _isPressed = false;

    // is this a POV based button?
    bool _isPov = false;

    // did we do an initialization yet?
    bool _init = false;
};

#endif // CONTROLSBUTTON_H
```

This define helps speed compiling and prevents the class from being defined twice. This way, if a .cpp file includes header files that happen to also include this header file, there will not be a problem.

Indicates that all items in this section can only be seen by this class (i.e. in controlsbutton.cpp). Can be variables and methods.

Member variable (for 2016 it was decided that they start with an underscore), it is initialized to *false*.

Team 1511: 2016 Robot Files - .cpp



```
#include "WPILib.h"
#include <ControlsButton.h>

// constructor
// joystick        - pointer to joystick that has the button to use
// button          - button number to use (1 means button 1)
ControlsButton::ControlsButton(Joystick *joystick, int button)
{
    _joystick = joystick;
    _button = button;
    _isPressed = false;
    _isPov = false;
    _init = false;
}

// destructor
ControlsButton::~ControlsButton()
{
    // nothing to do
}

// check to see if the button state changed.
// call this when processing controls.
// if it returns true, then call Pressed() to get the current state
bool ControlsButton::Process()
{
    bool ret = !_init;  // what to return if press not detected. that way we
                        // say the state changed the very first time Process()
                        // is called. this helps the caller get things started

    // have now initialized
    _init = true;

    // validate our variables
    if ((_joystick == NULL) || (_button < 1))
    {
        return (ret);
    }


    // button state did not change
    return (ret);
}

// get the current state of the button
bool ControlsButton::Pressed()
{
    return (_isPressed);
}
```

Constructor (method name is the same as the class name).

Public method Process(). Typically we have a method called this in every class. It is called during Autonomous() and OperatorControl() to do the work the class needs to do for each iteration of the loop.