Object Oriented C in Botball
Justin Yu
Los Altos Community Team 16-0323

Object Oriented C in Botball

Abstract. Over the years, many teams have developed a set of libraries that perform basic tasks such as driving forward a given distance, turning with a given radius, or setting a servo's position slowly with a given duration. These libraries are, in many aspects, good ways to reuse code from year to year. However, given that the KISSIDE no longer supports object oriented languages such as C++ or Python, the libraries are often cluttered with constants that need to be manually customized. In order to address these issues, I created a set of libraries for my team based on a more intuitive approach, targeting those who are familiar with or interested in learning object oriented programming (OOP).

1 Benefits of Object Oriented Programming

Why should any team switch to object oriented style in the first place? Wouldn't the traditional imperative style of C do the same thing? The short answer is: yes, OOP would accomplish the same thing. However, though it may seem like a burden at first, there are actually many benefits to adopting an OO paradigm.

1.1 Reusability

As mentioned before, OOP helps with the reusability of code from year to year. Instead of having to modify the actual source of the libraries, teams only need to instantiate a new object, feeding the the specs of that year's robot into the constructor. In section 3, this concept is explained in more detail.

1.2 Encapsulation

Currently, many teams resort to using macros (`#define`) to declare global constants. Such constants would include motor ports, the wheel diameter (used in calculating distance), the horizontal distance between wheels (used in calculating radius), etc. While this is not necessarily a bad thing, too many constants tend to clutter the code, making debugging and code revisioning more difficult.

With an object oriented approach, these constants would be encapsulated in classes. For example, all methods and properties related to the Wallaby controller would be encapsulated in a Controller class; the same goes to the Create and the camera:

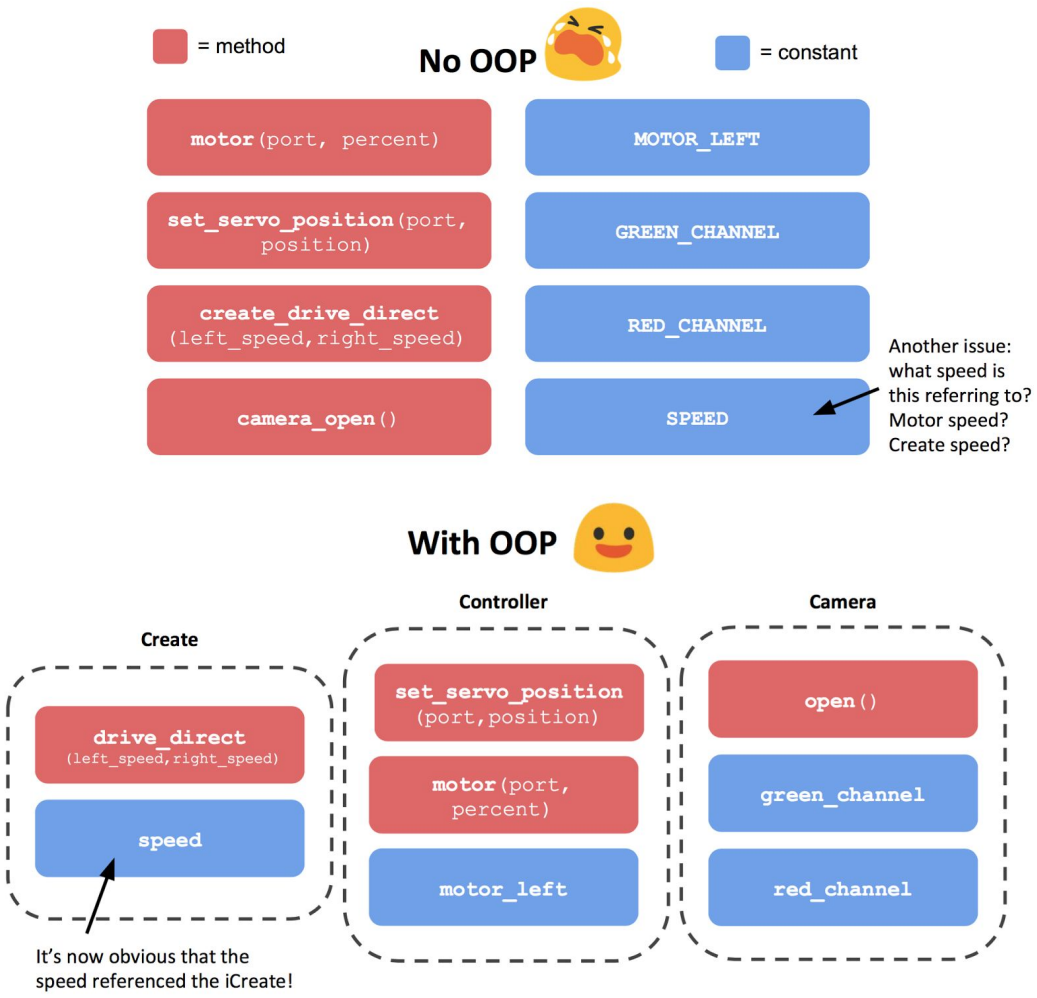

Figure 1

OOP makes it much easier to designate a property or function's purpose. Furthermore, integrating object oriented programming into Botball helps new programmers visualize and better understand the program as they are able to interface with an object in code that is also tangible in reality.

2 Methodology

In creating a new set of OOP libraries, I tried to mimic other OO languages like Java as much as I c-

ould, though limited by constraints of the C language. In each section, explanations are best understood through the code examples.

2.1 C `struct` and Classes

```
typedef struct MyClass { ... } MyClass; // class declaration
extern MyClass new_object(); // constructor

MyClass object; // global variable
```

In C, there is no such thing as a **`class`** keyword that you may find in other languages such as C++, Java, etc. However, the C struct is similar in many ways. A struct is a data structure that contains a group of variables called properties or attributes. The main caveat of structs is that functions cannot be defined within them, nor can they inherit from other structs [1].

To assign a custom type name to a struct, I used the **`typedef`** keyword [2]. The constructor, used to instantiate new instances of this class, is declared as a global function. The global variable `object` is used throughout the implementation as a substitution for the **`this`** keyword.

In the code snippet, the constructor `new_object` returns a struct with the type `MyClass`, and it is possible to create a variable (`object`) that has the type MyClass as well.

2.2 Properties (Instance Variables)

In OOP, a very important concept is that every instance of a class should be able to encapsulate its own variables, hence the term "instance" variable. Adding properties to the previously defined class is relatively easy:

```
typedef struct MyClass {
    int property;
} MyClass;
```

Using dot notation, it is possible to access and modify the instance variable, `property`. For example, I could print out the value using `printf("%d", object.property);` And I could assign a new value using `object.property = ...;`

2.3 Methods

Since structs do not allow function definitions or declarations inside them, assigning methods is trickier. While a function itself cannot be stored within a struct, the *address* of the function can be stored; the solution is to use a **function pointer** (a pointer that contains the address of a function) [3]. Adding a few methods to our class, the class definition now becomes:

```
typedef struct MyClass {
    int property;

    void (*instance_method)();
    int (*instance_method_with_params)(int x, int y);
} MyClass;
```

Notice how the syntax is fairly straightforward between the two functions, using the `*` to denote the pointer. After an object has been instantiated, I could call either function using dot notation (i.e. `object.instance_method()`).

2.4 Constructor

Everything up to this point has been declared inside a header file (*.h*), but to actually implement the aforementioned properties, instance methods and constructor, there also needs to be an implementation file (*.c*) similar to the ones that follows:

```
static void instance_method() { ... }
static int method_with_params(int x, int y) { ... }

MyClass new_object() {
    MyClass instance = {
        .property = 4,
        .instance_method = &instance_method,
        .instance_method_with_params = &method_with_params
    };
    object = instance; // workaround for `this` keyword
    return instance;
}
```

In this code snippet, the constructor, `new_object`, creates and returns an instance of the class. Within the instantiation of the struct are property assignments. The `property` instance variable, being an integer, is assigned a value of four while the two pointer properties are assigned the address of their corresponding functions using the `&` operator. While the name of the method in

the implementation file doesn't necessarily need to match the name of the property, **the *parameter names* and *return type* do.** And although there are no arguments passed into the constructor, it is possible — and in many cases very useful — to customize how the object is created through parameters.

3 Components

My approach to modularizing the functionality of the KIPR library was to separate it into three components: 1) a Create class 2) a Controller class 3) a Camera class. All of the code mentioned is found here: github.com/justinvyu/botball/tree/master/classes**.**

3.1 Create

The Create class refers to the iRobot Create, containing all the necessary movement, sensor, and OI functionality. Referencing a global create variable defined in the class's header file (similar to the global `object` variable defined in the previous section), a Create object can be instantiated using the `new_create` constructor.

Additionally, most of the Create functionality provided by KIPR has been ported, with minor modifications such as the truncation of method names to stay consistent with object oriented style. Other miscellaneous movement methods have been implemented (i.e. `forward`, `backward`, `left`, `right`, etc.), which shows how OOP makes it easier to organize and add to the complexity of a program.

```
create = new_create();

create.connect();
create.drive_direct(..., ...);

create.forward(10, 250); // distance, speed
create.left(90, 0, 250); // distance, radius, speed
```

3.2 Controller

The Controller class refers to the Wallaby, and it encompasses all the methods dealing with motor movement, servo positions, and sensor outputs.

A controller can be instantiated using one of *two* constructors, the default being `new_controller`, which takes in four parameters including the motor ports for wheels and other relevant measurements. The *alternate constructor*, `new_create_controller`, assumes

that the controller being instantiated is attached to an iRobot Create, meaning that motor ports and wheel measurements would be unnecessary since all movement is handled by a separate Create object. Again, a global variable `controller` is used, many KIPR functions have been ported, and miscellaneous movement methods have been implemented.

```
// left_motor, right_motor,
// distance_between_wheels, wheel_diameter
controller = new_controller(0, 1, 14.5, 5.0);
controller = new_create_controller(); // alternate constructor

controller.motor(controller.motor_left, 100);
controller.forward(10, 90); // distance, speed (0-100)
```

3.3 Camera

Finally, the Camera class encapsulates the functionality of the camera, another important piece of hardware provided in the kit.

```
camera = new_camera();

camera.open();
printf("%d", camera.get_object_count(..., ...));
```

3.4 Reusability (cont.)

Source code changes each year
Hard to understand

```
#define MOTOR_LEFT 3
#define MOTOR_RIGHT 0
#define SPD 100 //turning
#define SPDl 90. //left forward
#define SPDr 90. //right forward
#define rdistmult 1.0
#define SPDlb 8.  //left backward
#define SPDrb 8.  //right backward
#define rdistmultb (SPDrb/SPDlb)
#define wheeldiameter 5.0
#define ks 14.5 // Distance from one wheel to the other
```

Source code stays the same each year
Parameters change depending on the robot
Easier to understand

```
extern Controller new_controller(int motor_left, int motor_right,
                                 float distance_between_wheels, float wheel_diameter);
```

4 Inheritance

The last major aspect that object oriented programming is attributed with is inheritance. "Real" inheritance is not possible in C without the help of some external libraries, but it can be achieved on a much broader scale. Inheritance means that a child class can access all the properties and methods of its parent class. In other words, a child class should **encapsulate all the complexity of its parent class, providing yet another layer of abstraction.** In the example below, the Robot class "inherits" all the functionality provided in the three component classes.

```
typedef struct Robot {
    Controller controller;
    Create create;
    Camera camera;
} Robot;

extern Robot new_robot();
```

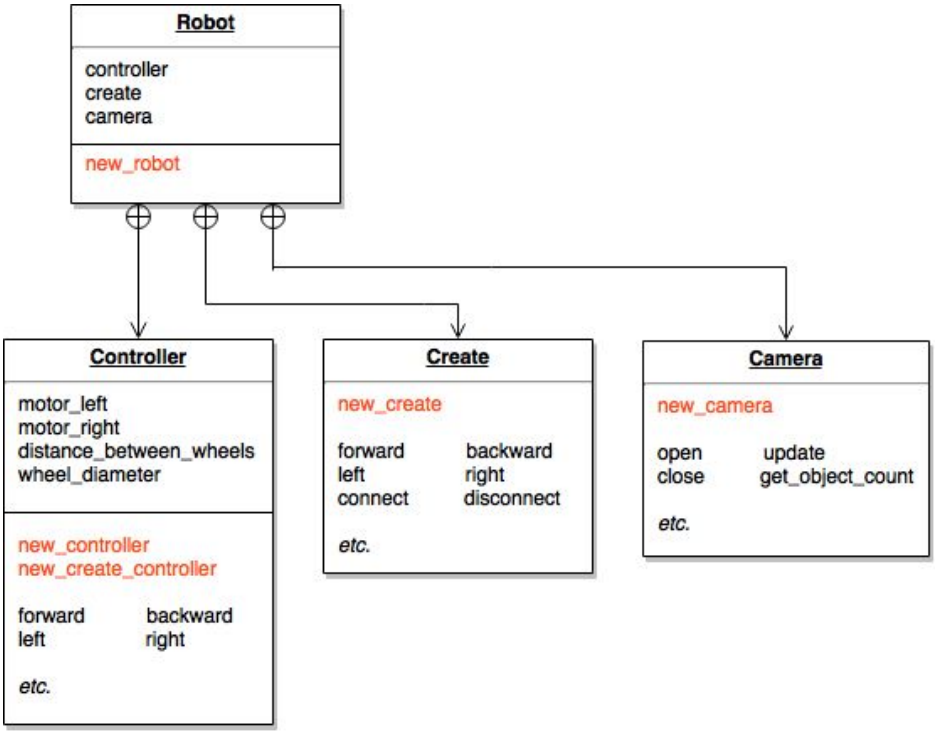

Figure 2

Above is a diagram that models the organization of the Robot class. Within the Robot class are three properties (controller, create, and camera) that all reference the bottom three component

classes. It's important to note that by instantiating a new Robot object, a new controller object, create object, and camera object are also created.

The robot object, therefore, would "own" a controller, a create, and a camera (like an actual robot!). Each of the objects is instantiated in the subclass constructor after assigning all custom properties and methods, looking something like this:

```
Robot new_robot() {
    Robot instance = { ... };

    instance.controller = new_create_controller();
    instance.create = new_create();
    instance.camera = new_camera();
    return instance;
}
```

In this new child class, the properties and methods of the individual component objects can no longer be accessed directly, but the properties and methods defined in the child class still can. This means that by referencing a component object **using dot notation, the child class can then indirectly access all other functionality** [3].

```
robot = new_robot();
robot.controller.motor(..., ...);
robot.create.drive_direct(..., ...);
robot.camera.open();
```

5 Summary

As a brief summary, object oriented Botball allows for improved reusability of code from year to year and the encapsulation of properties and functions. Classes are defined in header files, and the properties and methods declared within them are implemented in a source file. The three example component classes — Create, Controller, and Camera — demonstrate one possible implementation using OO principles, though there are still numerous implementations out there. Lastly, inheritance allows even more encapsulation.

If anything, object oriented Botball teaches important concepts such as defining classes and accessing / modifying instance properties while gaining an understanding of the intricacies of the C language. Please email me at justin.v.yu@gmail.com with any questions regarding either the libraries or the implementation.

Abbreviations

1. OO = object oriented
2. OOP = object oriented programming
3. KIPR = KISS Institute for Practical Robotics
4. KISSIDE = KISS's integrated development environment
5. OI = open interface

Sources

1. C - Structures,
   http://www.tutorialspoint.com/cprogramming/c_structures.htm
2. Object Oriented Programming in ANSI-C,
   https://www.cs.rit.edu/~ats/books/ooc.pdf
3. Classes in C
   http://www.pvv.org/~hakonhal/main.cgi/c/classes/