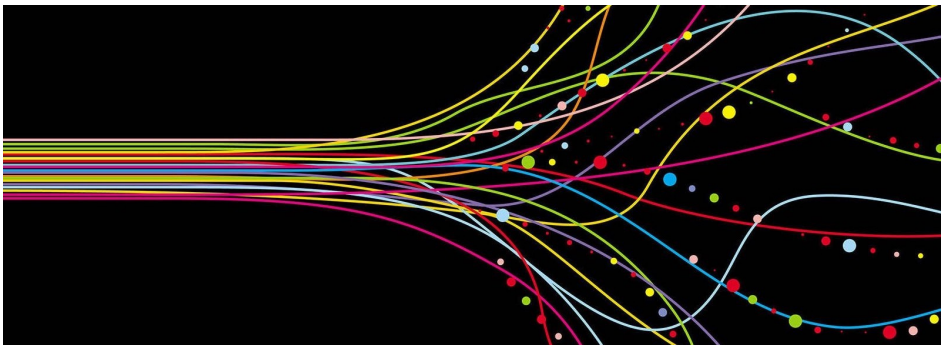
Concurrenct Object-Oriented C++ in Botball
Ziad Khattab
Qatar Academy

Concurrenct Object-Oriented C++ in Botball



1 Object-Oriented Architecture

1.1 Intro

As a Botball robot grows more complex and elaborate, it is a reasonable expectation that the codebase should grow to accommodate this change. As more motors and servos are added, they should be manipulated by interface functions that are easy and efficient to use and test. Robot movement should be abstracted entirely to constraints like overall movement distance, rather than fiddling with individual motors in order to simply move the robot forward.

Ultimately, applying this principle tends towards a certain type of architecture where it makes logical and logistical sense to group together the functions and variables concerning one part of the robot under one umbrella, such as each servo or sensor. Naming conventions can be somewhat helpful here, such as how camera-related functions are prefixed with "`camera`" (ie. `camera_open()`, `camera_update()`, etc). While naming conventions improve readability, from the perspective of the compiler, they are largely irrelevant. This means that while all camera-related functions may be appropriately prefixed, attempting to actually pass around some sort of unified, holistic "camera" is not actually possible.

Solutions to this problem ultimately lead to the metaphorical "Rome" of object-oriented programming, where all the variables and functions relevant to a the real-life representation of a physical object are grouped under some sort of programmatic structure that represents it; in this case that is a C++ `class`. For example, a robot's position on the board and its bearing (angle) could be stored as data members in a class, while the functions controlling its movement and turning would be methods in that class.

This approach helps significantly streamline efficiency, readability, and reusability of code in a project, especially with the help of C++'s `virtual` functions and polymorphic inheritance.

1.2 Implementation

Delving into specifics, there are different avenues of implementation that could be used to implement an object-oriented architecture for this sort of context.

The first approach is exclusive reliance on inheritance, where an abstract `class Robot` is defined by including a pure `virtual` (and a `virtual` destructor of course) using the suffix `= 0;` after the function prototype. `Robot` would only contain the variables and methods applicable to the robot itself independent of any servos, cameras, motors, or sensors, such as the robot's position and bearing. Its abstract status would indicate that it cannot operate in a standalone manner, which is indicative of a real-life Botball robot that is not fixed with any kind of parts or attachments.

This class would be accompanied by other classes representing the parts that would be added to a robot, such as a `class Servo`, `class Motor`, `class Camera`, etc. From here, a new child class of `Robot`, such as a `class Create`, would inherit from `Robot` as

well as from `Servo` and `Camera` as needed based on whether or not it will be fixed with these parts, as below:

```
class  Create : public Robot, public Servo, public Camera
{
    // Add class data here
};
```

The result is that `Create` contains all of the relevant data from `Robot`, `Servo`, and `Camera`. Taking this one step further, an abstract `class Sensor` could be defined, from which the different types of sensors (ie. `class Light_Sensor`, `class US_Sensor`, `class Slide_Sensor`, etc) can inherit. One possible implementation scenario would be as such:

```
class  Sensor
{
public:
    virtual int sensorReading() = 0;
};

class  Light_Sensor : public Sensor
{
public:
    virtual int sensorReading() override;
};

class  Slide_Sensor : public Sensor
{
public:
    virtual int sensorReading() override;
};

class  Robot { };

class  Create : public Robot, public Light_Sensor, public Slide_Sensor { };
```

The issue with this architecture becomes apparent when attempting to envision a realistic runtime scenario. To begin with, the double-inheritance of `Sensor` through `Light_Sensor` and `Slide_Sensor` will not achieve the desired outcome of a robot containing two sensors, but will rather contain some sort of overlapping amalgamation of the two derived classes, since inheritance does not create a new class instance but simply imports all of the data and methods from the parent class(es). Additionally, the function `sensorReading()` has two completely distinct overridden definitions in each child class of `Sensor`, but both of them are inherited, with no clear rule as to which one would be used at runtime by `Create`. While virtual inheritance can help alleviate the second issue, it still does not solve the first issue of single-instance due to that being part of the nature of inheritance.

The solution to this is to combine the useful aspects of polymorphic inheritane with a composition-based system. Instead of indiscriminately using inheritence, the choice

between composition and inheritance can be summed up as follows: "If an object *is* something, it inherits. If an object *has* something, it composes."

An example in implementation would take a form similar to the following:

```
class  Robot { };

class  Servo { };

class  Camera { };

class  Create : public Robot
{
public:
    Servo  servo_one, servo_two;
    Camera main_camera;
};
```

This architecture not only makes more logical sense, it is also far safer and easier to use as compared to the alternatives.

This way, the program can be constructed by representing a real-world object by an equivalent structure in a program that can be passed as a single argument to a function or stored as one standalone element in an array or data structure, for example.

2 Concurrent Programming

2.1 Intro

The concept of concurrent programming is one that has been clearly defined at least since the 1960s, and yet continues to play a major role in computing to this day, especially in fields such as systems programming.

Essentially, concurrent programming involves the use of multiple *threads* of execution in a single program (concurrent programming can also be achieved using multiple *processes*, however this is an entirely different study and does not largely pertain to Botball). These threads allow for the simultaneous execution of commands, in contrast to the standard mode of execution where a computer executes commands one by one.

Due to their nature in simultaneously performing operations, there are of course a number of issues that can arise with incorrectly implemented multithreaded code, making it one of the more difficult aspects of programming. However, effective implementation can create a formidable program that is capable at performing at a rate of efficiency orders of magnitude above its competition.

2.2 Motivation

The choice to implement a multithreaded environment is one that usually arises out of necessessity rather than desire. Threads are fairly complex, can be computationally

expensive, require careful awareness of synchronization and memory access, can leak memory if improperly handled, and are notoriously difficult to debug.

However, in the context of Botball, the need for multithreading arises due to the need to accommodate both complex engineering and programming architecture. Take for example the following code, which will move two servos to their required positions:

```
static const int  PORT_A = 0, PORT_B = 1;
    enable_servos();
set_servo_position(PORT_A, 1800);
set_servo_position(PORT_B, 1500);
disable_servos();
```

This is an example of a scenario where multithreading is required to accommodate a physical reality of the robot's construction, where it needs to move two servos at the same time, but is presently unable to do so, since the computer will execute the first call of `set_servo_position()`, and only once that function returns will it execute the second call. In a hypothetical scenario, a robot may need to execute a continuous scissor lift to create a vertical motion. The solution, of course, is multithreading:

```
#include   <thread>
static const int  PORT_A = 0, PORT_B = 1;
enable_servos();
std::thread servo_a_thread(set_servo_position, PORT_A, 1800);
std::thread servo_b_thread(set_servo_position, PORT_B, 1500);
servo_a_thread.join();
servo_b_thread.join();
disable_servos();
```

The code above creates two threads of execution that run the `set_servo_position()` function for each servo, with the second and third constructor arguments to `std::thread` being the arguments that the thread function should take. The threads are then instructed to wait for termination and then join using the `std::thread::join()` function.

This is the basic structure of thread handling in C++. The syntax for calling member functions of a class, however, is slightly different:

```
class  Object
{
public:
    void   function(int arg) { //... }
};

int main(int argc, char** argv)
{
    Object instance;
    int argument;
    std::thread Thread(&Object::function, &instance, argument);
    Thread.join();
}
```

Attempting to create a thread from within `Object` would of course mean that "`&instance`" would be replaced by "`this`".

2.3 Implementation

While a rudimentary implementation scenario has been covered above, multithreading becomes more involved in the context of a pre-existing object-oriented structure. Consider, for example, the following code for moving a robot's wheels and two servos at the same time.

```
#include   <thread>
static const int  PORT_A = 0, PORT_B = 1;
static const int  MOTOR_LEFT = 0, MOTOR_RIGHT = 1;
enable_servos();
std::thread servo_a_thread(set_servo_position, PORT_A, 1800);
std::thread servo_b_thread(set_servo_position, PORT_B, 1500);
mtp(MOTOR_LEFT, 1500, 700);
mtp(MOTOR_RIGHT, 1500, 700);
bmd(MOTOR_LEFT);
bmd(MOTOR_RIGHT);
servo_a_thread.join();
servo_b_thread.join();
disable_servos();
```

Purely in terms of functionality, the code above performs its task perfectly! It executes all three motions simultaneously and effectively. However, it fails miserably in terms of readability, maintainability, and reusability, rendering it near useless in the broader context of the project.

For one thing, the variables are only vaguely defined, and are all declared within `main()`. The movement code performs a low-level movement-and-blocking sequence of each motor *individually*, requiring four times as many lines as it should to perform one single task. Additionally, redundant/repeated arguments are passed each time, creating significantly more room for error.

The code is very difficult to read and understand at a glance, since there is no indication of function beyond variable names. `mtp()` and `bmd()` are vaguely indicative of their function, however one must individually read each argument to the functions in order to decipher that the robot is moving in a straight line.

Attempting to repeat a similar, but slightly modified sequence where all the servo positions and movement distances are different entails copy-paste modification of at least 6 numerical arguments, perhaps more, making the code very difficult to reuse.

The solution, of course, is to apply the principles and practices of object-oriented programming detailed earlier. The robot and servos ought to be represented by objects, where the robot would contain the two servos. The question now, of course, is as to how the multithreading would be handled in the context of an object-oriented architecture.

Now, we will define a `class Servo` that handles its own thread spawning and joining from within the object, as well as giving the user an option to run a command sequentially

or simultaneously.

```
#include   <thread>
#include   <new>

class  Servo
{
protected:
    int const  PORT;
    std::thread*   movement_thread = nullptr;
    virtual void   move_base(int distance, int time) { //... }
public:
    Servo(int port) : PORT(port) { }
    virtual ~servo() { }
    void   join_movement_thread()
    {
        if (movement_thread)
        {
            if(movement_thread->joinable()) movement_thread->join();
            delete movement_thread; movement_thread = nullptr;
        }
    }
    virtual void   move(int distance, int time, bool threaded)
    {
        movement_thread = new std::thread(&Servo::move_base, this,
            distance, time);
        if (!threaded) join_movement_thread();
    }
};

int main(int argc, char** argv)
{
    Servo  servo;

    // Concurrent movement
    servo.move(400, 100, true);
    servo.join_movement_thread();

    // Sequential movement
    servo.move(400, 100, false);
}
```

With this implementation, a `Servo` instance can be declared inside the `Create` class, for example, and then the user can decide when programming the actual sequence whether they want the thread to run in parallel or whether they want to wait for it to finish executing before moving onto the next command.

Now consider the case of a robot that has already used all the maximum allowable number of servos, and so now needs to use a motor as a servo. The functions `mtp()`, `mrp()`, and `bmd()` serve well towards this purpose as they move the motor based on

distance, however motors have a significant difference from servos in that they are not locked in place, and will drift out of position if force is applied to them. The remedy to this is to have some sort of constant check running "in the background" to determine if the motor is drifting and correct any drift accordingly.

While the function of this code may not be immediately apparent at a glance, it essentially inherits from the previous `Servo` class and adds the needed aspects of motor functionality. `move_base()` is overrided to suit the implementation needed for a motor as opposed to a servo. The thread creation functions identically to the base servo class in that it spawns a new thread when `move()` is called, which can later be joined with `join_movement_thread()` if `threaded` is set to `true`, or will automatically join if it is set to `false`.

The main difference comes with the addition of `drift_thread` and its associated functions, as well as the three `std::atomic<>` variables: `current_position`, `done`, and `moving`. Essentially, the constructor of `Servo_Motor` spawns the thread `drift_thread` and makes it run the function `check_drift()`. In order to check the drift, there are two conditions that must be fulfilled: we must have previous knowledge of the correct position that the motor has drifted from, and the robot should not attempt to correct motor drift while it is deliberately moving the motor. The variable `current_position` addresses the first concern; any calls to `move()` set `current_position` accordingly, meaning that it will only be updated by deliberate, intentional movements and serves as a point of reference against which a drifting motor can be compared. However, a normal `int` would not suffice since the variable is being modified by `movement_thread` while being read by `drift_thread`. The simultaneous access of one variable or location in memory by multiple threads can lead to what is known as a "race condition", where the appropriate timing for access to the variable is not controlled, resulting in poorly defined behavior.

The solution here is the use of `std::atomic<>`, a standard library template for atomic variables. Atomic variables solve this issue by internally controlling access to their memory and allowing for the same variable to be accessed by two threads.

The variable `moving` is set to `true` when the function `move()` is called, then set to `false` when it terminates. This way, the motor will not attempt to correct drift while it is being commanded to move. Additionally, drift only needs to be corrected while an instance of `Servo_Motor` is in scope, therefore `done` remains as `false` until the destruction of `Servo_Motor`, where it is set to `true` and `check_drift()` is able to terminate.

While `check_drift()` is running within `drift_thread`, it performs a simple comparison check to see whether the motor has drifted significantly from `current_position`, and adjusts it accordingly. This way, the motor is able to correct its drift throughout the duration of the program without holding up the program or taking any additional time, and without hindering the normal movement of the motor.

This is just one example of what can be done with object-oriented concurrent programming, and how it can be done. Opening the door for well-structured, maintainable code that can run multiple tasks simultaneously is an incredibly powerful tool in Botball and in most areas of robotics in general.