Good Coding Style
Wesley Myers
wmyers@andrew.cmu.edu

Good Coding Style

Introduction

Over the many years that I've been programming, it has come to my attention that good coding style is very important. A lot of problems that programmers encounter when learning to program are due to simple coding errors. When there is more than one programmer modifying code, which is encountered in Botball and other software projects, even more problems can arise. This paper discusses several ways to help reduce and even fix these errors.

Assignment Mishaps

Sometimes the simplest programming mistake can result in lengthy debugging sessions, even though the fix involves adding or changing a single character. This section illustrates one such error that can be easily avoided by adhering to a simple guideline.

A simple mistake that we have all made at some point is shown in Figure A.

```
if(var = 1) {
      do_something();
}
```

Figure A.

```
if(1 == var){
      do_something();
}
```

Figure B.

Sometimes, the do_someting() function would execute as intended and no one would know the difference. In other cases, do_something() would unexpectedly execute. What is obviously wrong with this code? The "if statement" will assign 1 to var and then go into the block and then do something. This is a simple error that can cause hours debugging just to find it.

Clearly the programmer intended to type "var == 1", however he/she mistyped and forgot to enter the second "=." A suggested programming style to prevent this is represented in Figure B. This code is what the programmer intended. Notice that the constant appears first. If the programmer makes the same error by typing a single equals sign, the compiler will throw an error saying that the statement is trying to assign a variable to a constant, which is not allowed. This simple change in style can save time, and it makes your code safer.

Bracket Everything

One of Botball's great advantages is that it is a team programming challenge. A common error I have come across in school and in Botball are bracketing issues. In team programming, different programmers edit code constantly, thus eventually leading to

simple errors. So in this section, we'll talk about how to avoid a problem that can arise during group programming when it comes to bracketing.

It is simple just to enter the code below and get the desired results. Though this is simple and works, lets look at the group programming aspect of this.

```
if(1 == var)
      do_something();
```

An issue that plagues even the best of us is when someone, or even ourselves, *returns and adds* a line of code intended to be in the logical block. Sometimes the following happens:

```
if(1 == var)                    if(1 == var){
      do_something();                 do_something();
      do_something_again();           do_something_again();
                                }
```

Written Code Intended Code

What happens in the written code? This code has "do_something()" executing under the control of the if statement and, however "do_something_again()" will always run. To the untrained eye, the code to the left appears to be correct (maybe just because of the indenting!). If one were to add brackets in the original code, then the "do_something_again()" would've been properly placed within the brackets as intended. So my advice is to always block your logical statements, even if there is only a single statement. This style may save you hours of debugging over something very trivial.

Good Documentation

I cannot stress how important commenting code is. A Botball team cannot succeed without commenting their code. As I've already mentioned, Botball is very much a team oriented programming competition. Often, when one student has finished writing code, he or she does not always comment their code. If other students add to that code and fail to comment, confusion may result. Here we will talk about a simple guideline to help this out.

A problem that usually arises at this point is that the second student either knows what the code does, doesn't know, or assumes it does something. If the student understands what the code does, then all is good. However if the student does not know, then the student must spend valuable time figuring out what the code does. You can save so much time by just commenting! In my group programming projects at school, we make each other document what each of us add and change. Below is an example:

```
/**********************************
Added below code to better implement server data transfer
**********************************/
```

This block of code sticks out like a sore thumb when you change something minor. The above code example should only be used if you want to highlight a change that would

otherwise go unnoticed. For simple comments either just use "//" or the "/* code */" style.

There are also more ways to make good comments. At the head of a .c file, it is good practice to put a description of what is contained in the file and how it fits in with the overall larger project. Before each function should be a function header. The function header should contain a series of valuable information. It should include a description of the function, its arguments, what the function returns, error cases relevant to the caller, and any assumptions the function might make. Commenting large sections of code or even small-complicated sections can be quite helpful to the reader. [1]

Never assume you know what the code is doing. This only creates headaches! Always understand your code inside and out, even when programming in a team setting. When you assume you know what the code is doing, this can create unexpected and usually undesired results.

Good Variable Names

In Botball and even at Carnegie Mellon, I have helped people with code that had strange variable names. In this section, we'll talk about why this is bad.

Below is some example code from a student at CMU. He came to me asking what was wrong with his code. As I was going through his code I found this function below. The goal of this code was to convert a hex value or integer that is initially a string to an unsigned int.

```
unsigned int thingy(char *hehe) {
      unsigned int haha;

      if(strncmp("0x", hehe, 2) == 0) {
            sscanf(hehe, "0x%X", &haha);
            return haha;
      }
      else
            return atoi(hehe);
}
```

This code doesn't have anything wrong with it, as long as you use it properly. If you didn't recognize what this code was doing, would you immediately know? "Thingy," "hehe," and "haha" do not help explain what this code's intent was. It pains me even just to retype this. The point is: always use variable names that are descriptive [1]. This will help you and anyone else that is looking at your code understand what is going on. On the next page is the same code with the good coding style that I've talked about so far.

```
/*
convertX(char *)
This code takes in a hex or integer value in the form of a string
and returns an unsigned int
*/

unsigned int convertX(char *s)
{
      unsigned int value;

      /*look at the first two chars for "0x"*/
      if(0 == strncmp("0x", s, 2)){
            /*grab number and return*/
            sscanf(s, "0x%X", &value);
            return value;
      }
      else {
            /*just return atoi output*/
            return atoi(s);
      }
}
```

Modularity of Code

Often in robotics, we will tell a robot to do a series of motions. Often these motions are repeated multiple times given the task at hand. If we can generalize a series of repeated motions into a series of the same function, then we have succeeded in simplifying the code even more [1]. Take the below code example. Here the robot is driving forward by 10 inches and rotating 90 degrees to the right, essentially driving the path of a 10x10 square.

```
void main()
{
      drive_foward(10);
      rotate(90);
      drive_foward(10);
      rotate(90);
      drive_foward(10);
      rotate(90);
      drive_foward(10);
      rotate(90);
}
```

This can easily be simplified into this

```
void forward_and_rotate()
{
      drive_foward(10);
      rotate(90);
}
```

```
void main()
{
      forward_and_rotate();
      forward_and_rotate();
      forward_and_rotate();
      forward_and_rotate();
}
```

With this style of code, we've taken a rather ugly form of code and made it simpler. Notice how the code has become self-documenting. While this example is rather basic, imagine a much more complicated series of motions. In the first example, if one had made a typo for the motion commands, then one would have to fix four or eight numbers and only hope to get them all correct. In the second example, only one or two numbers would have to be changed to fix the problem. Even generalizing the function could help. What if the robot wanted to go straight at different distances and turn in different directions? This makes the code simpler and easier to read. Below is an example of turning this into a function.

```
void forward_and_rotate(int fd, int rot)
{
      drive_foward(fd);
      rotate(rot);
}
```

Summary

So in the course of this paper, I've discussed ways to improve one's programming skills by following a couple of guidelines. I've discussed simple assignment solutions, bracketing solutions, and even naming conventions. The best things to take away though are the commenting and debugging sections. These sections are where one will get their kudos later on in their programming careers. Being able to document one's code will help other programmers to easily maintain their code. The most valuable programmers are sometimes the one that can figure out problems with their debugging techniques and this effort is helped by a good coding style.

References

[1] Kesden, Greg. "Code Style." *15-213/18-243 Introduction to Computer Systems*. Carnegie Mellon University, 01 May 2011. Web. 13 Jun 2011. <http://www.cs.cmu.edu/~213/codeStyle.html>.