**Good Coding Habits**

**Part 2: Good Formatting**

**Foreword:**

Hello readers.

As before, this article is mainly aimed at beginners, but I'd recommend reading it even if you aren't a beginner because frankly everyone is capable of writing messy code and it's good to be reminded of that from time to time.

This time the article is going to be focusing more on form than contents. Having meaningful names is all well and good, but even the best variable names in the world would be useless if they are strewn across the page like a Jackson Pollock painting.

(Spoiler: I'm not a fan of Jackson Pollock paintings.)

And so begins part two...

**Introduction:**

This second part is about using good formatting. Once again this is something beginners sometimes struggle with, and equally something experienced programmers let slip due to laziness or time constraints.

Visual structure is important in all languages, not just programming languages. Without spaces, paragraphs and punctuation we would all be struggling to discern great walls of text (which admittedly Japan seems to manage by some miracle).

Programming languages don't just have symbols and braces for the sake of the language, they're also there as a tool for the programmer to keep the code nicely formatted and by extension keep the code readable.

So in this article I'm going to be discussing formatting in an attempt to save the world from messy code.

**Tip 1: Format Your Code Wisely**

Even with short functions it's possible to end up with code that looks cluttered. Sometimes people write code that ends up being quite squashed up and hard to read because they don't use blank lines, spaces or brackets to keep their code neat.

This tip comes with a few little 'rules of thumb' to help you do this:

* *Make sure operators have spaces around them.*
* *Break a line up into smaller logical chunks if it's too long.*
* *Use brackets to separate expressions when appropriate.*
* *Use well named variables to store intermediate results.*

I think this tip and its rules are best expressed by example, so here's some code:

bool intersects(Point point, Rectangle rectangle)

{

return point.x>=rectangle.x&&point.x<rectangle.x+rectangle.width&&point.y>=rectangle.y&&point.y<rectangle.y+rectangle.height;

}

It works perfectly and everything is reasonably well named, but it's still hard to read because it's being let down by the way it's formatted.

So to make it better, I'll apply the rule *“Make sure operators have spaces around them”*, which results in this:

bool intersects(Point point, Rectangle rectangle)

{

return point.x >= rectangle.x && point.x < rectangle.x + rectangle.width && point.y >= rectangle.y && point.y < rectangle.y + rectangle.height;

}

It's not brilliant, but it's a start – at least now it's clearer what operators are being used. If you squint you can probably make out a couple of comparison operators, some and-operators and some addition operators.

Next I'll apply the rule *“Break a line up into smaller logical chunks if it's too long”*, which results in this:

bool intersects(Point point, Rectangle rectangle)

{

return

point.x >= rectangle.x &&

point.x < rectangle.x + rectangle.width &&

point.y >= rectangle.y &&

point.y < rectangle.y + rectangle.height;

}

Now it's getting somewhere. From this it's now clear that the function is returning the result of a long conditional expression made up of four smaller conditions.

But it could still be better, so next I'll apply the rule *“Use brackets to separate expressions when appropriate.”*. Of course 'when appropriate' is open to interpretation and different people will have different opinions of when that is, but usually you'll be able to tell what you think looks better. In my case, I think I like the look of this arrangement best:

bool intersects(Point point, Rectangle rectangle)

{

return

(point.x >= rectangle.x) &&

(point.x < (rectangle.x + rectangle.width)) &&

(point.y >= rectangle.y) &&

(point.y < (rectangle.y + rectangle.height));

}

I could have decided I only wanted to put brackets around the addition parts, or I could have gone further and put in more brackets, but I think this is a suitable balance. Here there is no ambiguity as to what the order of the operators are and there aren't so many brackets that the code is overloaded with them.

Normally I would be perfectly happy with this arrangement, since I think it looks clear and easy to maintain. However more experienced programmers (like me) sometimes take for granted that code like this isn't as clear to beginners as it is to us, so we tend to settle for something halfway between minimalism and verbosity. Truly, most experienced programmers would be able to understand this

However, there is more that could be done to make this code more understandable, and I want to take it a few steps further so I'm going to apply another rule, *“Use well named variables to store intermediate results.”*.

bool intersects(Point point, Rectangle rectangle)

{

const int rectangleLeft = rectangle.x;

const int rectangleRight = rectangle.x + rectangle.width;

const int rectangleTop = rectangle.y;

const int rectangleBottom = rectangle.y + rectangle.height;

return

(point.x >= rectangleLeft) &&

(point.x < rectangleRight) &&

(point.y >= rectangleTop) &&

(point.y < rectangleBottom);

}

Adding meaningful variables can make code a lot easier to understand in some cases.

In this case, the variables now make the code's functionality a lot clearer. Before the change it might not have been as obvious, but now it's clear that rectangle.x was actually the left side of the rectangle and rectangle.x + rectangle.width was actually the right side of the rectangle.

So, I've applied all the rules, but actually, I think there's one more thing I'd like to change before I'm happy with this.

bool intersects(Point point, Rectangle rectangle)

{

const int rectangleLeft = rectangle.x;

const int rectangleRight = rectangle.x + rectangle.width;

const int rectangleTop = rectangle.y;

const int rectangleBottom = rectangle.y + rectangle.height;

return

((point.x >= rectangleLeft) && (point.x < rectangleRight)) &&

((point.y >= rectangleTop) && (point.y < rectangleBottom));

}

One of the rules is *“Group code logically.”*, and in this case I think using less lines and extra brackets actually makes the code more understandable. Here not only can you see what the comparisons are, but it makes the bigger picture clearer. The first part of the return statement is checking whether point.x is between the left and right sides of the rectangle, and the second part is checking whether point.y is between the top and bottom sides of the rectangle.

In fact, I think that's actually quite a good definition for what it means for a point to be intersecting a rectangle. When you write it clearly enough, code can actually be pretty similar to a description of what it does.

As I said before though, this is probably more descriptive than what you might find in production code (even very high quality production code). This is the other end of the scale, where great effort has been put into making the code extra readable. More advanced programmers probably won't write like this unless they're specifically writing code for educational purposes or want their code to be especially clear.

Good use of space can make the difference between code that's easy to read and code that's too messy to make sense of. Sometimes it's alright to break these rules, especially if there's a measurable difference in code size or speed, but if you do, please comment your code to explain what it does.

**Tip 2: Keep Formatting Consistent**

A lot of different people write their code with different formatting styles. This is a fact of programming and it won't be changing any time soon. In fact it's so notable that there's an entire Wikipedia article about different ways of placing braces and indents in code (according to which my preferred programming style is called 'Allman style' after Eric Allman).

I won't sit here and discuss the different brace styles since there are a lot of arguments for and against each of them, and frankly I'm biased anyway.

I will however give you a sample of four of the more popular brace styles:

// K & R style

// named after Brian Kernighan and Dennis Ritchie

// the authors of "The C Programming Language"

// Dennis Ritchie created C

int main(int argc, char \*argv[])

{

while (x == y) {

something();

somethingelse();

if (some\_error)

do\_correct();

else

continue\_as\_usual();

}

finalthing();

}

// "One true brace style" (origin unknown)

int main(int argc, char \*argv[])

{

while (x == y) {

something();

somethingelse();

if (some\_error) {

do\_correct();

} else {

continue\_as\_usual();

}

}

finalthing();

}

// Stroustrup

// named after Bjarne Stroustrup

// the author of "The C++ Programming Language"

// and the creator of C++

int main(int argc, char \*argv[])

{

while (x == y) {

something();

somethingelse();

if (some\_error) {

do\_correct();

}

else {

continue\_as\_usual();

}

}

finalthing();

}

// Allman style

// named after Eric Allman

// the creator of sendmail and syslog

int main(int argc, char \*argv[])

{

while (x == y)

{

something();

somethingelse();

if (some\_error)

do\_correct();

else

continue\_as\_usual();

}

finalthing();

}

There's nothing wrong with preferring a certain style, but it's important to be consistent when writing code – which means picking a style and sticking to it. You don't have to use the same style all the time; in fact, being able to switch between styles is a good skill to have (one I admit I struggle with). However it's important to keep individual projects in the same style.

**Summary**

Formatting isn't just aesthetic, it serves an important purpose. Good formatting can be the difference between readable code that is easy to maintain and code that is difficult to read and thus hard to make sense of.

Some of you are probably thinking “but nobody else reads my code”. That doesn't matter though, because even if it's only you reading your code it can still cause issues. If you stop working on a project for a while and then come back later only to find that you can't read or understand your code then you'll only have yourself to blame (and that's never a fun experience).

Worse yet, if your code isn't well formatted then you're not only making it hard to read, you're also making it hard to spot bugs. If you've got a stray increment floating around the page it could be the difference between spotting a bug immediately or sitting there scratching your head for three hours.

I liked having a quote to end on last time, so I went out and found another one (albeit a less famous one):

*“A developer who doesn't care about style is like an artist who doesn't care about colour.”* - Brian Ensink, Stack Overflow.