Deciphering Strange Arduino Code

Created by Phillip Burgess

```
if (wOffset == rOffset) { // Is an RGB-type strip, 3 bytes/pixel
  for (uint32_t i = 0; i < numBytes; i += 3) {
    p = &pixels[i]; // -> NeoPixel lib buffer (8-bit)

    // Blend values between p1 & p2 buffers (if blending is disabled,
    // p1 & p2 both point to the same data, so we don't need separate
    // code for blended vs not).

    c = *p1++ * weight1 + *p2++ * weight2; // 32-bit result
    // Determine base index into gamma table (high byte of 32-bit
    // result), and weighting of next gamma entry.
    idx = c >> 24; // High byte = base gamma table index
    w2 = c >> 16;  // Mid-byte = next-entry weight
    c = g16[0][idx] * (256 - w2) + g16[0][idx + 1] * w2;
    p[rOffset] = (c >> 16) + ((c & dither_mask) > d);
    // w2 (and its implied inverse) are gamma table weights. Their sum is
    // always 256, but w2 only goes up to 255, again on purpose and by
    // design. The weight of the second entry should be at most 255/256 --
    // if it were 256/256, we'd just +1 the base index and use 0 for w2;

    // Same operation, green channel
    c = *p1++ * weight1 + *p2++ * weight2;
    idx = c >> 24;
    w2 = c >> 16;
    c = g16[1][idx] * (256 - w2) + g16[1][idx + 1] * w2;
    p[gOffset] = (c >> 16) + ((c & dither_mask) > d);

    // Same operation, blue channel
    c = *p1++ * weight1 + *p2++ * weight2;
    idx = c >> 24;
    w2 = c >> 16;
```

https://learn.adafruit.com/deciphering-strange-arduino-code

Last updated on 2024-06-03 01:59:38 PM EDT

Table of Contents

Overview

```
uint8_t *in = pixels, *out = dmaBuf, i, abef, cdgh;
uint32_t expanded;
for (uint16_t p = numBytes; p--;) {
  cdgh = (*in++ * brightness) >> 8;
  abef = cdgh & 0b11001100; // ab00ef00
  cdgh &= 0b00110011;       // 00cd00gh
  expanded = ((abef * 0b1010000010100000) & 0b0100100000000010010000000) |
             ((cdgh * 0b0000101000001010) & 0b0000000010010000000010010) |
             0b1001001001001001001001001;
  *out++ = expanded >> 16;
  *out++ = expanded >> 8;
  *out++ = expanded;
```

As one works their way up from the Arduino “blink” sketch into increasingly sophisticated projects, part of that process entails more exposure to others’ code… open source projects, libraries that encapsulate particular functionality and so forth. Such code originates from folks of all backgrounds and skill levels, and you will invariably encounter some head-scratching moments. **Not necessarily bad code, but weird.**

This guide explains a few lesser-seen elements of C syntax. **“The Arduino language” is really C++, which is basically C with added flavor crystals.** C was developed back in the early 1970s (https://adafru.it/10sc) and has become one of the most widely-used programming languages. With such a long and diverse history, you’re bound to find some odd creatures under the floorboards.

There are a couple of reasons this might be of interest:

- To expand one’s own C (and C++, and thus Arduino) programming knowledge; **maybe there’s valuable tips here to use.**
- To write code that’s friendlier to newcomers; **maybe these are things to avoid.** If it confused you, it may confuse others, and that’s not ideal for teaching!

How Does This Happen?

There are a number of ways that strange code might manifest…

- **Familiarity** with a programming language can lead to **contractions,** just as with spoken languages. In English, “is not” and “can not” become “isn’t” and “can’t.” C and most other programming languages have some shorthands that aren’t always obvious to new learners.
- With C’s long history, some of today’s projects are written by folks with decades of experience…who may have habits ingrained when disk and screen real estate

were **highly constrained,** and who's natural "flow state" is to always **pack more code in fewer characters.**

- Again with long-time programmers...even into the late 1990s, one was fortunate to get a free (and rudimentary) C compiler with a $30,000 UNIX workstation. Commercial optimizing compilers could add thousands more. So there was a tendency to **attempt optimizations manually.** Some of the code in today's projects is carried over from posterity. Some is just written weird out of habit.
- Programmers have a wicked sense of humor. **Obfuscation**—intentionally writing to be difficult to understand—can be funny, and there's even contests for such things (https://adafru.it/10sd). **You don't want this in teachable code,** but occasionally one might drop a punchline for other veteran programmers. On rare occasion obfuscation is done with intent to be "31337" or to conceal trade secrets...but that's petty, please don't.
- **Maybe I wrote it.** If reading the source code to some Arduino library and it's like "What were they thinking?", check for my name on it. With the exception of malicious obfuscation, I'm guilty of all of these. Sorry about that.

This is not a comprehensive C programming guide; it only covers a few specific sticky topics that have raised questions in the Adafruit Forums (https://adafru.it/jlf). For well-rounded learning, there are many classic books on the subject, or just by following along with Arduino tutorials.

Nothing discussed here is broken or illegal. It's all part of the formal C lexicon, just less-used...like the English language words "brambles" or "lugubrious."

Funny "for" Loops

C provides several ways to iterate—repeat a sequence of steps. Chief among them are `while`, `do-while` and `for` loops. Arduino adds the `loop()` function for the main body of a program; canonically C does not provide that one on its own.

Background

A `while` loop resembles the following:

```
// Initialization can go here
while (condition) {
  // Body of loop
}
```

"`condition`" is a placeholder—this gets replaced with some true/false statement; perhaps comparing a variable against a number, or checking the return status of

another function. As long as the statement evaluates as true, then the code in the body of the loop is executed again and again.

Initializing one or more variables before the loop is common, but not required of all programs.

A `do-while` loop moves the test condition to the end:

```
// Initialization can go here
do {
  // Body of loop
} while (condition);
```

"`condition`"—the true/false statement—is evaluated after the body of the loop. So the loop always executes at least once.

Some logic just works better one way or the other.

A `for` loop is sort of a **contraction** of the `while` loop:

```
for (initialization; condition; operation) {
  // Body of loop
}
```

"`initialization`" is any code that sets up the loop...for example, it's very common to set up a counter variable here. 92% of `for` loops start with an "`i=0`" initialization (programmer humor, not a true statistic, but it's quite common).

"`condition`" is just like the `while` loop—a testable true/false statement, evaluated before each iteration of the loop. Very often tests a counter against some limit, e.g. `i<10`.

"`operation`" is a simple statement that's performed after each iteration of the loop body. Very often used to increment a counter, e.g. `i++`;

So three lines of code before now all fit into one. It's a nice shorthand.

FUN FACT: conditional statements can also test integers for non-zero-ness. A variable with value `0` is equivalent to `false`. `1`, `42`, `-303` or any other non-zero integers are `true`. Okay, maybe not fun, but it's a fact.

`for` Shenanigans

Not often seen, hence occasional confusion, **but any of the parts of a `for` loop are optional;** they can just be omitted when not needed, as long as the other remaining syntax (semicolons, etc.) is observed. These are all valid `for` loops:

```
for (; condition; operation) {
  // Body of loop
}

for (initialization; ; operation) {
  // Body of loop
}

for (initialization; condition; ) {
  // Body of loop
}

for (initialization; condition; operation);
// I ain't got no body
```

`while` loops can also omit the body and just end with a semicolon, but `do-while` loops always have the braces.

Recent C++ compilers allow declaring temporary variables right inside the initialization of a `for` loop. This wasn't always the case, so you don't see it in older code. The "scope" of such variables is limited to the body of the loop (i.e. can't be referenced after the loop).

One can also use a comma separator to pack more stuff into a line, e.g.

```
for (int a=0, b=1, c=2; a<10; a+=1, b+=2, c*=3) {
    Serial.printf("%d %d %d\n", a, b, c);
}
```

(Note: `Serial.printf()` is typically present only for 32-bit microcontrollers in Arduino. With Arduino UNO and other 8-bit devices, one must `Serial.print()` eac h value separately.)

Old Habits Die Hard

It's very common in C/C++/Arduino to structure an infinite loop (that is, repeats until power is cut) like the following:

```
while (1) { // Always true; repeats forever
  // Body of loop
}
```

Occasionally there's no body, if a program encounters an error and needs to "hang" because it can't safely continue. The line just ends with a semicolon.

In the introduction, aged programmers, poor-quality compilers and code I wrote were all mentioned. Combined with a `for` loop's aforementioned optional elements, this yields a valid but little-seen infinite loop syntax:

```
for (;;) {
  // Body of loop
}
```

I developed this habit because one particular compiler on one particular system decades ago would generate machine code that actually evaluates the truth of a `while(1)` statement every time, instead of seeing the obvious infinite loop and simply jumping without evaluating. The empty `for` was more efficient.

No compiler nowadays is that stupid, they will always do the right thing. But in the time and circumstances, it made programs a few cycles faster and a few bytes smaller.

I only stick with `for(;;)` because it looks cool and I get to tell Big Iron Stories...but it's preferable for teachable code to use `while(1)` because that's universally understood today as "infinite loop."

What the ?

It's fitting that the one thing that seems to generate the most C code questions is a single character: **?** — the question mark.

This is C's **ternary operator.** Or as I like to call it, the Altoids operator, as it packs a lot of flavor into a tiny package.

Consider the following code:

```
if (condition) {
    a = 42;
} else {
    a = 17;
}
```

The ternary operator provides a shorthand for this construct, packing a conditional test, if/else and assignment into a single line:

```
a = condition ? 42 : 17;
```

"`condition`," again, is a placeholder. This is not real code. An actual program would have a meaningful thing to test there.

If the condition evaluates as `true`, the first value (left of the colon) is used. If false, the second value (after colon) is used.

While this if/else/assign pattern doesn't come up a whole lot, it's used just enough that you can easily imagine how veteran programmers, who may have grown up on a single fixed 80x25 character screen, would love something so nutrient-dense.

The downside is that, on its own, it's kind of nonsensical. Other C keywords like `do`, `while`, `if` and `else` provide at least some vague indication of what they do. But `?` is like they'd used up every last character on the keyboard for something else and it came down to this. I guess you could say it's **a condition phrased like a question.**

It's not just for A/B assignment. You can wrap this up in a conditional statement (e.g. `while`), or put function calls on one or both sides of the colon:

```
while (value ? func1() : func2()) {
  // Body of loop
}
```

Much as I appreciate its density, it's only right to **avoid** inflicting this on those just starting out, that would be mean. You'll never see this in NeoPixel strandtest.

But to seasoned coders, it's expressing a complex idea in a small space. Deep down inside libraries I'll use it all over the place, but that's basically the code equivalent of at-home in a bathrobe.

And for projects aimed at intermediate coders? Love to put one or two in there just to make them ask questions.

---

goto

Many folks aren't even aware that C/C++/Arduino has a `goto` command. It is so reviled, tutorials often skip over this.

`goto` continues program execution at a different place within a function.

The reason it's considered so distasteful is that overdependence quickly leads to **spaghetti code**. Unlike the braces and conditionals that visually represent the structure and flow of code, `goto` throws that all out the window...it just goes places and is hard to follow.

The canonical use case for `goto` is breaking out of nested loops:

```
for (int a=0; a<10; a++) {
  for (int b=0; b<10; b++) {
    if (condition) goto linguini;
  }
}
linguini:
// Code continues here
```

The nested inner part of the loops will execute 100 times, unless some condition arises. Well okay, that's one way out of the situation.

Keep in mind that anything that can be written with `goto` can (maybe should) be written without. One could add conditional tests to the `a` and `b` loops...at the expense of a few cycles. One could put the nested loops inside their own function and use `return` instead of `goto`....at the expense of a few-bytes-larger program. Always some small expense.

Potentially, infrequently, `goto` could get you out of a bind, but other programmers will look upon you like the cat threw up on the rug. Extremely, extremely infrequently, I might use one, preceded by a whole apologetic paragraph in comments explaining the justification for its use. But usually I'll avoid it and go through the extra hoops just to steer away from the whole tired argument.

Peripheral Registers

Sometimes in an Arduino project, especially projects tied to hardware, you'll see a line accessing a variable that isn't even declared anywhere in the code.

```
TCCR5A = 42;
```

Like...who is `TCCR5A`? Nobody's declared it anywhere, yet the program compiles fine (for certain boards). Meanwhile, I miss one variable declaration in my code and the compiler throws absolute fits! What's going on?

A microcontroller doesn't just run programs, it also talks to hardware like buttons, LEDs, weather sensors and small displays. So alongside the CPU, which does run programs, there sit additional silicon circuits called **peripherals** to assist with those other tasks.

The CPU needs a way to exchange data with those peripherals...read a button, or turn on an LED. To do this, peripherals are **memory-mapped**. They appear to the CPU just

like part of main memory, like variables or arrays, but behind the scenes are actually connected to these peripheral functions and not RAM at all.

So, somewhere off in a header file that the Arduino compiler includes automatically, all of those peripheral registers are declared and assigned absolute addresses as if in memory. If you read from or write to those "variables," you're actually talking directly to the hardware!

`TCCR5A` in this case is a timer control register, specific to some AVR microcontrollers...the ATmega1280 and ATmega2560 seen on Arduino Mega boards. This register's used to set up timekeeping or PWM on certain pins.

And that's the thing...these registers are extremely device-specific; you won't find `TCCR5A` on other chips. There may be some overlap within a few related chips of a family, but by and large this does not work when moving code between unrelated devices. If you see Arduino code doing this, it's really getting into some low-level sorcery.

We can only scratch the surface of the topic here, just wanting to explain what these mystery variables are about. This guide about the TiCoServo library (https://adafru.it/10sC) explains a bit further. If you get deep into the hardware side of things, understanding peripherals and navigating chip datasheets becomes a vital skill.

---

stdint

Any C or Arduino tutorial quickly gets into integer types, where there's a tradeoff between the memory required for a variable, and the range of values it can hold:

```
char a;
int b;
long c;
```

But soon you might encounter code that looks like this, perhaps with no explanation:

```
int8_t a;
int16_t b;
int32_t c;
```

A funny thing about C is that those first three types—`char`, `int`, `long`—do not have a canonical size that's true among all situations! It's totally implementation-dependent...

For example, on the Arduino UNO and similar, `int` (and cousin `unsigned int`) are 16-bit values...two bytes in RAM...with a range from −32,768 to 32,767 (or 0 to 65,535 for the unsigned variant).

But switch to most any 32-bit microcontroller and now those `int`s are 32-bit values... four bytes in RAM, with a much larger range. Same type names, but different utilization. **Sometimes, code falls over due to this change.**

The **stdint.h** header file...included automatically by the Arduino compiler...declares a set of known-sized integer types which can be relied on.

`int8_t` is always, by definition an 8-bit signed value (−128 to +127), `uint8_t` is 8 bits unsigned (0 to 255). One byte in RAM.

`int16_t` is always, by definition an 16-bit signed value (−32,768 to +32,767), `uint16_t` is 16 bits unsigned (0 to 65,535). Two bytes in RAM.

`int32_t` is always, by definition a 32-bit signed value (−2,147,483,648 to +2,147,483,647), `uint32_t` is 32 bits unsigned (0 to 4,294,967,295). Four bytes in RAM.

If you get into programming outside the Arduino environment, these types are not automatically declared, and one must explicitly:

```
#include <stdint.h>
```

Octal Literals

Do programming for any length of time and you'll soon learn about different number bases, such as binary or hexadecimal. It's especially common with graphics or peripherals...these different notations relate numbers more naturally to the microcontroller's internal representation.

```
int a = 0b00101010; // Binary representation of 42
int b = 0x2A;       // Hexadecimal representation of 42
```

The topic is well covered in existing programming tutorials so **this guide won't go into detail** (here's Wikipedia's entry on radix (https://adafru.it/10sF)), but I must specifically warn about **octal** notation, which occasionally trips folks up...

Octal is base 8 notation...three bits per digit, or values from 0 to 7. Notice how the binary and hexadecimal examples above have `0b` or `0x` at the start? Octal just has a `0`.

What occasionally happens is that folks will "zero pad" regular decimal values to make them fill out space nicely in a table, not realizing that they're actually mixing in octal numbers with totally different values, or will use digits beyond the valid octal range (e.g. `089`), then can't figure why the program fails. This is strange in that the other representations don't mind being zero-padded...

```
int table1[] = {
  0b00000000, // Binary representation of 0
  0b00101010, // Binary rep. of 42 (harmless leading zeros!)
  0b11111111, // Binary rep. of 255
};

int table2[] = {
  0x0000, // Hexadecimal representation of 0
  0x002A, // Hex rep. of 42 (harmless leading zeros!)
  0x00FF, // Hex rep. of 255
};

int table3[] = {
  000, // Octal representation of 0
  042, // *** Octal rep. of 34! Danger! Danger! ***
  089, // *** Invalid octal value! Danger! ***
  255, // Decimal rep of 255
};
```

Use of octal notation is super rare nowadays, but the formal C specification requires its inclusion for compatibility. If you encounter code using this at all it's probably been ported from truly ancient platforms. Because...this might blow your mind...bytes weren't always 8 bits!

What blows my mind even more is that the very first stored-program computer ever—the Manchester Baby (https://adafru.it/Nb3)—had a "modern" 32-bit word length. Its immediate successor...and a majority of systems for the next two decades...took other directions that favored octal numbers before mainstreaming back to 32 bits in the late 1960s.

This fact has nothing to do with strange C code...C didn't even exist yet...I just think it's so neat!

Compare vs Assign

Something that will bite you in the hiney is C's reliance on single vs. doubled-up characters for certain related but not congruent functionality.

It's not if, but when you'll lose hours or even days hunting an "inexplicable" bug...

```
if (a = 42) {
  // Do the thing
}
```

You know this by now, but for posterior: a single equals (`=`) is an assignment, double equals (`==`) is a comparison. In vs. out. The code above is assigning where it should be comparing...and because the result is non-zero, always evaluates as `true`.

This happens all the time, and will happen forever.

Complexifying matters, and on theme with this guide's focus, seasoned coders sometimes economize on space by intentionally putting an assignment inside an `if` (or `while`, etc.) statement...

```
if ((value = func())) {
  // Do the thing
}
```

This calls `func()`, assigns the returned result to `value`, then does the thing if the result was non-zero. This is actually normal and valid C syntax, and at most it's optional but recommended to have double parenthesis (as above) as if to tell others, "I meant to do that."

Assignment sometimes gets paired up with another, different character (or two) to perform an operation on a variable in one compact step, for example...

```
x += 3;  // Add 3 to x, assign result back to x; equiv. to x = x + 3
x -= 3;  // Subtract 3 from x, assign back to x; equiv to x = x - 3
x *= 3;  // Multiply x by 3, assign result back to x; equiv x = x * 3
x /= 3;  // Divide x by 3, and so forth...
x %= 3;  // x modulo 3, "
x &= 3;  // Bitwise AND of x and 3, "
x |= 3;  // Bitwise OR of x and 3, "
x ^= 3;  // Bitwise XOR of x and 3, "
x <<= 3; // Shift x left 3 bits, assign result back to x
x >>= 3; // Shift x right 3 bits, assign result back to x
```

But a few of these, syntactically similar, are actually comparisons...

```
if (x != 3) // If x does not equal 3...
if (x <= 3) // If x is less than or equal to 3...
if (x >= 3) // If x is greater than or equal to 3...
```

Of these, the bitwise shifts vs greater/less-than comparisons are most likely to cause misunderstandings.

Logical vs Bitwise

Another single-vs-double character situation occurs between logical operators (evaluating the `true`/`false` condition of a statement) and bitwise operators (applying mathematical combinations to numbers).

```
if (flag1 || flag2) {        // This is a LOGICAL OR operator
  result = value1 | value2;  // This is a BITWISE OR operator
}
```

The compiler will rarely complain if you mix these up, because integers can be evaluated as `true` (non-zero) or `false` (zero).

`||` is the **logical OR** operator, true if the operand on either side is true.

`&&` is the **logical AND** operator, true only if operands on both sides are true.

`!` is a **logical NOT**, which negates the operand following it. `true` becomes `false` and vice versa. Notice this one's only a single character.

Bitwise, meanwhile...

`|`, `&`, `^` and `~` are bitwise **OR**, **AND**, **XOR** and **NOT** operators. `<<` and `>>` are bitwise **shift** operations, ensuring that we can't make a single, all-encompassing "one versus two characters" differentiation. We'll not go into detail of the operations themselves, already well covered elsewhere (https://adafru.it/10ta). As mentioned on the prior page, any of these can be paired up with an assignment (`=`) as a shorthand operation.

Bitwise shifts have the interesting property of dividing or multiplying integers by powers of two...

```
a = b << 1; // Multiply by 2
a = b << 2; // Multiply by 4
a = b << 3; // Multiply by 8, etc...
a = b >> 1; // Divide by 2
a = b >> 2; // Divide by 4
a = b >> 3; // Divide by 8, etc...
```

Old-school code often has a lot of bitwise shifts because multiply and divide were relatively costly operations at the time. But modern compilers spot this opportunity and perform such optimizations automatically if beneficial, even at “low” settings. So… if you need to divide a number by 8, just divide by 8, it reads better. In other situations bit-shifts are still very appropriate…in graphics for instance, conceptually it may be more readable to shift pixels left or right by a number of bits. All depends on the task and the idea you want to convey.

Pointers

Pointers are an integral part of the C language and this won’t go into any depth, but to mention a bit of strange syntax…

Any variable, array or function in C takes up some amount of memory. Code can request a pointer to the start of that memory (“address of”) with the ampersand (`&`) operator. And a variable that is itself a pointer to something else is declared with an asterisk (`*`) operator:

```
char a[100];  // 100 bytes
char *b = &amp;a; // b is a pointer to start of a[] array
```

That’s simple enough. Where it gets strange is that pointers-to-pointers exist:

```
char **c = &b;  // Pointer to b variable, which is itself a pointer
char ***d = &c; // Pointer to c variable, itself a pointer-to-pointer
```

It’s pointers all the way down! In practical use, I don’t recall ever seeing more than `**` pointers-to-pointers (also sometimes called a vector). It’s all a bit obscure, but perfectly valid…one might pass a pointer-to-pointer to a buffer to a library function that may then reallocate it, changing the pointed-to pointer’s value. Don’t be alarmed when you reach that.

Arrow Operator

Classes in C++ (and structures in both C and C++) are a way of grouping related variables into a single encapsulating thing. Classes can also contain functions in addition to variables…but let’s look at a struct first, because it’s simpler:

```
struct {
  int month;
  int day;
  int year;
} apple = { 4, 1, 1976 };
```

```
Serial.println(apple.year); // Prints "1976"
```

Written this way, memory for the structure called “apple” is allocated at compile time, when it’s being declared, because its contents are a known thing initialized from constant values (April 1st, 1976). To access variables within that structure, a period (“.”) is normally used.

Sometimes a program doesn’t know ahead of time what’s going into a structure or class, or whether it’s even going to be used at all. Such code might allocate the struct or class dynamically, using malloc() (for C) or new (for C++). Consider this code that creates two Adafruit_NeoPixel objects…one a static declaration, the other dynamic:

```
Adafruit_NeoPixel pixels1(42, 5, NEO_GRB);
Adafruit_NeoPixel *pixels2 = new Adafruit_NeoPixel(42, 6, NEO_GRB);
```

Most NeoPixel code is written like the first line. Everything about the NeoPixel strip is known ahead of time…how many pixels there are, what pin it’s connected to, and the color format used.

When it comes time to set a pixel’s color or update the strip, those functions are called with the object name, a period, and the function name:

```
pixels1.setPixelColor(0, 255, 0, 0); // Set first pixel red
pixels1.show();                      // Update strip
```

Trying to do the same with the “pixels2” object would generate errors though, because that object was allocated dynamically…it’s an object pointer. Such items require a slightly different syntax:

```
pixels2-&gt;setPixelColor(0, 0, 0, 255); // Set first pixel blue
pixels2-&gt;show();                      // Update strip
```

Since pixels2 is really a pointer to an object, not an object itself, the right arrow “points” to where the object was actually allocated.

Why would anyone do this? Well…in the NeoPixel case…suppose you’re producing a lot of something, in multiple configurations. A toy with 10 pixels than animate in one pattern, and a different toy with 20 pixels in another pattern…but you don’t know ahead of time how many of each you’ll be producing. With the static declaration, if you program the wrong number of boards for each design, some will require re-programming with the correct code. With the dynamic declaration…there might be a switch or jumper that’s set during assembly to select which toy this is going inside…

and every board can then be programmed with the same code (which checks the jumper and adapts accordingly), no do-overs needed.

Pre/Post Increment/Decrement

Again, little to add here because pre- and post- increment and decrement are simply features of the language. But there's a funny thing you'll sometimes see in old-school code.

The rules of these operations are pretty straightforward...

```
a = ++b; // Add 1 to value of b, assign back to b, also assign to a
a = --b; // Subtract 1 from value of b, assign back to b, also to a
a = b++; // Assign value of b to a, then add 1 and assign back to b
a = b--; // Assign value of b to a, subtract 1 and assign back to b
```

You'll see these with pointers and counters a lot, often with the "`a=`" part left off, just performing operations on a single variable. Often the third element of a `for` loop, e.g. `i++`.

Python doesn't have these operators, preferring `a += 1` or `a -= 1` instead, so they're a little strange if just coming into C from that language.

Most modern CPUs support all of these operations down at the instruction level; you often get any of these increments or decrements "free" as part of another operation. That wasn't always true though. If I recall correctly, I think the Motorola 68000 (maybe others) only had pre-decrement and post-increment, and the other combinations required extra cycles as they're performed "manually." That chip's DBRA opcode (decrement and branch if not zero) was particularly efficient for loops.

So...in old code, or even in modern code written by folks with habits ingrained in that era...you may see programmers prostrating themselves to structure program flow into a `do-while` loop (decrementing and testing against zero at the end)...trying to optimize the code manually, because compilers sucked...rather than a simpler, more readable (and likely just as efficient now) implementation.

I still do this all the time, but mostly for nostalgia's sake.

Spaces, Tabs and Braces

> "Never attempt to teach a pig to sing; it wastes your time and annoys the pig."

– Robert Heinlein

This is **not actually about strange code,** just some productive advice...

As you interact with more of other people's code, you'll find everyone has a distinct style. Not just in the flow of logic, or the names of functions and variables, but simply details of layout.

As long as you always pair up the curly braces and end every statement with a semicolon, C is otherwise nonjudgmental about what goes where. This contrasts with others like Python, which can be strict about consistent indentation, and also has a whole culture of favored coding practices.

These two versions of a C function are roughly equivalent:

```
void do_if_odd(uint16_t x) { if(x & 1) do_the_thing(); }

void do_if_odd(unsigned short x)
{
    if (x % 2)
    {
        do_the_thing();
    }
}
```

One is packed tightly like a suitcase, the other spread out like an Apple store. One uses a bitwise AND to check for oddness, the other a modulo operator...but same result. The only functional difference is an obscure one: there might be rare platforms where `uint16_t` and `unsigned short` are different sizes. "Readable" is largely in the eye of the beholder; some will grasp program flow better when more is visible on screen, but generally the airy code will be kinder on newcomers. There are countless middle-grounds as well.

C Programmers sometimes get into heated debates about The Only Good And Proper Way To Structure Code™. Should indentation be performed with spaces or with tabs? Should the curly braces starting code blocks appear on their own line, or with the prior statement? Is goto in fact irredeemably evil?

Everyone has preferences. That's healthy and fine! I get such a thrill when lining up comments in adjacent lines of code. But these are subjective things, you will always find dissenters.

If you're **contributing to an open source project**, or are part of a **team:** the **winningest strategy I've observed of top programmers**—or maybe they're just extremely jaded programmers—is **don't go there.** The correct answer is: there is no One Answer™.

Instead, within the scope of a project, **adopt the conventions of the project lead, or a mutually agreed-upon style guide.**

Stay flexible. Do what you enjoy in your own code, but in collaborative code, the most prolific are doing the work, not arguing over spaces or curly braces, or GIF vs JIF, or pineapple on pizza or tails on werewolves. Take the group conventions to heart and then do the work. Fix that bug. Add that feature. Stop reformatting code like you're rearranging deck chairs and **Do. The. Work.**

In Adafruit libraries we've adopted the practices (https://adafru.it/10uF) of using **clang-format** for consistently formatting C/C++ code (with default settings, nothing special), and **doxygen** for API documentation. If new code is written close to those conventions, it's minimally disruptive to then filter it through those tools.

I'm quite certain that not a single person here 100% agrees with these tools' formatting conventions...but it's always at least adequate, consistent, and avoids religious debates. No time wasted, no pigs annoyed!