Painful Expressions

by Eric B. Wertz for ME106 Fall 2011

I've noticed over the past few semesters that students are having difficulty with expressions in C, particularly logical expressions, whether they're arithmetic or bit-wise expressions. Specifically, there are three common mistakes that I've noticed students make, so I'm going to try to kill them all with this one bullet.

Logical expressions are those expressions that are (typically only) used in the test condition-test part of `if`-statements, and in `for` and `while` loops. It's the guy that determines if you execute the code in the `if`-statement, or the `else`-statement if your `if`-statement has one. It's the same guy that determines whether you keep (or sometimes even start) executing the body of the loop.

The conventional thinking is that "if the expression/condition is 'true', then the program executes whatever's in the block of code that follows". The problem is that what's "true" is somewhat tricky, and not always obvious. The short answer is that any expression, whether it's a mathematical expression OR a logical one, that evaluates to a non-zero value is "true", and only zero is "false". While it's technically more correct to ask yourself if the tested condition in an `if/for/while` is zero or non-zero, you can still think about them as being "true" or "false" as long as you really, really understand that true and false are non-zero and zero.

The bottom line is that every expression in C results in a value, which might be as simple as any of the following:

- a constant, `5`
- a variable (scalar, floating-point, or pointer), `n`
- a arithmetic operation, `n+1`
- a bit-wise operation, `switches & 0xC0`
- a function call value, `getValue()`
- an arithmetic comparison operation, `n > 100`
- a bit-wise arithmetic operation, `button1_pressed && button2_pressed`
- an assignment operation, `a=5`

The "trick" is understanding what all of these expressions evaluate to, and there are two that are used incorrectly often enough that they have to be highlighted.

The first problematic type of expression is a seemingly useless one (to beginners[1]), and is particularly nasty because it's sometimes used intentionally. It's the assignment expression, the last one in the list above. As it turns out, assignments have a value, and the expression's value is the value being assigned – for example, the value of the expression (n = 0) is 0, and (n = 3) is 3. Normally one wouldn't think or care about why an assignment expression has a value, because you (pretty much always) only care about the assignment being done. The problem is that because (n = 0) [assignment] and (n == 0) [equality] look so much alike (especially to beginners), it shows up often enough to be an

1 There are two useful applications for assignments being expressions. First, it's somewhat useful to be able to be write statements like `a = b = c = 0;`, and the second reason is for combination function-call-and-returnvalue-test constructs like: if ((f=fopen("myfile.txt")) == NULL) { printf("Cannot open file!"); exit(1); }. There are other uses of the assignment-as-expression feature, but these or their variants are the two most common ones you may find in the wild.

issue.

The second non-obvious expression values are those of mathematical comparison (greater-than, less-than and/or equals) and logical (and, or, not) expressions, and both groups of these behave similarly. The rule is that if the evaluated expression is "true", the expression's value is 1, and if the expression evaluates to false, it's 0. These values follow the same rule as was described above – zero is false, and any non-zero (here specifically, 1) means true. Therefore, every expression involving one of these types of operators, <, >, ==, `!`, `&&` and `||`, all ultimately evaluate to 0 or 1.

Assuming that n has the value 3 at this point, all of the following expressions evaluate to non-zero, therefore "true".

```
n                       // 3
(n-2)                   // 1
(n > 0)                 // 1
(n < 10)                // 1
(n != 0)                // 1
((n>0) && ((n-1)>0))    // (1 && 1), therefore 1
((n == 3) || (n == 4)   // (1 || 0), therefore 1
(n | 3)                 // (3 | 3) is (0b11 | 0b11), therfore 9b11, or 3
(n & 2)                 // (3 & 2) is (0b11 & 0b10), therefore 0b10, or 2
(n=3)                   // 3
(n==3)                  // 1
```

And the following are all "false", meaning what..? Yes, zero.

```
(n-3)                   // 0
(n > 5)                 // 0
(n == 2)                // 0
(n != 3)                // 0
((n == 0) || (n == 1)) // (0 || 0), therefore 0
(n & 4)                 // (3 & 4) is (0b011 & 0b100), therefore 0b0, or 0
n==2                    // 0
n=0                     // 0
!n                      // !3 also meaning "not true", therefore 0
```

This background information is enough to fully understand three of the more common errors that pretty much every student makes in the first few weeks of the course. In fact, many students continue to struggle with these through the introduction to bit-wise operations (which you'll get to in the next week or so) because expression evaluation and bit-wise operations are critical in embedded programming.

First, the easiest beginner error to make and the hardest one to see no matter how long you stare at it – equals (=) vs. equals-equals (==). As explained above, because assignment (single-equals) is an expression, it compiles just fine as a conditional-test in an if-statement, so it's not a syntax error, but often a usage error. For example, compare the correct and incorrect versions below:

// Broken example
 	
int counter = 0;
 	
...code counts something going on...
	// Correct example
	
	int counter = 0;
	
	...code counts something going on...

`if (counter = 0) {`
`  printf("Nothing counted !\n");`
`}`
`else {`
`  printf("Counted.\n", counter);`
`}`
	`if (counter == 0) {`
	`  printf("Nothing counted !\n");`
	`}`
	`else {`
	`  printf("Counted.\n", counter);`
	`}`

Knowing what we know now about what an assignment expression evaluates to (the value being assigned), the value of the expression `(counter = 0)`, is 0. The test expression will then always be interpreted as false, and the remaining code will interpret this as nothing having been counted, no matter how successful your event-counting code was, or the hardware you built to count was. The other side effect is that after the expression `(counter = 0)` is executed, the (correct value of) counter will be reset to 0.

Second is the incorrect assumption that an expression that we've grown-up with since algebra class, which compiles, does do what you expect:

```
// broken range-checking example
int temperature = get_body_temp();

if (98 <= temperature <= 99) {
  printf("Lookin' good!\n");
}
else if (97 <= temperature <=  100) {
  printf("You ok, man?\n");
}
else {
  printf("Dude, you be illin' - get away from me!\n");
}
```

The problem is one of not appreciating that most operators (`+, -, *, /, ==, <, <=, >=, &&, ||`, etc.) only work on two values at a time, and the logical operators are ***not*** commutative like addition and multiplication are. Like every sequence of operations, there is a defined order that complex expressions get evaluated in unless you force them into a specific order (which we *highly* recommend you do!) with parenthesis. In this case, the above expression above is evaluated, left-to-right, as:

```
    ((98 <= temperature) <= 99)
```

Knowing what you know now about how these types of (sub-)expressions are evaluated, hopefully it's clear that nothing good is going to happen here. If it's not clear, work it through one step at a time and see how it goes. When testing expressions like this, ensure that you try values outside, inside and at the edges of the range being compared against.

Third is an incorrect construction that looks like it concisely specifies an intention, but doesn't do what you want either, as written:

```
#define PIN_BUTTON0 12
#define PIN_BUTTON1  8
```

```
// test for either button0 or button1 being pressed
if (digitalRead(PIN_BUTTON0 || PIN_BUTTON1) == LOW) {  // WRONG!
  digitalWrite(PIN_LED0, HIGH);
}
```

Once again, knowing what you now know, what's the equivalent of the if-statement test, once the test expression has been evaluated? Without telling you what it is checking for, let's suffice it to say that the following is the correct construction for such test:

```
// test for either button0 or button1 being pressed (CORRECT)
if ((digitalRead(PIN_BUTTON0) == LOW) ||
    (digitalRead(PIN_BUTTON1) == LOW)) {
  digitalWrite(PIN_LED0, HIGH);
}
```

Hopefully this focus on expression evaluation will cause you to think about, and more fully understand, how you form correct condition-test expressions in your code.