Overloaded with operator overloading

Ben Klemens

29 April 2009

Last time I discussed some pleasant uses of integer division, but I think most of us really think of it as an annoyance. We don't expect all the decimals to be truncated. If I type in `3/2`, I expect `1.5`, darn it, not `1`.

Indeed, this is an annoying gotcha to C and other integer-arithmetic languages, and more broadly, it shows us the dangers of *operator overloading*. O.o. is when an operator, like `/`, does something different depending on the types involved. For two integer types, the slash does the divide-and-truncate operation, and for anything else it does the usual division.

Oh, you can do pointer arithmetic, wherein you add a pointer and an integer. This too can lead to confusion: given `int x=3`, if you should mistakenly ask for `&x+3`, you don't get a compilation error, and the system just steps forward three steps as requested (which may or may not segfault).

Other languages actually *encourage* o.o., and give you tools to create a different meaning for `/` for any given pair of types. Well, you just saw the tradeoff: we can do things and create meaning for something that had been incoherent (like a pointer plus an integer), but if our expectations are wrong, then we have that much less keeping us from doing the wrong thing.

Human language is very redundant, which is a good thing. Redundancy is a good thing because it allows error-checking. When Nina Simone says *ne me quitte pas*, it's OK if you space out at the beginning, because … *me quitte pas* has the *pas* to indicate negation. It's OK if you space out at the end, because *ne me quitte* … has the *ne* to indicate negation.

Programming languages don't do this. We express negation exactly once, typically with only one character (!), and don't worry about things like case and gender. So if you space out in the first half of writing a line of code, there's nothing to call you on errors.

I'm not talking about sex enough on this blog, so here are some words for genitalia. The Spanish for penis is *pene*, masculine; the feminine equivalent *vagina*, is gramatically feminine. But as you can imagine, there are vulgar forms for when these terms sound too medicinal: the masculine *pene* becomes *polla* (f), and the girl-parts become the masculine (and very vulgar) *coño* (m). I could think of no better demonstration of how little gender in grammar has to do with actual gender. Instead, just think of them as noun classes.

So it's not about boys versus girls, but about redundancy, and giving the listener a few more clues about what the person across the room is trying to get across.

Programming languages *do* have genders, except they're called types. Generally, your verbs and your nouns need to agree in type (as in Russian, Amharic, Arabic, Hebrew, among other languages). That means redundancy, and perhaps a different verb form for the same action when executed on different types. With this redundancy, you'd need `matrix_multiply(a, b)` when you have two matrices, and `complex_multiply(a, b)` when you have two complex numbers (however expressed).

With operator overloading, of course, you don't need any of that. Express matrix multiplication as `a * b` and complex multiplication as `a * b`. This is much more brief, but you've lost redundancy.

I've said it above, but let me say it again: redundancy is a good thing. It'd be hard to confuse a complex scalar with a real matrix, but it's darn common to confuse a pointer-to-int and an int, or take a one-dimensional matrix to be a vector. As you add types, it only gets worse, and some systems will give you a list, vector, and unordered list to confuse, and the power to multiply together any two of them with `a * b`.

From here on to more complex types, there are a lot of subtleties involved. SQL, the language for manipulating database tables, is based on an algebra, meaning that there is an operation that maps to addition, an operation that maps to multiplication, a distributive property, et cetera. What if SQL were expressed as such, so you would write joins as `t1 * t2` instead of the verbose `select ... where t1.x = t2.x` form we do use? Things would be a lot more brief, but not necessarily any easier to read, write, or understand, because the `*` operator doesn't give you any information about what a product means in this context on these types. You just have to have the documentation open or have memorized the rules. The typical form for the rule is something like, 'It's sort of like multiplying scalars, but for the following additional rules and caveats....'

So, once more, redundancy is good, because the metaphor between the product in the real scalar context and the product in the context of the new type is probably only partially correct.

So there's the tradeoff: you've saved space on the page, and didn't have to type much of anything, but have lost all redundant hints that `b` is actually a list and not the vector you thought it was.

From here, the final decision is entirely subjective. I am a klutz and often commit the sort of errors I describe above, so I benefit heavily from a redundant language. You may be working primarily with only a few types that are hard to confuse, in which case all of my warnings are not an issue, and you only benefit from the brevity. But from the frequency of kvetching about how `int / int` behaves differently from `float / float`, it seems a lot of people lean toward preferring redundancy.

This is the only real example of o.o. I can think of in C. The `*` gets reused as binary multiplication and unary pointer-dereference, but those are very different actions and there's never confusion.