Fuck Operator Overloading

An essay by Blue-Maned_Hawk

Preface

This essay was largely written as a response to the ISO/IEC JTC1/SC22/WG14 paper N3051 by jacob navia. I do not know this person, and this essay is not intended as a personal attack against them. It is only intended as a criticism of the ideas presented in the paper.

Introduction

A paper has recently been proposed to Working Group 14 of Sub-Committee 22 of Joint Technical Committee 1 between the International Standardization Organization and the International Electrotechnical Commision, sent in by one jacob navia, titled "n3051 Operator overloading in C", which proposes the addition of the feature it describes. It includes the motivation for the paper, the rules for how the feature would work, a method for overriding overloads, the syntax for such overloading, and gives several examples.

This paper is unlikely to be incorporated. It's by a new contributor, and the proposed usages for this new system would be some extreme changes to the C language that that compatibility-break–fearing WG14 would likely be strongly against. In fairness, the paper itself does state that

> *This document is designed as a strictly "start of the discussion" framework. The ideas here could be off, many things probably are wrong. Nevertheless, it is a start.*

I don't intend to be hyper-nitpicky about the mistakes in this paper, like the pigeonholing name mangling scheme or the fact that `operator` would *definitely* be a new keyword. These superficialities do not matter to me because the concept of operator overloading itself is, in my eyes, flawed.

Operator overloading is unnecessary

The paper lists five motivations for operator overloading:

- Reducing the number of numeric types
- Allowing for counted strings and arrays
- Allowing for fat pointers
- Allowing new operations
- Hey, everyone else is doing it

We'll ignore that last one, because i think it's mostly there to satisfy WG14's prior art requirement, and i really don't have anything to say about it.

Here's the thing: *every one of these can already be done without operator overloading.* Really! Counted strings would probably be the most difficult simply because null-terminated strings are baked into the language, but it wouldn't be

impossible to do. Reducing the number of numeric types is unlikely to happen, but if it did, and decimal types and complex types were put into the standard library instead, then while operator overloading would make things a bit nicer, the fact that it wouldn't be possible to define new operators (for example, to get the conjugate of a complex number or construct a new decimal number without casting loss) means that some stuff would still need to be functions. Fat pointers can already be done in C by means of structure types; the fact that the C standard library doesn't do them is simply a mishap of historical mistakes.

The penultimate one in the list is the weirdest to me. After all, if we're restricted to the existing operators, what could it possibly mean for there to be new operations? The paper says that the `<stdckd.h>` header could be obsoleted by means of "a simple `#define`", and i can't make heads or tails of what that's supposed to mean. Defining the type to something checked would cause it to be implemented across the entire codebase instead of in just one place…and, well, i can't think of anything else that could be defined to anything.

Overall, operator overloading would be naught but syntactical sugar. Such sugar can play a very important role in making something more understandable, but when it's as restricted as it is here, i can't see it as being very useful. Furthermore, putting this power in the hands of the programmers means that it can be abused to make confusing messes, which brings me on to my next point…

Operator overloading is confusing

Suppose you come across this fragment of code:

```
…
score = bonus + score + extra;
if (score == next_level_requirement)
      unlock_next_level = true;
…
```

When operator overloading isn't a possibility, you can be pretty confident about what this does. You know that all of the identifiers in this *must* be numeric types, and there's only a limited number of ways that this could work. You also know some properties of the code—for example, you know that you could rewrite the first line as `score = score + bonus + extra;`, and it will do *exactly* the same thing while being slightly less confusing.

However, when operator overloading is a thing, you can't memorize all the possible ways the operators could work, because any nontrivial project will define a unmemorizable multitude of types. You would need to figure out the type of all of the variables, then search to see if there's an operator overload that applies in this situation. You'd then need to do a bit of reading to figure out what the overload even does; at least subroutines can be given useful names to assist with this. Your intutions could very well be completely wrong.[1]

Of course, most people won't bother to do this. They'll just trust their intuitions and think that, for example, the minus operator does the opposite of the plus operator, or that one can freely rearrange the operands of the asterisk operator. When their intuitions fail them, they'll get a compilation error at best and a mysterious, hard-to debug, logic error that breaks things without crashing the program at worst.

1 One could argue that this needs to be done for functions as well, but generally one need only look up the function definition, and the way that the types work in the function will be pretty clear from context.

Operator overloading is incomplete

On its own, operator overloading is not enough. After all, if operators can be overloaded, why not functions? And if operators can be overloaded, why can't we make our own? Without these, operator overloading is incomplete.

Function overloading is probably not going to happen. It would require some form of name mangling scheme to differentiate the overloaded functions, and that would necessitate a break in the binary interface. With C being the *lingua franca* of the programming world, compiler creators have a duty of care to those who rely on them, and breaking the binary interface would require pretty much everything to be reconstructed. Good fuckin' luck with that!

Allowing people to contstruct new operators is a bad idea. Let's ignore exactly how it's implemented for a moment and just acknowledge that the more powerful the system is, the more complicated compilers would need to be. For an exceptionally complicated system where a program could completely overhaul the syntax, the system would likely be so complicated that it would be completely impossible for those on low-end systems to spare the storage space or computational power to make their own programs, leaving them at the mercy of those on higher-end systems, which is fucking terrifying to think about. All this suffering for the small reward of some syntactic sugar.

What makes this all worse is that the more powerful an operator system is, the worse it becomes for someone new to a project to get used to things—they might need to get used to what could reasonably be considered an entirely different language! But if the operator system isn't powerful enough, then it will feel incomplete. Much like the markets, the only winning move here is not to play.

Conclusion

As i said at the beginning of this essay, n3051 is unlikely to be accepted. Maybe it will set off further thinking on the matter...but probably not. Partially because of this, i don't plan to send this essay (or, more realistically, a more refined paper based upon this essay) to WG14 to try to convince them away from operator overloading, because i don't think that would be necessary.

Operator overloading is pure syntactic sugar. It is unnecessary, confusing, and incomplete without a more robust system behind it. In order for such a system to be powerful, it would need to be so complicated as to be classist. It would be so much work to construct such a system, all for such a small reward—and i, for one, don't think that all that's worth it.
