Syntactic Heroin

Rodney Bates, Wichita State University

User-defined overloading is a drug. At first, it gives you a quick, feel-good fix. No sense in cluttering up code with verbose and ugly function names such as `IntAbs`, `FloatAbs`, `DoubleAbs`, or `ComplexAbs`; just name them all `Abs`. Even better, use algebraic notation such as `A+B`, instead of `ComplexSum(A,B)`. It certainly makes coding more compact. But a dangerous addiction soon sets in. Languages and programs that were already complex enough to stretch everyone's ability suddenly get much more complicated.

WHAT IS IT?

In general, *overloading* means that a function name or an operator has two or more distinct meanings. When you use it, the types of its operands are used by the language to determine which meaning should apply. This is common in programming languages for built-in operators such as `+`, which could, for example, be integer addition or floating-point addition. The two operations work on different types of operands and require different machine code to be generated, so they are truly distinct meanings. It also happens for some built-in functions, such as an absolute value function `ABS`, which, like `+`, can work on either integer or floating operands.

Equality comparisons, spelled `==` in some languages and `=` in others, commonly take this idea much further, with many overloaded meanings for most of the types in the language. This is likely to include not only built-in types, but also an unbounded subset, if not all, of the programmer-defined types.

When only built-in functions and operators are overloaded and each has only a fixed set of language-defined meanings, there is usually not much trouble. But I am talking about *programmer-defined* overloading. In some languages, the programmer can declare several different functions, in the same scope, with the same name, but different parameter types. A call using this function name is resolved at compile time to one of the several functions by looking at the types of the actual parameters.

The programmer can also overload operators by declaring one or more functions that are called using operator notation. The declarations are pretty much as with any

A dangerous coding

ADDITION THAT LEADS TO

A READABILITY DISASTER.

function, except there is a syntax for giving them names such as `+`, which is already a built-in operator.

When an expression such as `A+B` appears, the types of `A` and `B` determine whether a built-in meaning or programmer-defined meaning of `+` is to be used, and which one. If it's one of the programmer-defined meanings, it amounts to a call on the function that declared it. Sometimes, programmers can even replace built-in meanings of the operator—for example, `+` on integers—with their own functions.

IS IT SYNTACTIC SUGAR?

The best you can say for programmer-defined overloading is that it is only *syntactic sugar*—that is, it makes things look nicer, but it adds no significant programming capabilities at all. Anything you can do with it can be done by just giving all your functions distinct names that are identifiers, not operator symbols, and using plain old-fashioned function calls to invoke them. But is it really sugar?

Overloading function names saves time during writing by saving keystrokes and, more significantly, saves the mental effort of thinking up unique names. The effect on readability is purely negative, however, because things with different meanings look the same.

Even for operators, the effect on readability is debatable at best. At least when the operators have some tradition in mathematics, operator notation is easier to read than function call notation. Like function names, overloaded operators also suffer from different things looking the same. Fortunately, operators have at most two operands, whereas functions sometimes have many.

Unfortunately, given this new toy, programmers can seldom resist overusing it. The result is that all sorts of fundamentally different meanings get overloaded on the same name or operator. Readability goes down the drain.

So there's no real syntactic sugar here. With the operators, at least, you can concoct a few specific examples of calls that look sweeter. But, overall, it's a big dose of vinegar.

Continued on page 62

Continued from page 64

THE PIPER DEMANDS HIS PAY

Even if you like the sugar high, the cost of programmer-defined overloading is exorbitant. Language designers, compiler writers, developers, and users all suffer. Consider C++, starting with declarations. The programmer declares a function with the same name as some other in the same scope. This could actually be a redeclaration of the same function, a different overloaded function, or an illegal overload. It takes careful scrutiny just to be sure what has been declared, and you must compare every declared function with every other in the scope.

Now consider calls. The programmer writes (or, more commonly, reads) an innocent-looking call, say, $F(X,Y,Z)$. What does it mean? First, there is a list of *candidate* functions named F . These are searched for in a multibranching tree of scopes, with five rules giving different ways this tree can branch. Fortunately, the search stops with the first scope that contains any declaration named F , so all candidates will at least come from the same scope.

The candidates are then checked for *viability*. A candidate is viable if all the actual arguments in the call can be implicitly converted to the types of the corresponding formal parameters of the candidate. There are 29 possible kinds of implied conversions that are *standard* (i.e., defined by the language). I have counted the rules conservatively; a liberal count is 138.

Then there are an unlimited number of programmer-defined conversions. These come in two varieties: one, the *user-defined conversion functions*, would likely be written only for this purpose; the other, the *constructors with one parameter*, will usually have been written for another purpose (constructing), and so are likely to get into the overload resolution stew inadvertently.

A single implied conversion sequence can consist of as many as seven conversions in a row, two of them chosen from a set of three possibilities, two from a set of six possibilities, and two from a set of 20. The remaining conversion is from the unbounded set of user-defined conversions. If we assume there is just one of each kind of user-defined conversion, this gives more than a million implied conversion sequences, although many are probably semantically illegal. That is for each parameter. This has to be multiplied by the number of parameters, and that in turn multiplied by the number of candidates. Remember that none of these conversions is written in the code. They are all done invisibly by the compiler.

By now, the initial drug rush is gone, and the crash has begun, as the programmer's brain is addled by lots of complexity beyond what the real problem has. But we are

still a long way from figuring out what $F(X,Y,Z)$ actually calls. Each function that turned out to be viable now needs to be compared with all the others for *goodness* of fit, which is only a partial ordering. For each pair of viable candidates, it has to be first evaluated for each parameter, which is in turn done by ranking the two implied-conversion sequences. There are 18 rules giving partial ordering relations for two implied-conversion sequences. I have also counted these conservatively, as some of the rules have multiple subcases. One of the two sequences can be better or worse than the other, if one of the 18 rules applies. Otherwise, they are not ranked for goodness.

Goodness ranks of parameters are combined by another partial-ordering rule to compare overall goodness of two viable candidates. For one candidate to be better than another, it must be at least as good in every parameter and better in at least one. If the two are unranked in any parameter, they are unranked as functions.

This entire process must be applied to each pair of viable candidates. If there are n of these, there are $(n*(n-1))/2$ pairs. From all this, there must be exactly one viable candidate that is better than every other. If so, it is the function called. If not, there is a compile-time error.

THE WOES OF ADDICTION

Professional language designers, language lawyers, and compiler writers can spend the staff-years necessary to slog through this morass of complexity, even though it is an enormous and completely unnecessary expense. But programmers cannot possibly go through this analysis every time they write or read simple function calls, not even compiler writers working on their own compilers. Only a tiny minority of working C++ programmers could give even something resembling the foregoing, greatly abbreviated summary of the process.

The sad result of this addiction is that there are millions of lines of code, full of operators and function calls, and neither the original programmers nor current maintainers have more than a vague intuitive hope that they know the meaning of what they have written and read. The best that can be hoped for is that the rules of the language, together with programmers' habits in declaring overloaded functions, give either the right answer or compile-time errors. When people's fortunes, health, and lives depend on software, this is not nearly good enough.

WHAT ABOUT BUILT-IN OVERLOADINGS?

I began by describing overloaded meanings that are built in to a language. Are these a drug, too? They don't have the same high price, for several reasons. There is a huge

difference between having a fixed set of specific overloaded meanings in a language and providing a general-purpose system that allows the programmer to declare and invoke an unboundedly expandable set of these. The language complexity of the former is at least an order of magnitude less than the latter.

For example, built-in operators are always available in any context. This eliminates all the complexities of where to look for candidate declarations when resolving an overloaded operator. Also, the set of built-in meanings that a programmer must remember doesn't change from program to program or place to place within a program, as programmer-defined overloads do.

KICK THE DRUG HABIT

User-defined overloading is a syntactic drug that has seduced all too many language designers, programmers, and project managers. Its benefits are at best minor and purely cosmetic. At worst, it is a semantic train wreck, with enormous and unnecessary costs in development of languages and their compilers, programmer training, excessive software development and maintenance budgets and schedules, and poor-quality software.

Those with real courage will break the addiction by

refusing to use programming languages that push this drug. Those who can't manage to break the addiction completely can at least establish coding standards that disallow writing overloaded declarations and hope inadvertent violations of this rule aren't too frequent. Q

REFERENCES

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Reading, MA: Addison-Wesley.

ISO/IEC. Annotated Ada Reference Manual. International Standard 8652:1995(E).

ISO/IEC. Programming Languages—C++. International Standard 14882:1998(E).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RODNEY BATES has worked for 33 years in computer software, two-thirds in industry, the rest in academia. He has been involved mostly with operating systems, compilers, and as a resident programming language lawyer. He is an assistant professor and graduate coordinator in the computer science department at Wichita State University.

© 2005 ACM 1542-7730/05/0600 \$5.00

acm
queue
architecting tomorrow's computing

Unpacking Utility Computing

Ian Foster on grids and their kin
Enterprise grid computing
How on-demand computing changes everything

Plus:
An interview with the man
behind SETI@home and BOINC,
David Anderson

Coming in the July/August issue