commandcenter.blogspot.com

Less is exponentially more

14-17 minutes

Here is the text of the talk I gave at the Go SF meeting in June, 2012.

This is a personal talk. I do not speak for anyone else on the Go team here, although I want to acknowledge right up front that the team is what made and continues to make Go happen. I'd also like to thank the Go SF organizers for giving me the opportunity to talk to you.

I was asked a few weeks ago, "What was the biggest surprise you encountered rolling out Go?" I knew the answer instantly: Although we expected C++ programmers to see Go as an alternative, instead most Go programmers come from languages like Python and Ruby. Very few come from C++.

We—Ken, Robert and myself—were C++ programmers when we designed a new language to solve the problems that we thought needed to be solved for the kind of software we wrote. It seems almost paradoxical that other C++ programmers don't seem to care.

I'd like to talk today about what prompted us to create Go, and why the result should not have surprised us like this. I promise this will be more about Go than about C++, and that if you don't know C++ you'll be able to follow along.

The answer can be summarized like this: Do you think less is more, or less is less?

Here is a metaphor, in the form of a true story. Bell Labs centers were originally assigned three-letter numbers: 111 for Physics Research, 127 for Computing Sciences Research, and so on. In the early 1980s a memo came around announcing that as our understanding of research had grown, it had become necessary to add another digit so we could better characterize our work. So our center became 1127. Ron Hardin joked, half-seriously, that if we really understood our world better, we could drop a digit and go down from 127 to just 27. Of course management didn't get the joke, nor were they expected to, but I think there's wisdom in it. Less can be more. The better you understand, the pithier you can be.

Keep that idea in mind.

Back around September 2007, I was doing some minor but central work on an enormous Google C++ program, one you've all interacted with, and my compilations were taking about 45 minutes on our huge distributed compile cluster. An announcement came around that there was going to be a talk presented by a couple of Google

employees serving on the C++ standards committee. They were going to tell us what was coming in C++0x, as it was called at the time. (It's now known as C++11).

In the span of an hour at that talk we heard about something like 35 new features that were being planned. In fact there were many more, but only 35 were described in the talk. Some of the features were minor, of course, but the ones in the talk were at least significant enough to call out. Some were very subtle and hard to understand, like rvalue references, while others are especially C++-like, such as variadic templates, and some others are just crazy, like user-defined literals.

At this point I asked myself a question: Did the C++ committee really believe that was wrong with C++ was that it didn't have enough features? Surely, in a variant of Ron Hardin's joke, it would be a greater achievement to simplify the language rather than to add to it. Of course, that's ridiculous, but keep the idea in mind.

Just a few months before that C++ talk I had given a talk myself, which you can see on YouTube, about a toy concurrent language I had built way back in the 1980s. That language was called Newsqueak and of course it is a precursor to Go.

I gave that talk because there were ideas in Newsqueak that I missed in my work at Google and I had been thinking about them again. I was convinced they would

make it easier to write server code and Google could really benefit from that.

I actually tried and failed to find a way to bring the ideas to C++. It was too difficult to couple the concurrent operations with C++'s control structures, and in turn that made it too hard to see the real advantages. Plus C++ just made it all seem too cumbersome, although I admit I was never truly facile in the language. So I abandoned the idea.

But the C++0x talk got me thinking again. One thing that really bothered me—and I think Ken and Robert as well—was the new C++ memory model with atomic types. It just felt wrong to put such a microscopically-defined set of details into an already over-burdened type system. It also seemed short-sighted, since it's likely that hardware will change significantly in the next decade and it would be unwise to couple the language too tightly to today's hardware.

We returned to our offices after the talk. I started another compilation, turned my chair around to face Robert, and started asking pointed questions. Before the compilation was done, we'd roped Ken in and had decided to do something. We did not want to be writing in C++ forever, and we—me especially—wanted to have concurrency at my fingertips when writing Google code. We also wanted to address the problem of "programming in the large" head on, about which more later.

We wrote on the white board a bunch of stuff that we wanted, desiderata if you will. We thought big, ignoring detailed syntax and semantics and focusing on the big picture.

I still have a fascinating mail thread from that week. Here are a couple of excerpts:

Robert: *Starting point: C, fix some obvious flaws, remove crud, add a few missing features.*

Rob: *name: 'go'. you can invent reasons for this name but it has nice properties. it's short, easy to type. tools: goc, gol, goa. if there's an interactive debugger/interpreter it could just be called 'go'. the suffix is .go.*

Robert *Empty interfaces: interface {}. These are implemented by all interfaces, and thus this could take the place of void*.*

We didn't figure it all out right away. For instance, it took us over a year to figure out arrays and slices. But a significant amount of the flavor of the language emerged in that first couple of days.

Notice that Robert said C was the starting point, not C++. I'm not certain but I believe he meant C proper, especially because Ken was there. But it's also true that, in the end, we didn't really start from C. We built from scratch, borrowing only minor things like operators and brace brackets and a few common keywords. (And of course we also borrowed ideas from other languages we knew.) In

any case, I see now that we reacted to C++ by going back down to basics, breaking it all down and starting over. We weren't trying to design a better C++, or even a better C. It was to be a better language overall for the kind of software we cared about.

In the end of course it came out quite different from either C or C++. More different even than many realize. I made a list of significant simplifications in Go over C and C++:

- regular syntax (don't need a symbol table to parse)
- garbage collection (only)
- no header files
- explicit dependencies
- no circular dependencies
- constants are just numbers
- int and int32 are distinct types
- letter case sets visibility
- methods for any type (no classes)
- no subtype inheritance (no subclasses)
- package-level initialization and well-defined order of initialization
- files compiled together in a package
- package-level globals presented in any order

- no arithmetic conversions (constants help)
- interfaces are implicit (no "implements" declaration)
- embedding (no promotion to superclass)
- methods are declared as functions (no special location)
- methods are just functions
- interfaces are just methods (no data)
- methods match by name only (not by type)
- no constructors or destructors
- postincrement and postdecrement are statements, not expressions
- no preincrement or predecrement
- assignment is not an expression
- evaluation order defined in assignment, function call (no "sequence point")
- no pointer arithmetic
- memory is always zeroed
- legal to take address of local variable
- no "this" in methods
- segmented stacks
- no const or other type annotations
- no templates

- no exceptions
- builtin string, slice, map
- array bounds checking

And yet, with that long list of simplifications and missing pieces, Go is, I believe, more expressive than C or C++. Less can be more.

But you can't take out everything. You need building blocks such as an idea about how types behave, and syntax that works well in practice, and some ineffable thing that makes libraries interoperate well.

We also added some things that were not in C or C++, like slices and maps, composite literals, expressions at the top level of the file (which is a huge thing that mostly goes unremarked), reflection, garbage collection, and so on. Concurrency, too, naturally.

One thing that is conspicuously absent is of course a type hierarchy. Allow me to be rude about that for a minute.

Early in the rollout of Go I was told by someone that he could not imagine working in a language without generic types. As I have reported elsewhere, I found that an odd remark.

To be fair he was probably saying in his own way that he really liked what the STL does for him in C++. For the purpose of argument, though, let's take his claim at face value.

What it says is that he finds writing containers like lists of ints and maps of strings an unbearable burden. I find that an odd claim. I spend very little of my programming time struggling with those issues, even in languages without generic types.

But more important, what it says is that *types* are the way to lift that burden. *Types*. Not polymorphic functions or language primitives or helpers of other kinds, but *types*.

That's the detail that sticks with me.

Programmers who come to Go from C++ and Java miss the idea of programming with types, particularly inheritance and subclassing and all that. Perhaps I'm a philistine about types but I've never found that model particularly expressive.

My late friend Alain Fournier once told me that he considered the lowest form of academic work to be taxonomy. And you know what? Type hierarchies are just taxonomy. You need to decide what piece goes in what box, every type's parent, whether A inherits from B or B from A. Is a sortable array an array that sorts or a sorter represented by an array? If you believe that types address all design issues you must make that decision.

I believe that's a preposterous way to think about programming. What matters isn't the ancestor relations between things but what they can do for you.

That, of course, is where interfaces come into Go. But they're part of a bigger picture, the true Go philosophy.

If C++ and Java are about type hierarchies and the taxonomy of types, Go is about composition.

Doug McIlroy, the eventual inventor of Unix pipes, wrote in 1964 (!):

> We should have some ways of coupling programs like garden hose--screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.

That is the way of Go also. Go takes that idea and pushes it very far. It is a language of composition and coupling.

The obvious example is the way interfaces give us the composition of components. It doesn't matter what that thing is, if it implements method M I can just drop it in here.

Another important example is how concurrency gives us the composition of independently executing computations.

And there's even an unusual (and very simple) form of type composition: embedding.

These compositional techniques are what give Go its flavor, which is profoundly different from the flavor of C++ or Java programs.

==========

There's an unrelated aspect of Go's design I'd like to touch upon: Go was designed to help write big programs, written and maintained by big teams.

There's this idea about "programming in the large" and somehow C++ and Java own that domain. I believe that's just a historical accident, or perhaps an industrial accident. But the widely held belief is that it has something to do with object-oriented design.

I don't buy that at all. Big software needs methodology to be sure, but not nearly as much as it needs strong dependency management and clean interface abstraction and superb documentation tools, none of which is served well by C++ (although Java does noticeably better).

We don't know yet, because not enough software has been written in Go, but I'm confident Go will turn out to be a superb language for programming in the large. Time will tell.

==========

Now, to come back to the surprising question that opened my talk:

Why does Go, a language designed from the ground up for what what C++ is used for, not attract more C++ programmers?

Jokes aside, I think it's because Go and C++ are profoundly

different philosophically.

C++ is about having it all there at your fingertips. I found this quote on a C++11 FAQ:

> The range of abstractions that C++ can express elegantly, flexibly, and at zero costs compared to hand-crafted specialized code has greatly increased.

That way of thinking just isn't the way Go operates. Zero cost isn't a goal, at least not zero CPU cost. Go's claim is that minimizing programmer effort is a more important consideration.

Go isn't all-encompassing. You don't get everything built in. You don't have precise control of every nuance of execution. For instance, you don't have RAII. Instead you get a garbage collector. You don't even get a memory-freeing function.

What you're given is a set of powerful but easy to understand, easy to use building blocks from which you can assemble—compose—a solution to your problem. It might not end up quite as fast or as sophisticated or as ideologically motivated as the solution you'd write in some of those other languages, but it'll almost certainly be easier to write, easier to read, easier to understand, easier to maintain, and maybe safer.

To put it another way, oversimplifying of course:

Python and Ruby programmers come to Go because they

don't have to surrender much expressiveness, but gain performance and get to play with concurrency.

C++ programmers *don't* come to Go because they have fought hard to gain exquisite control of their programming domain, and don't want to surrender any of it. To them, software isn't just about getting the job done, it's about doing it a certain way.

The issue, then, is that Go's success would contradict their world view.

And we should have realized that from the beginning. People who are excited about C++11's new features are not going to care about a language that has so much less. Even if, in the end, it offers so much more.

Thank you.