The Go Programming Language

The Go Blog

Go's Declaration Syntax

7 July 2010

Introduction

Newcomers to Go wonder why the declaration syntax is different from the tradition established in the C family. In this post we'll compare the two approaches and explain why Go's declarations look as they do.

C syntax

First, let's talk about C syntax. C took an unusual and clever approach to declaration syntax. Instead of describing the types with special syntax, one writes an expression involving the item being declared, and states what type that expression will have. Thus

```
int x;
```

Next article

Share Memory By Communicating

Previous article

Go Programming session video from Google I/O

Links

golang.org
Install Go
A Tour of Go
Go Documentation
Go Mailing List
Go on Google+
Go+ Community
Go on Twitter

Blog index

declares x to be an int: the expression 'x' will have type int. In general, to figure out how to write the type of a new variable, write an expression involving that variable that evaluates to a basic type, then put the basic type on the left and the expression on the right.

Thus, the declarations

```
int *p;
int a[3];
```

state that p is a pointer to int because '*p' has type int, and that a is an array of ints because a[3] (ignoring the particular index value, which is punned to be the size of the array) has type int.

What about functions? Originally, C's function declarations wrote the types of the arguments outside the parens, like this:

```
int main(argc, argv)
    int argc;
    char *argv[];
{ /* ... */ }
```

Again, we see that main is a function because the expression main(argc, argv) returns an int. In modern notation we'd write

```
int main(int argc, char *argv[]) { /* ... */ }
```

but the basic structure is the same.

This is a clever syntactic idea that works well for simple types but can get confusing fast. The famous example is declaring a function pointer. Follow the rules and you get this:

```
int (*fp)(int a, int b);
```

Here, fp is a pointer to a function because if you write the expression (*fp)(a, b) you'll call a function that returns int. What if one of fp's arguments is itself a function?

```
int (*fp)(int (*ff)(int x, int y), int b)
```

That's starting to get hard to read.

Of course, we can leave out the name of the parameters when we declare a function, so main can be declared

```
int main(int, char *[])
```

Recall that argv is declared like this,

```
char *argv[]
```

so you drop the name from the middle of its declaration to construct its type. It's not obvious, though, that you declare something of type char *[] by putting its name in the middle.

And look what happens to fp's declaration if you don't name the parameters:

```
int (*fp)(int (*)(int, int), int)
```

Not only is it not obvious where to put the name inside

```
int (*)(int, int)
```

it's not exactly clear that it's a function pointer declaration at all. And what if the return type is a function pointer?

```
int (*(*fp)(int (*)(int, int), int))(int, int)
```

It's hard even to see that this declaration is about fp.

You can construct more elaborate examples but these should illustrate some of the difficulties that C's declaration syntax can introduce.

There's one more point that needs to be made, though. Because type and declaration syntax are the same, it can be difficult to parse expressions with types in the middle. This is why, for instance, C casts always parenthesize the type, as in

```
(int)M_PI
```

Go syntax

Languages outside the C family usually use a distinct type syntax in declarations. Although it's a separate point, the name usually comes first, often followed by a colon. Thus our examples above become something like (in a fictional but illustrative language)

```
x: int
p: pointer to int
a: array[3] of int
```

These declarations are clear, if verbose - you just read them left to right. Go takes its cue from here, but in the interests of brevity it drops the colon and removes some of the keywords:

```
x int
p *int
a [3]int
```

There is no direct correspondence between the look of [3]int and how to use a in an expression. (We'll come back to pointers in the next section.) You gain clarity at the cost of a separate syntax.

Now consider functions. Let's transcribe the declaration for main as it would read in Go, although the real main function in Go takes no arguments:

```
func main(argc int, argv []string) int
```

Superficially that's not much different from C, other than the change from `char` arrays to strings, but it reads well from left to right:

function main takes an int and a slice of strings and returns an int.

Drop the parameter names and it's just as clear - they're always first so there's no confusion.

```
func main(int, []string) int
```

One merit of this left-to-right style is how well it works as the types become more complex. Here's a declaration of a function variable (analogous to a function pointer in C):

```
f func(func(int,int) int, int) int
```

Or if f returns a function:

```
f func(func(int,int) int, int) func(int, int) int
```

It still reads clearly, from left to right, and it's always obvious which name is being declared - the name comes first.

The distinction between type and expression syntax makes it easy to write and invoke closures in Go:

```
sum := func(a, b int) int { return a+b } (3, 4)
```

Pointers

Pointers are the exception that proves the rule. Notice that in arrays and slices, for instance, Go's type syntax puts the brackets on the left of the type but the expression syntax puts them on the right of the expression:

```
var a []int
x = a[1]
```

For familiarity, Go's pointers use the * notation from C, but we could not bring ourselves to make a similar reversal for pointer types. Thus pointers work like this

```
var p *int
x = *p
```

We couldn't say

```
var p *int
x = p*
```

because that postfix * would conflate with multiplication. We could have used the Pascal ^, for example:

```
var p ^int
x = p^
```

and perhaps we should have (and chosen another operator for xor), because the prefix asterisk on both types and expressions complicates things in a number of ways. For instance, although one can write

```
[]int("hi")
```

as a conversion, one must parenthesize the type if it starts with a *:

```
(*int)(nil)
```

Had we been willing to give up * as pointer syntax, those parentheses would be unnecessary.

So Go's pointer syntax is tied to the familiar C form, but those ties mean that we cannot break completely from using parentheses to disambiguate types and expressions in the grammar.

Overall, though, we believe Go's type syntax is easier to understand than C's, especially when things get complicated.

Notes

Go's declarations read left to right. It's been pointed out that C's read in a spiral! See The "Clockwise/Spiral Rule" by David Anderson.

By Rob Pike

Related articles

- Two recent Go articles
- Two recent Go talks
- Go videos from Google I/O 2012