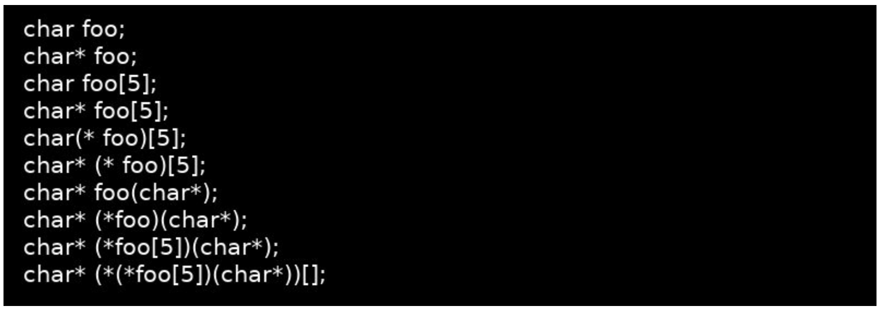
Brian Barto Follow
Dec 29, 2016 · 4 min read

Untangling Complex Declarations in C

Here is a list of C declarations of increasing complexity. Until recently I could not have told you what all of them meant.



```
char foo;
char* foo;
char foo[5];
char* foo[5];
char(* foo)[5];
char* (* foo)[5];
char* foo(char*);
char* (*foo)(char*);
char* (*foo[5])(char*);
char* (*(*foo[5])(char*))[];
```

Now I can explain all of them fairly confidently after having learning about the precedence rules for C declarations, a topic that seems important and yet was strangely absent from the highly rated book that I purchased when I initially set out to learn C. It wasn't until I read a second book on advanced C topics that I learned about these rules.

For me, learning how to interpret complex declarations was an important step towards using complex data structures and expanding my capabilities as a programmer. If I can't read a complex declaration, then I can't write one either, and I therefor might avoid using them altogether and opt for less ideal solutions.

I originally assumed that becoming fluent with complex declarations was, like many things in life, a process that slowly adds clarity with repeated exposure and use. But I was surprised to find that there are only a few rules to memorize that allow you to easily break down any declaration into small comprehensible components.

They are referred to as the precedence rules for C declarations. From high to low, they are:

1. Parentheses grouping together parts of a declaration.
2. The postfix operators: parentheses () indicating a function, and square brackets [] indicating an array.
3. The prefix operator: the asterisk denoting "pointer to"

To put it another way: Parentheses that are grouping together multiple parts of a declaration have the highest precedence. Next are the postfix operators () and []. Last is the prefix operator *.

The 4th declaration on my list above might be a good example to start with.

```
char* foo[5];
```

The precedence rules say that the square brackets are higher than the asterisk. So *foo* is **an array of pointers to a char**, not a pointer to an array of characters.

The step-by-step process for applying the precedence rules to this example is as follows.

Starting with the name *foo*, the interpretation starts out as: "**foo is a…**"

Next, the precedence rules state that the square brackets on the right outweigh the asterisk on the left.

```
char* foo[5];
```

Now the interpretation is: "foo is an **array of 5**"

The only option next is the asterisk on the left.

```
char* foo[5];
```

"foo is an array of 5 **pointers to**"

And the last part of the declaration is the data type *char*.

```
char* foo[5];
```

The final interpretation is: "foo is an array of 5 pointers to **a char**"

If I had wanted *foo* to be a pointer to an array, instead of an array of pointers, I would have needed to include a set of parentheses like in the following example.

```
char(* foo)[5];
// foo is a

char(* foo)[5];
// foo is a pointer

char(* foo)[5];
// foo is a pointer to an array of five

char(* foo)[5];
// foo is a pointer to an array of five chars
```

This time the parentheses gave the asterisk a higher precedence than the square brackets, which changed *foo* from an array of pointers to a pointer to an array.

The 8th declaration on the list includes two sets of parentheses.

```
char* (*foo)(char*);
// foo is a

char* (*foo)(char*);
// foo is a pointer

char* (*foo)(char*);
// foo is a pointer to a function that accepts a pointer to
a char

char* (*foo)(char*);
// foo is a pointer to a function that accepts a pointer to
a char and returns a pointer

char* (*foo)(char*);
// foo is a pointer to a function that accepts a pointer to
a char and returns a pointer to a char
```

I took the liberty of interpreting the function parentheses and it's parameters as a single step, but it may be easier to interpret each parameter separately. The same precedence rules can be applied for each parameter as they are for the declaration as a whole.

As a final example, here is the most complex declaration on the list:

```
char* (*(*foo[5])(char*))[];
// foo is a

char* (*(*foo[5])(char*))[];
// foo is an array of 5

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to a function that accepts
a pointer to a char

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to a function that accepts
a pointer to a char and returns a pointer to

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to a function that accepts
a pointer to a char and returns a pointer to an array of

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to a function that accepts
a pointer to a char and returns a pointer to an array of
pointers to

char* (*(*foo[5])(char*))[];
// foo is an array of 5 pointers to a function that accepts
a pointer to a char and returns a pointer to an array of
pointers to a char.
```

I should probably mention that this last example is used to show how to apply these precedence rules in an extreme case, but using this sort of declaration in a real world scenario may be considered unnecessarily complex and poor practice.

Also note that none of these examples include keywords like ***const*** or ***volatile***, and they don't show examples of declarations for ***structs***, ***enums***, or ***unions***. If you want to read more on this topic you can google "declaration precedence rules for C" to find lots of good resources to help complete your understanding of this topic.