…Even relatively new C programmers have no trouble reading simple C declarations such as

```
int       foo[5];       // foo is an array of 5 ints
char     *foo;          // foo is a pointer to char
double    foo();        // foo is a function returning a double
```

but as the declarations get a bit more involved, it's more difficult to know exactly what you're looking at.

```
char *(*(**foo[][8])())[];  // huh ?????
```

It turns out that the rules for reading an arbitrarily-complex C variable declaration are easily learned by even beginning programmers (though how to actually *use* the variable so declared may be well out of reach).

This Tech Tip shows how to do it.

Basic and Derived Types

In addition to one variable name, a declaration is composed of one "basic type" and zero or more "derived types", and it's crucial to understand the distinction between them.

The complete list of basic types is:

• **char**	• **signed char**	• **unsigned char**	
• **short**	• **unsigned short**		
• **int**	• **unsigned int**		
• **long**	• **unsigned long**		
• **float**	• **double**	• **void**	
• **struct *tag***	• **union *tag***	• **enum *tag***	
• **long long**	• **unsigned long long**	• **long double**	*ANSI/ISO C only*

A declaration can have exactly **one** basic type, and it's always on the far left of the expression.

The "basic types" are augmented with "derived types", and C has three of them:

- *** pointer to...**

  This is denoted by the familiar ***** character, and it should be self evident that a pointer always has to point ***to*** something.

- **[] array of...**

  "Array of" can be undimensioned -- **[]** -- or dimensioned -- **[10]** -- but the sizes don't really play significantly into reading a declaration. We typically include the size in the description. It should be clear that arrays have to be "arrays ***of***" something.

- **() function returning...**

  This is usually denoted by a pair of parentheses together - **()** - though it's also possible to find a prototype parameter list inside. Parameters lists (if present) don't really play into reading a declaration, and we typically ignore them. We'll note that parens used to represent "function returning" are different than those used for grouping: grouping parens *surround* the variable name, while "function returning" parens are always on the right.

> Functions are meaningless unless they *return* something (and we accommodate the **void** type by waving the hand and pretend that it's "returning" void).

A derived type *always* modifies something that follows, whether it be the basic type or another derived type, and to make a declaration read properly one must always include the preposition ("to", "of", "returning"). Saying "pointer" instead of "pointer to" will make your declarations fall apart.

It's possible that a type expression may have no derived types (e.g., **"int i"** describes "i is an int"), or it can have many. Interpreting the derived types is usually the sticking point when reading a complex declaration, but this is resolved with operator precedence in the next section.

Operator Precedence

Almost every C programmer is familiar with the operator precedence tables, which give rules that say (for instance) multiply and divide have higher precedence than ("are preformed before") addition or subtraction, and parentheses can be used to alter the grouping. This seems natural for "normal" expressions, but the same rules do indeed apply to declarations - they are *type* expressions rather than *computational* ones.

The "array of" **[]** and "function returning" **()** type operators have higher precedence than "pointer to" *****, and this leads to some fairly straightforward rules for decoding.

Always start with the variable name:

> **foo is** ...

and *always* end with the basic type:

> foo is ... **int**

The "filling in the middle" part is usually the trickier part, but it can be summarize with this rule:

> "go right when you can, go left when you must"

Working your way out from the variable name, honor the precedence rules and consume derived-type tokens to the right as far as possible without bumping into a grouping parenthesis. Then go left to the matching paren.

A simple example

We'll start with a simple example:

```
long **foo[7];
```

We'll approach this systematically, focusing on just one or two small part as we develop the description in English. As we do it, we'll show the focus of our attention in ***red***, and ~~strike out~~ the parts we've finished with.

```
long **foo [7];
```

> Start with the variable name and end with the basic type:
>
> ***foo is*** ... ***long***

```
~~long~~ ** ~~foo~~[7];
```

> At this point, the variable name is touching two derived types: "array of 7" and "pointer to", and the rule is to go right when you can, so in this case we consume the "array of 7"
>
> foo is ***array of 7*** ... long

: ~~long~~ ***~~ ~~foo[7]~~ ;

Now we've gone as far right as possible, so the innermost part is only touching the "pointer to" - consume it.

foo is array of 7 ***pointer to*** ... long

: ~~long~~ ***~~ ~~*foo[7]~~ ;

The innermost part is now only touching a "pointer to", so consume it also.

foo is array of 7 pointer to ***pointer to*** long

This completes the declaration!

A hairy example

To really test our skills, we'll try a very complex declaration that very well may never appear in real life (indeed: we're hard-pressed to think of how this could actually be used). But it shows that the rules scale to very complex declarations.

: ***char*** ***(*(*****foo*** **[][8])())[];**

All declaration start out this way: "variable name is .... basictype"

foo is ... ***char***

: ~~**char**~~ ***(*(**~~foo~~** ***[]*** **[8])())[];**

The innermost part touches "array of" and "pointer to" - go right.

foo is ***array of*** ... char

: ~~**char**~~ ***(*(**~~foo[]~~** ***[8]*** **)())[];**

It's common in a declaration to alternate right and left, but this is not the rule: the rule is to go as far right as we can, and here we find that the innermost part still touches "array of" and "pointer to". Again, go right.

foo is array of ***array of 8*** ... char

: ~~**char**~~ ***(*(*~~ ~~foo[][8]~~** **)())[];**

Now we've hit parenthesis used for grouping, and this halts our march to the right. So we have to backtrack to collect all the parts to the left (but only as far as the paren). This consumes the "pointer to":

foo is array of array of 8 ***pointer to*** ... char

: ~~**char**~~ ***(*(~~*** ~~*foo[][8]~~** **)())[];**

Again we are backtracking to the left, so we consume the next "pointer to":

foo is array of array of 8 pointer to ***pointer to*** ... char

: ~~**char**~~ ***(*~~(**foo[][8])~~** ***~~()~~*** **)[];**

After consuming the "pointer to" in the previous step, this finished off the entire parenthesized subexpression, so we "consume" the parens too. This leaves the innermost part touching "function returning" on the right, and "pointer to" on the left - go right:

foo is array of array of 8 pointer to pointer to ***function returning*** ... char

: ~~**char**~~ ***(~~*** ~~(**foo[][8])()~~** **)[];**

Again we hit grouping parenthesis, so backtrack to the left:

foo is array of array of 8 pointer to pointer to function returning ***pointer to*** ... char

: ~~**char**~~ *** ~~(*(**foo[][8])())~~** ***~~[]~~*** **;**

Consuming the grouping parentheses, we then find that the innermost part is touching "array of" on the right, and "pointer to" on the left. Go right:

foo is array of array of 8 pointer to pointer to function returning pointer to ***array of*** ... char

: ~~char~~ * ~~(*(**foo[][8])())[]~~ ;

Finally we're left with only "pointer to" on the left: consume it to finish the declaration.

foo is array of array of 8 pointer to pointer to function returning pointer to array of ***pointer to*** char

We have no idea how this variable is useful, but at least we can describe the type correctly.

Note that example won't compile unless it's initialized to provide the dimension of the innermost array, though none of this changes the fact that nobody would ever *actually* use this for anything:

```
// explicit initialization
char *(*(**foo[][8])())[] = { 0 };

// implicit init from function call

void myfunction(char *(*(**foo[][8])())[])
{
        ...
}
```

Abstract Declarators

The C standard describes an "abstract declarator", which is used when a type needs to be described but not associated with a variable name. These occur in two places -- casts, and as arguments to **sizeof** -- and they can look intimidating:

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

To the obvious question of "where does one start?", the answer is "find where the variable name would go, then treat it like a normal declaration". There is only one place where a variable name could possibly go, and locating it is actually straightforward. Using the syntax rules, we know that:

- to the right of all the "pointer to" derived type tokens
- to the left of all "array of" derived type tokens
- to the left of all "function returning" derived type tokens
- inside all the grouping parentheses

Looking at the example, we see that the rightmost "pointer to" sets one boundary, and the leftmost "function returning" sets another one:

~~int (*(*~~ • ~~)~~ • ~~())()~~

The red • indicators show the only two places that could possibly hold the variable name, but the leftmost one is the only one that fits the "inside the grouping parens" rule. This gives us our declaration as:

```
int (*(*foo)())()
```

which our "normal" rules describe as:

foo is a pointer to function returning pointer to function returning int

Semantic restrictions/notes

Not all combinations of derived types are allowed, and it's possible to create a declaration that perfectly follows the syntax rules but is nevertheless not legal in C (e.g., *syntactically* valid but *semantically* invalid). We'll touch on them here.

- **Can't have arrays of functions**

  Use "array of pointer to function returning..." instead.

- **Functions can't return functions**

  Use "function returning pointer to function returning..." instead.

- **Functions can't return arrays**

  Use "function returning pointer to array of..." instead.

- **In arrays, only the leftmost [] can be undimensioned**

  C supports multi-dimensional arrays (e.g., **char foo[1][2][3][4]**), though in practice this often suggests poor data structuring. Nevertheless, when there is more than one array dimension, only the leftmost one is allowed to be empty. **char foo[]** and **char foo[][5]** are legal, but **char foo[5][]** is not.

- **"void" type is restricted**

  Since **void** is a special pseudo-type, a variable with this basic type is only legal with a final derived type of "pointer to" or "function returning". It's not legal to have "array of void" or to declare a variable of just type "void" without any derived types.

```
void *foo;          // legal
void foo();         // legal
void foo;           // not legal
void foo[];         // not legal
```

Adding calling-convention types

On the Windows platform, it's common to decorate a function declaration with an indication of its *calling convention*. These tell the compiler which mechanism should be used to call the function in question, and the method used to call the function *must* be the same one which the function expects. They look like:

```
extern int __cdecl main(int argc, char **argv);

extern BOOL __stdcall DrvQueryDriverInfo(DWORD dwMode, PVOID pBuffer,
                                DWORD cbBuf, PDWORD pcbNeeded);
```

These decorations are very common in Win32 development, and are straightforward enough to understand. More information can be found in Unixwiz.net Tech Tip: Using Win32 calling conventions

Where it gets somewhat more tricky is when the calling convention must be incorporated into a pointer (including via a typedef), because the tag doesn't seem to fit into the normal scheme of things. These are often used (for instance) when dealing with the **LoadLibrary()** and **GetProcAddress()** API calls to call a function from a freshly-loaded DLL.

We commonly see this with typedefs:

```
typedef BOOL (__stdcall *PFNDRVQUERYDRIVERINFO)(
    DWORD   dwMode,
    PVOID   pBuffer,
    DWORD   cbBuf,
    PDWORD  pcbNeeded
    );
```

```
...

/* get the function address from the DLL */
pfnDrvQueryDriverInfo = (PFNDRVRQUERYDRIVERINFO)
        GetProcAddress(hDll, "DrvQueryDriverInfo")
```

The calling convention is an attribute of the *function*, not the *pointer*, so in the usual reading puts it after the pointer but inside the grouping parenthesis:

```
BOOL (__stdcall *foo)(...);
```

is read as:

foo is a pointer
to a __stdcall function
returning BOOL.
