

Princeton University

COS 217: Introduction to Programming Systems

Complex C Declarations

The document shown below is retrieved from this site: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html. It describes the "right-left rule," a technique for understanding and composing complex C declarations.

The "right-left" rule is a completely regular rule for deciphering C declarations. It can also be useful in creating them.

First, symbols. Read

*	as "pointer to"	- always on the left side
[]	as "array of"	- always on the right side
()	as "function returning"	- always on the right side

as you encounter them in the declaration.

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

STEP 2

Look at the symbols on the right of the identifier. If, say, you find "(" there, then you know that this is the declaration for a function. So you would then have "identifier is function returning". Or if you found a "[" there, you would say "identifier is array of". Continue right until you run out of symbols *OR* hit a *right* parenthesis ")". (If you hit a left parenthesis, that's the beginning of a () symbol, even if there is stuff in between the parentheses. More on that below.)

STEP 3

Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like "int"), just say it. Otherwise, translate it into English using that table above. Keep going left until you run out of symbols *OR* hit a *left* parenthesis "(".

Now repeat steps 2 and 3 until you've formed your declaration. Here are some examples:

```
int *p[];

1) Find identifier.      int *p[];
                        ^
                        "p is"

2) Move right until out of symbols or left parenthesis hit.
                        int *p[];
                        ^^
                        "p is array of"

3) Can't move right anymore (out of symbols), so move left and find:
                        int *p[];
                        ^
                        "p is array of pointer to"

4) Keep going left and find:
                        int *p[];
                        ^^^
                        "p is array of pointer to int".
```

(or "p is an array where each element is of type pointer to int")

Another example:

```
int *(*func())();
```

1) Find the identifier. int *(*func())();
 ^^^^

"func is"

2) Move right. int *(*func())();
 ^^

"func is function returning"

3) Can't move right anymore because of the right parenthesis, so move left.
 int *(*func())();
 ^

"func is function returning pointer to"

4) Can't move left anymore because of the left parenthesis, so keep going right.
 int *(*func())();
 ^^

"func is function returning pointer to function returning"

5) Can't move right anymore because we're out of symbols, so go left.
 int *(*func())();
 ^

"func is function returning pointer to function returning pointer to"

6) And finally, keep going left, because there's nothing left on the right.
 int *(*func())();
 ^^^

"func is function returning pointer to function returning pointer to int".

As you can see, this rule can be quite useful. You can also use it to sanity check yourself while you are creating declarations, and to give you a hint about where to put the next symbol and whether parentheses are required.

Some declarations look much more complicated than they are due to array sizes and argument lists in prototype form. If you see "[3]", that's read as "array (size 3) of...". If you see "(char *,int)" that's read as "function expecting (char *,int) and returning...". Here's a fun one:

```
int ((*fun_one)(char *,double))[9][20];
```

I won't go through each of the steps to decipher this one.

Ok. It's:

```
"fun_one is pointer to function expecting (char *,double) and  
returning pointer to array (size 9) of array (size 20) of int."
```

As you can see, it's not as complicated if you get rid of the array sizes and argument lists:

```
int ((*fun_one())[][]);
```

You can decipher it that way, and then put in the array sizes and argument lists later.

Some final words:

It is quite possible to make illegal declarations using this rule, so some knowledge of what's legal in C is necessary. For instance, if the above had been:

```
int ((*fun_one())[][]);
```

