C Declarations: A Short Primer

Based on an article by Greg Comeau,

published in the September 1998 edition of the Microsoft Systems Journal

In the ariticle "A Guide to Understanding Even the Most Complex C Declarations", Greg Comeau presents a set of rules that can be applied to interpret any C declaration however complex it may seem. While the rules are intuitive and might appeal to most advanced C programmers, the beginner may find them difficult to grasp. He does however start the article by presenting a simple rule-set to read and write Kernighan and Ritchie (of the famous book, The C Progamming Language, 1978) style declarations. In this Primer I will present these rules and elaboarte on them with examples.

Here is the standard sytax for C Declarations:

The sytax of a C declaration is of the form: storage-class type qualifier declarator = initializer; where storage-class is only one of the following: typedef extern static auto register A type could be one or more of the following: void char short, int, long float, double signed, unsigned struct ... union ... typedef type A qualifier could be one or more of the following: const volatile A declarator contains an identifier and one or more, or none at all, of the following in a variety of combinations: * () [] possibly grouped within parentheses to create different bindings

The term storage-class refers to the method by which an object is assigned space in memory. Chapter 4 of the C Primer gives detailed descriptions of what each of the storage-classes mean. Suffice it to say that understanding what is being declared has no bearing on the storage-class, as it specifically tells you where it is being declared and assigned space in memory. Also, qualifiers (const, volatile) refer respectively to the non-modifiability of an entity, and the fact that the entity in question is modified elsewhere. Therefore, henceforth we will ignore these two pieces of information.

The above definition simply says what a declaration ought to look like (the syntax that is). The key phrase in the above definition is "to create different bindings". What this means is, to give different interpretations to the declaration based on parenthesizing the declaration. All one has to understand any complex C declaration then, is to know that these declarations are based on the C operator precedence chart, the same one you use to evaluate expressions in C:

Precedence	Operators	Associativity
highest	() [] . -> ++(postfix) --(postfix)	left to right
	++(prefix) --(prefix) !~ **sizeof**(type) +(unary) -(unary) &(address) *(dereference)	right to left
	* / %	left to right

	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= <<= >>= |= &= ^=	right to left
lowest	,	left to right

This chart is complicated because it gives the precedence and associativity of all C operators. With declarations, we are only dealing with unary tokens (unary operators need only one operand) so it is a lot simpler The operators of interest to us are marked in red in the above table.

So, here then are the rules for reading and writing C declarations:

1. Parenthesize declarations as if they were expressions.
2. Locate the innermost parentheses.
3. Say "identifier is" where the identifier is the name of the variable.
   a. Say "an array of X" if you see [X].
   b. Say "a pointer to" if you see *.
   c. Say "A function returning" if you see ();
4. Move to the next set of parentheses.
5. If more, go back to 3.
6. Else, say "type" for the remaining type left (such as short int)

Here are some examples to clarify this process:

Example 1:

    int i;

The parenthesization of the above declaration is:

    int (i); {1}

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
No more parentheses left so we say "an int". {4,5,6}
That is, "**i is an int**"

Example 2:

    int *i;

The parenthesization of the above declaration is:

    int (*(i)); {1}

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is (i) {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: (*(i)) {4}
Go back to step 3. {5}
We say "a pointer to" since we see a * {3.b}
No more parentheses left so we say "an int". {4,5,6}
That is, "**i is a pointer to an int**"

Example 3:

    int *i[3];

The parenthesization of the above declaration is:

int (*((i)[3])); {1} // Note that () and [] have the same
// precedence but we deal with them from left to right.

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is **(**i**)** {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: **(**(i)[3]**)** {4}
Go back to step 3. {5}
We say "an array of 3 ..." since we see a [3] {3.a}
Move to the next set of parenthesis: **(***((i)[3])**)** {4}
Go back to step 3. {5}
No more parentheses left so we say "ints". {4,5,6}
That is, "**i is an array of 3 ints**"

Example 4:

int (*i)[3];

The parenthesization of the above declaration is:

int ((*(i))[3]); {1} // Note that parentheses are valid tokens
// in a declaration and therefore must be
// be left in place when finding the final
// parenthesization

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is **(**i**)** {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: **(***(i)**)** {4}
Go back to step 3. {5}
We say "a pointer to ..." since we see a * {3.b}
Move to the next set of parenthesis: **(**(*(i))[3]**)** {4}
Go back to step 3. {5}
We say "an array of 3 ..." since we see a [3] {3.a}
No more parentheses left so we say "ints". {4,5,6}
That is, "**i is a pointer to an array of 3 ints**"

Example 5:

int *i();

The parenthesization of the above declaration is:

int (*((i)())); {1} // Note that * has a lower precedence than
// parentheses, ()

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is **(**i**)** {2}
// One could argue that () is also the innermost parenthesis but
// it does not contain anything so we know it must indicate
// a function
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: **(**(i)()**)** {4}
Go back to step 3. {5}
We say "a function returning" since we see a () {3.c}
Move to the next set of parenthesis: (*((i)())) {4}
Go back to step 3. {5}
We say "a pointer to ..." since we see a * {3.b}
No more parentheses left so we say "an int". {4,5,6}
That is, "**i is a function returning a pointer to an int**"

Example 6:

int (*i)();

The parenthesization of the above declaration is:

int ((*(i))()); {1} // Note that parentheses are valid tokens
// in a declaration and therefore must be
// be left in place when finding the final
// parenthesization

Applying the rules (see above) to the parenthesized expression can be done as follows:

The innermost parentesis is **(**i**)** {2}
i is the variable name, therefore we say "i is ..." {3}
Move to the next set of parenthesis: **(***(i)**)** {4}
Go back to step 3. {5}
We say "a pointer to" since we see a * {3.b}
Move to the next set of parenthesis: ((*(i))()) {4}
Go back to step 3. {5}
We say "a function returning" since we see a () {3.c}
No more parentheses left so we say "an int". {4,5,6}
That is, "**i is a pointer to a function returning an int**"

This is the pretty much all one needs to know to read and write declarations in C.