

C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

The declarations in the last group specify the return types of the `abs`, `swap`, and `find_largest` functions, but provide no information about their arguments. Leaving the parentheses empty isn't the same as putting the word `void` between them, which indicates that there are no arguments. The empty-parentheses style of function declaration has largely disappeared. It's inferior to the prototype style introduced in C89, since it doesn't allow the compiler to check whether function calls have the right arguments.

If all declarators were as simple as these, C programming would be a snap. Unfortunately, declarators in actual programs often combine the `*`, `[]`, and `()` notations. We've seen examples of such combinations already. We know that

```
int *ap[10];
```

declares an array of 10 pointers to integers. We know that

```
float *fp(float);
```

declares a function that has a `float` argument and returns a pointer to a `float`. And, in Section 17.7, we learned that

```
void (*pf)(int);
```

declares a pointer to a function with an `int` argument and a `void` return type.

Deciphering Complex Declarations

So far, we haven't had too much trouble understanding declarators. But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

This declarator combines `*`, `[]`, and `()`, so it's not obvious whether `x` is a pointer, an array, or a function.

Fortunately, there are two simple rules that will allow us to understand any declaration, no matter how convoluted:

- *Always read declarators from the inside out.* In other words, locate the identifier that's being declared, and start deciphering the declaration from there.