- ***When there's a choice, always favor*** `[]` ***and*** `()` ***over*** `*`. If `*` precedes the identifier and `[]` follows it, the identifier represents an array, not a pointer. Likewise, if `*` precedes the identifier and `()` follows it, the identifier represents a function, not a pointer. (Of course, we can always use parentheses to override the normal priority of `[]` and `()` over `*`.)

Let's apply these rules to our simple examples first. In the declaration

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

the identifier is `ap`. Since `*` precedes `ap` and `[]` follows it, we give preference to `[]`, so `ap` is an *array* of *pointers*. In the declaration

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

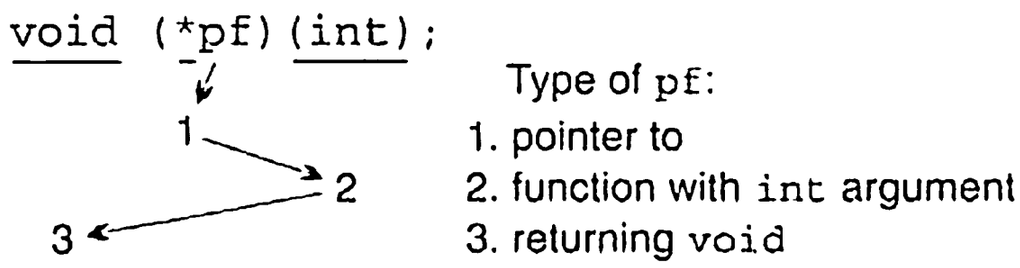
the identifier is `fp`. Since `*` precedes `fp` and `()` follows it, we give preference to `()`, so `fp` is a *function* that returns a *pointer*.

The declaration

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

is a little trickier. Since `*pf` is enclosed in parentheses, `pf` must be a pointer. But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument. The word `void` represents the return type of this function.

As the last example shows, understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



Let's use this zigzagging technique to decipher the declaration given earlier:

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

First, we locate the identifier being declared (`x`). We see that `x` is preceded by `*` and followed by `[]`; since `[]` have priority over `*`, we go right (`x` is an array). Next, we go left to find out the type of the elements in the array (pointers). Next, we go right to find out what kind of data the pointers point to (functions with no arguments). Finally, we go left to see what each function returns (a pointer to an `int`). Graphically, here's what the process looks like: