The symbol table should store essential information for everything that even looks like a variable, recording what kind of thing it is (local variable, global variable, local array, global array, function name, named constant, struct name, whatever) together with any extra details that may be needed (such as FP offset for locals, value for named constants, and sizes for struct names).

If the C-like rule, that the name of an array represents its address, then a few simple additions to the existing scheme will handle everything:

1. New syntax to declare arrays with their size and distinguish them from normal variables, for example

```
local v, array 10 w, array 12 x, y, z
```

or

```
local v, array w : 10, array x : 12, y, z
```

or whatever you like, to be equivalent to the C/C++

```
int v, w[10], x[12], y, z;
```

and of course, there is nothing to stop you from going with the familiar C syntax if you are careful with your design.

2. Make the polish converter use

```
FP-something
```

when a local array name appears, instead of the

```
[FP-something]
```

that it already uses for local variables. Global arrays are similar.

3. Introduce a unary follow-the-pointer operator, `->`, `^`, `#`, whatever you like

4. Realise that a normal array access, like `a[10]` or `b[i*2]` is now equivalent to

```
-> + a 10    or    -> + b * i 2
```

5. Perhaps introduce a new operator `[] x y` that is equivalent to `-> + x y` so that it really looks like an array access.

6. Think about assigning to an array element: what should happen if `set` is immediately followed by `[]` or something along those lines.

Once you have arrays, objects follow naturally. Invent a clear, easy-to-read syntax for defining a struct, perhaps

```
struct item a, b, c
```

to stand for the familiar

```
struct item { int a, b, c; };
```

The autocode system would perhaps note that `item` is a struct name with a size of 3, and automatically define three named constants: `item_a=0`, `item_b=1`, `item_c=2`.

Then perhaps some distinctive syntax such as

```
local struct item x
local array item x
local item x
```

or whatever you like would work exactly the same way as an array declaration, reserving three memory locations for `x`, and just pretending it is an array, so that the C++ `x.a` would be written as `[] x item_a`. Just go with syntax that will make the coding of the autocoder simple for you. Once it is working nicely, you can make the syntax more friendly if you like.