Part 51: Arrays, part 2

In the last part of our compiler writing journey, I realised that I had implemented arrays not exactly right. In this part of our compiler writing journey, I'll try to rectify things.

To start with, I stepped back and thought a bit about arrays and pointers. I realised that an array is similar to a pointer except:

1. You can't use the unadorned array identifier as an rvalue.
2. The size of an array is the size of all of its elements. The size of a pointer does not include the elements of any array that it points to.
3. The address of an array (e.g. `&ary`) doesn't mean anything useful, unlike the address of a pointer (e.g. `&ptr`).

As an example of the first point above, consider:

```
int ary[5];
int *ptr;

int main() {
   ptr= ary;            // OK, put base address of ary into ptr
   ary= ptr;            // Bad, can't change ary's base address
```

And, for those C purists out there, yes I know that point 3 isn't entirely true. But I'm not going to use `&ary` anywhere, so I can get our compiler to reject it, and that means I won't need to implement this functionality!

So, exactly what do we need to change?

- allow a scalar or an an array identifier before a '[' token
- allow an unadorned array identifier but mark it as an rvalue
- add some more errors when we try to do bad things with arrays

That's about it. I've made these changes to the compiler. I hope that they cover all the array issues, but it's likely that I've overlooked something else. If so, we'll revisit again.

Changes to `postfix()`

In the last part, I put in a "band-aid" fix to `postfix()` in `expr.c` , but it's time to go back and fix it properly. We need to allow unadorned array identifiers but mark them as an rvalues. Here are the changes:

```
static struct ASTnode *postfix(void) {
  ...
  int rvalue=0;
  ...
  // An identifier, check that it exists. For arrays, set rvalue to 1.
  if ((varptr = findsymbol(Text)) == NULL)
    fatals("Unknown variable", Text);
  switch(varptr->stype) {
    case S_VARIABLE: break;
    case S_ARRAY: rvalue= 1; break;
    default: fatals("Identifier not a scalar or array variable", Text);
  }

  switch (Token.token) {
    // Post-increment: skip over the token. Also same for post-decrement
  case T_INC:
    if (rvalue == 1)
      fatals("Cannot ++ on rvalue", Text);
  ...
    // Just a variable reference. Ensure any arrays
    // cannot be treated as lvalues.
  default:
    if (varptr->stype == S_ARRAY) {
      n = mkastleaf(A_ADDR, varptr->type, varptr, 0);
      n->rvalue = rvalue;
    } else
      n = mkastleaf(A_IDENT, varptr->type, varptr, 0);
  }
  return (n);
}
```

Now either scalar or array variables can be used unadorned, but arrays can't be lvalues. Also, arrays can't be pre- or post-incremented. We either load the address of the array base, or load the value in the scalar variable.

Changes to `array_access()`

Now we need to modify `array_access()` in `expr.c` to allow pointers to be used with '[' ']' indexing. Here are the changes:

```
static struct ASTnode *array_access(void) {
  struct ASTnode *left, *right;
  struct symtable *aryptr;

  // Check that the identifier has been defined as an array or a pointer.
  if ((aryptr = findsymbol(Text)) == NULL)
    fatals("Undeclared variable", Text);
  if (aryptr->stype != S_ARRAY &&
        (aryptr->stype == S_VARIABLE && !ptrtype(aryptr->type)))
    fatals("Not an array or pointer", Text);

  // Make a leaf node for it that points at the base of
  // the array, or loads the pointer's value as an rvalue
  if (aryptr->stype == S_ARRAY)
    left = mkastleaf(A_ADDR, aryptr->type, aryptr, 0);
  else {
    left = mkastleaf(A_IDENT, aryptr->type, aryptr, 0);
    left->rvalue= 1;
  }
  ...
}
```

We now check that the symbol exists and is either an array or a scalar variable of pointer type. Once this is OK, we either load the address of the array base, or load the value in the pointer variable.

Testing the Code Changes

I won't go through all the tests; instead I'll summarise them:

- `tests/input124.c` checks that `ary++` can't be done on an array.
- `tests/input125.c` checks that we can assign `ptr= ary` and then access the array though the pointer.
- `tests/input126.c` checks that we can't do `&ary` .

- `tests/input127.c` calls a function with `fred(ary)` and ensures that we can receive it as a pointer parameter.

Conclusion and What's Next

Well, I was worried that I'd had to rewrite a whole pile of code to get arrays to work correctly. As it stood, the code was nearly right but just needed some more tweaking to cover all the functionality that we needed.

In the next part of our compiler writing journey, we will go back to mopping up. Next step