Part 52: Pointers, part 2

In this part of our compiler writing journey, I started with a pointer issue that needed to fix, and I ended up restructuring about half of `expr.c` and changing the API to another quarter of the functions in the compiler. So this is a big step in terms of number of lines touched, but not a big step in terms of fixes or improvements.

The Problem

We'll start with the problem that caused all of this. When running the compiler's source code through itself, I realised that I couldn't parse a chain of pointers, e.g. something like the expression:

```
  ptr->next->next->next
```

The reason for this is that `primary()` is called and gets the value of the identifier at the beginning of the expression. If it sees a following postfix operator, it then calls `postfix()` to deal with it. `postfix()` deals with, for example, one `->` operator and returns. And that's it. There is no loop to follow a chain of `->` operators.

Even worse, `primary()` looks for a single identifier. This means that it won't parse the following, either:

```
  ptrarray[4]->next     OR
```

```
    unionvar.member->next
```

because neither of these are single identifers before the `->` operator.

How Did This Happen?

This happened because of the rapid prototyping nature of our development. I only add functionality one small step at a time, and I don't usually look too far ahead in terms of future needs. So, now and then, we have to undo what has been written to make it more general and flexible.

How to Fix It?

If we look at the BNF Grammar for C, we see this:

```
primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression '->' IDENTIFIER
        | postfix_expression '++'
        | postfix_expression '--'
        ;
```

In other words, we have things backwards. `postfix` should call `primary()` to get an AST node that represents the identifier. Then, we can loop looking for any postfix tokens, parse them and add new AST parent nodes on to the identifier node that we received back from `primary()` .

It all sounds nice and simple except for one thing. The current `primary()` doesn't build an AST node; it only parses the identifier and leaves it in `Text` . It's the job of `postfix()` to build the AST node or AST tree for the identifier plus any postfix operations.

At the same time, the AST node structure in `defs.h` only knows about the primitive type:

```
// Abstract Syntax Tree structure
struct ASTnode {
  int op;             // "Operation" to be performed on this tree
  int type;           // Type of any expression this tree generates
  int rvalue;         // NOTE: no ctype
  ...
};
```

The reason for this is that we only recently added structs and unions. This, and the fact that `postfix()` did most of the parsing work meant that we haven't needed to store a pointer to the struct or union symbol for an identifier which is a struct or union.

So, to fix things, we need to:

1. Add in a `ctype` pointer to `struct ASTnode` so that the full type is stored in each AST node.
2. Find and fix all the functions that build AST nodes, and all the calls to these function, so that the `ctype` of a node is stored.
3. Move `primary()` up near the top of `expr.c` and get it to build an AST node.
4. Get `postfix()` to call `primary()` to get the unadorned AST node for an identifier (A_IDENT).
5. Get `postfix()` to loop while there are postfix operators to process.

That's a lot and, as the AST node calls are sprinkled everywhere, every single source file in the compiler will need to be touched. Sigh.

Changes to the AST Node Functions

I'm not going to bore you with all the details, but we can start with the change to the AST node structure in `defs.h`, and the main function in `tree.c` that builds an AST node:

```
// Abstract Syntax Tree structure
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  int type;                     // Type of any expression this tree generates
  struct symtable *ctype;       // If struct/union, ptr to that type
  ...
};

// Build and return a generic AST node
struct ASTnode *mkastnode(int op, int type,
                          struct symtable *ctype, ...) {
  ...
```

```
    // Copy in the field values and return it
    n->op = op;
    n->type = type;
    n->ctype = ctype;
    ...
  }
```

There are also changes to `mkastleaf()` and `mkastunary()`: they now receive a `ctype` and call `mkastnode()` with this argument.

In the compiler there are about 40 calls to these three functions, so I'm not going to go through each and every one. For most of them, there is a primitive `type` and `ctype` pointer available. Some calls set the AST node type to P_INT and thus the `ctype` is NULL. Some calls set the AST node type to P_NONE and, again, the `ctype` is NULL.

Changes to `modify_type()`

The `modify_type()` is used to determine if an AST node's type is compatible with another type and, if necessary, to widen the node to match the other type. It calls `mkastunary()` and thus we also need to provide it with a `ctype` argument. I've done this and, as a consequence, the six calls to `modify_type()` have had to be modified to pass in the `ctype` of the type which we are comparing the AST node against.

Changes to `expr.c`

Now we get to the meat of the changes, the restucturing of `primary()` and `postfix()`. I've already outlined what we have to do above. As with much of what we've done, there are a few wrinkles along the way to iron out.

Changes to `postfix()`

`postfix()` actually looks much cleaner now:

```
// Parse a postfix expression and return
// an AST node representing it. The
// identifier is already in Text.
static struct ASTnode *postfix(void) {
  struct ASTnode *n;

  // Get the primary expression
  n = primary();
  // Loop until there are no more postfix operators
  while (1) {
    switch (Token.token) {
    ...
    default:
      return (n);
    }
  }
```

We now call `primary()` to get an identifier or a constant. Then we loop applying any postfix operators to the AST node we received from `primary()`. We call out to helper functions like `array_access()` and `member_access()` for `[..]`, `.` and `->` operators.

We do post-increment and post-decrement here. Now that there is a loop, we have to check that we don't try to do these operations more than once. We also check that the AST we received from `primary()` is an lvalue and not an rvalue, as we need an address in memory to increment or decrement.

A New Function, `paren_expression()`

I realised that the new `primary()` function was getting a bit too big, so I split some of its code off into a new function, `paren_expression()`. This parses expressions that are enclosed in `(..)`: casts and ordinary parenthesised expressions. The code is nearly identical to the old code, so I won't go into it here. It returns an AST node with the tree that represents either a cast expression or a parenthesised expression.

Changes to `primary()`

This is where the biggest change has occurred. Firstly, here are the tokens it looks for:

- 'static', 'extern' which it complains about, because we can only be parsing expressions in a local context.
- 'sizeof()'
- integer and string literals
- identifiers: these could be known types (e.g. 'int'), names of enums, names of typedefs, function names, array names and/or scalar variable names. This section is the biggest part of `primary()` and, on reflection, perhaps I should make this into its own function.
- `(..)` which is where `paren_expression()` gets called.

Looking at the code, `primary()` now builds AST nodes for each of the above to return to `postfix()`. This used to be done in `postfix` but I now do it in `primary()`.

Changes to `member_access()`

With the previous `member_access()`, the global `Text` variable still held the identifier, and `member_access()` built the AST node to represent the struct/union identifier.

In the current `member_access()`, we receive the AST node for the struct/union identifier, and this could be an array element or a member of another struct/union.

So the code is different in that we don't build the leaf AST node for the original identifier anymore. We still build AST nodes to add on the offset from the base and dereference the pointer to the member.

One other difference is this code:

```
// Check that the left AST tree is a struct or union.
// If so, change it from an A_IDENT to an A_ADDR so that
// we get the base address, not the value at this address.
if (!withpointer) {
  if (left->type == P_STRUCT || left->type == P_UNION)
    left->op = A_ADDR;
  else
    fatal("Expression is not a struct/union");
}
```

Consider the expression `foo.bar`. `foo` is the name of a struct, for example, and `bar` is a member of that struct`.

In `primary()` we will have created an A_IDENT AST node for `foo`, because we can't tell if this is a scalar variable (e.g. `int foo`) or a structure (e.g. `struct fred foo`). Now that we know it's a struct or a union, we need the base address of the struct and not the value at the base address. So, the code converts the A_IDENT AST node operation into an A_ADDR operation on the identifier.

Testing the Code

I think I spent about two hours running through our hundred plus regression tests, finding things I'd missed and fixing them up. It certainly felt good to get through all the tests again.

`tests/input128.c` now checks that we can follow a chain of pointers, which was the whole point of this exercise:

```
struct foo {
  int val;
  struct foo *next;
};

struct foo head, mid, tail;

int main() {
  struct foo *ptr;
  tail.val= 20; tail.next= NULL;
  mid.val= 15; mid.next= &tail;
  head.val= 10; head.next= &mid;

  ptr= &head;
  printf("%d %d\n", head.val, ptr->val);
  printf("%d %d\n", mid.val, ptr->next->val);
  printf("%d %d\n", tail.val, ptr->next->next->val);
  return(0);
}
```

And `tests/input129.c` checks that we can't post-increment twice in a row.

One Other Change: `Linestart`

There is one more change that I made to the compiler as part of our effort to get it to self-compile.

The scanner was looking for a '#' token. When it saw this token, it assumed that we had hit a C pre-processor line and it parsed this line. Unfortunately, I hadn't tied the scanner down to looking in the first column of each line. So, when our compiler hit this source code line:

```
  while (c == '#') {
```

it got upset that the ')' '{' were not a C pre-processor line.

We now have a `Linestart` variable which flags if the scanner is at the first column of a new line or not. The main function which is modified is `next()` in `scan.c`. I think the changes are a bit ugly but they work; I should come back sometime and see if I can clean this up a bit. Anyway, we only expect C pre-processor lines when we see a '#' in column 1.

Conclusion and What's Next

In the next part of our compiler writing journey, I'll go back to feeding the compiler source code to itself, see what errors pop and up choose one or more to fix. Next step