Part 32: Accessing Members in a Struct

This part of our compiler writing journey turned out to be quite simple. I've added the '.' and '->' tokens to our language, and I've implemented one level of member access to global struct variables.

I'll give our test program, `tests/input58.c`, here so that you can see the language features that I've implemented:

```
int printf(char *fmt);

struct fred {                   // Struct declaration, done last time
  int x;
  char y;
  long z;
};

struct fred var2;               // Variable declaration, done last time
struct fred *varptr;            // Pointer variable declaration, done last tim

int main() {
  long result;

  var2.x= 12;   printf("%d\n", var2.x);         // Member access as lvalue, ne
  var2.y= 'c';  printf("%d\n", var2.y);
  var2.z= 4005; printf("%d\n", var2.z);

  result= var2.x + var2.y + var2.z;             // Member access as rvalue, ne
  printf("%d\n", result);
```

```
    varptr= &var2;                                  // Old behaviour
    result= varptr->x + varptr->y + varptr->z;      // Member access through point
    printf("%d\n", result);
    return(0);
  }
```

The New Tokens

We have two new tokens, T_DOT and T_ARROW, to match the '.' and '->' elements in the input. As always, I won't give the code in `scan.c` to identify these.

Parsing the Member References

This turned out to be very similar to our existing array element accessing code. Let's look at the similarities and the differences. With this code:

```
int x[5];
int y;
...
y= x[3];
```

we get the base address of the `x` array, multiply 3 by the size of the `int` type in bytes (e.g.3*4 is 12), add that to the base address, and treat this as the address of the `int` that we want to access. Then we dereference this address to get the value at that array position.

Accessing a struct member is similar:

```
struct fred { int x; char y; long z; };
struct fred var2;
char y;
...
y= var2.y;
```

We get the base address of `var2`. We get the offset of the `y` member in the `fred` struct, add this to the the base address, and treat this as the address of the `char` that we want to access. Then we dereference this address to get the value there.

Postfix Operators

T_DOT and T_ARROW are postfix operators, like the '[' of an array reference, as they come after an identifier's name. So it makes sense to add their parsing in the existing `postfix()` function in `expr.c`:

```
static struct ASTnode *postfix(void) {
  ...
    // Access into a struct or union
  if (Token.token == T_DOT)
    return (member_access(0));
  if (Token.token == T_ARROW)
    return (member_access(1));
  ...
}
```

The argument to the new `member_access()` function in `expr.c` indicates if we are accessing a member through a pointer or directly. Now let's look at the new `member_access()` in stages.

```
// Parse the member reference of a struct (or union, soon)
// and return an AST tree for it. If withpointer is true,
// the access is through a pointer to the member.
static struct ASTnode *member_access(int withpointer) {
  struct ASTnode *left, *right;
  struct symtable *compvar;
  struct symtable *typeptr;
  struct symtable *m;

  // Check that the identifier has been declared as a struct (or a union, late
  // or a struct/union pointer
  if ((compvar = findsymbol(Text)) == NULL)
    fatals("Undeclared variable", Text);
  if (withpointer && compvar->type != pointer_to(P_STRUCT))
    fatals("Undeclared variable", Text);
  if (!withpointer && compvar->type != P_STRUCT)
    fatals("Undeclared variable", Text);
```

First, some error checking. I know I will have to add checking for unions here, so I'm not going to refactor the code just yet.

```
  // If a pointer to a struct, get the pointer's value.
  // Otherwise, make a leaf node that points at the base
```

```
// Either way, it's an rvalue
if (withpointer) {
  left = mkastleaf(A_IDENT, pointer_to(P_STRUCT), compvar, 0);
} else
  left = mkastleaf(A_ADDR, compvar->type, compvar, 0);
left->rvalue = 1;
```

At this point we need to get the base address of the composite variable. If we are given a pointer, we simply load the pointer's value by making an A_IDENT AST node. Otherwise, the identifier *is* the struct or union, so we had better get its address with an A_ADDR AST node.

This node can't be an lvalue, i.e. we can't say `var2. = 5`. It has to be an rvalue.

```
// Get the details of the composite type
typeptr = compvar->ctype;

// Skip the '.' or '->' token and get the member's name
scan(&Token);
ident();
```

We get a pointer to the composite type so that we can walk the list of members in the type, and we get the member's name after the '.' or '->' (and confirm that it is an identifier).

```
// Find the matching member's name in the type
// Die if we can't find it
for (m = typeptr->member; m != NULL; m = m->next)
  if (!strcmp(m->name, Text))
    break;

if (m == NULL)
  fatals("No member found in struct/union: ", Text);
```

We walk the member's list to find the matching member's name.

```
// Build an A_INTLIT node with the offset
right = mkastleaf(A_INTLIT, P_INT, NULL, m->posn);

// Add the member's offset to the base of the struct and
// dereference it. Still an lvalue at this point
left = mkastnode(A_ADD, pointer_to(m->type), left, NULL, right, NULL, 0);
left = mkastunary(A_DEREF, m->type, left, NULL, 0);
```

```
    return (left);
  }
```

The member's offset in bytes is stored in `m->posn` so we make an A_INTLIT node with this value, and A_ADD it to the base address stored in `left`. At this point we have an address of the member, so we dereference it (A_DEREF) to get access to the member's value. At this point, this is still an lvalue; this allows us to do both `5 + var2.x` and `var2.x= 6`.

Running Our Test Code

The output of `tests/input58.c` is, unsurprisingly:

```
12
99
4005
4116
4116
```

Let's have a look at some of the assembly output:

```
                                        # var2.y= 'c';
        movq    $99, %r10               # Load 'c' into %r10
        leaq    var2(%rip), %r11        # Get base address of var2 into
%r11
        movq    $4, %r12
        addq    %r11, %r12              # Add 4 to this base address
        movb    %r10b, (%r12)           # Write 'c' into this new address

                                        # printf("%d\n", var2.z);
        leaq    var2(%rip), %r10        # Get base address of var2 into
%r11
        movq    $4, %r11
        addq    %r10, %r11              # Add 4 to this base address
        movzbq  (%r11), %r11            # Load byte value from this address
into %r11
        movq    %r11, %rsi              # Copy it into %rsi
        leaq    L4(%rip), %r10
        movq    %r10, %rdi
        call    printf@PLT              # and call printf()
```

Conclusion and What's Next

Well, this was a nice pleasant surprise to get structs to work this easily! I'm sure the future parts of our journey will make up for it. I also know that our compiler as it stands still is pretty limited. For example, it can't do this:

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

struct foo *listhead;
struct foo *l;

int main() {
  ...
  l= listhead->next->next;
```

as this requires following two pointer levels. The existing code can only follow one pointer level. We will have to fix this later.

It is probably also a good time to indicate that we will have to spend a lot of time getting the compiler to "do it right". I've been adding functionality, but only enough to get one specific feature to work. At some point these specific features will have to be made more general. So there will be a "mop up" stage in this journey.

Now that we have structs mostly working, in the next part of our compiler writing journey, I will try to add unions. Next step