Part 33: Implementing Unions and Member Access

Unions also turned out to be easy to implement for one reason: they are like structs except that all members of a union are located at offset zero from the base of the union. Also, the grammar of a union declaration is the same as a struct except for the "union" keyword.

This means that we can re-use and modify the existing structs code to deal with unions.

A New Keyword: "union"

I've added the "union" keyword and the T_UNION token to the scanner in `scan.c`. As always, I'll omit the code that does the scanning.

The Union Symbol List

As with structs, there is a singly-linked list to store unions (in `data.h`):

```
extern_ struct symtable *Unionhead, *Uniontail;   // List of struct types
```

In `sym.c`, I've also written `addunion()` and `findunion()` functions to add a new union type node to the list and to search for a union type with a given name on the list.

> I'm considering merging the struct and union lists into a single composite type list, but I haven't done it yet. I'll probably do it when I get around to some more refactoring.

Parsing Union Declarations

We are going to modify the existing struct parsing code in `decl.c` to parse both structs and unions. I'll only give the changes to the functions, not the whole functions.

In `parse_type()`, we now scan the T_UNION token and call the function to parse both struct and union types:

```
  case T_STRUCT:
    type = P_STRUCT;
    *ctype = composite_declaration(P_STRUCT);
    break;
  case T_UNION:
    type = P_UNION;
    *ctype = composite_declaration(P_UNION);
    break;
```

This function `composite_declaration()` was called `struct_declaration()` in the last part of our journey. It now takes the type that we are parsing.

The `composite_declaration()` Function

Here are the changes:

```
// Parse composite type declarations: structs or unions.
// Either find an existing struct/union declaration, or build
// a struct/union symbol table entry and return its pointer.
static struct symtable *composite_declaration(int type) {
  ...
  // Find any matching composite type
  if (type == P_STRUCT)
    ctype = findstruct(Text);
  else
    ctype = findunion(Text);
  ...
  // Build the composite type and skip the left brace
  if (type == P_STRUCT)
    ctype = addstruct(Text, P_STRUCT, NULL, 0, 0);
  else
```

```
    ctype = addunion(Text, P_UNION, NULL, 0, 0);
  ...
  // Set the position of each successive member in the composite type
  // Unions are easy. For structs, align the member and find the next free byt
  for (m = m->next; m != NULL; m = m->next) {
    // Set the offset for this member
    if (type == P_STRUCT)
      m->posn = genalign(m->type, offset, 1);
    else
      m->posn = 0;

    // Get the offset of the next free byte after this member
    offset += typesize(m->type, m->ctype);
  }
  ...
  return (ctype);
}
```

That's it. We simply change the symbol table list we are working on, and always set the member offset to zero for unions. This is why I think it would be worth merging the struct and union type lists into a single list.

Parsing Union Expressions

As with the union declarations, we can reuse the code that deals with structs in expressions. In fact, there are very few changes to make in `expr.c`.

```
// Parse the member reference of a struct or union
// 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) {
  ...
  if (withpointer && compvar->type != pointer_to(P_STRUCT)
      && compvar->type != pointer_to(P_UNION))
    fatals("Undeclared variable", Text);
  if (!withpointer && compvar->type != P_STRUCT && compvar->type != P_UNION)
    fatals("Undeclared variable", Text);
```

Again, that's it. The rest of the code was generic enough that we can use it for unions unmodified. And I think there was only one other major change, which was to a function in `types.c`:

```
// Given a type and a composite type pointer, return
// the size of this type in bytes
int typesize(int type, struct symtable *ctype) {
  if (type == P_STRUCT || type == P_UNION)
    return (ctype->size);
  return (genprimsize(type));
}
```

Testing the Union Code

Here's our test program, `test/input62.c` :

```
int printf(char *fmt);

union fred {
  char w;
  int  x;
  int  y;
  long z;
};

union fred var1;
union fred *varptr;

int main() {
  var1.x= 65; printf("%d\n", var1.x);
  var1.x= 66; printf("%d\n", var1.x); printf("%d\n", var1.y);
  printf("The next two depend on the endian of the platform\n");
  printf("%d\n", var1.w); printf("%d\n", var1.z);

  varptr= &var1; varptr->x= 67;
  printf("%d\n", varptr->x); printf("%d\n", varptr->y);

  return(0);
}
```

This tests that all four members in the union are at the same location, so that a change to one member is seen as the same change to all members. We also check that pointer access into a union also works.

Conclusion and What's Next

This was another nice and easy part of our compiler writing journey. In the next part of our compiler writing journey, we will add enums. Next step