rzaharia Updated all readme files to contain links to the next step 2 years ago

522 lines (413 loc) · 16.8 KB

Preview Code Blame Raw

Part 31: Implementing Structs, Part 1

In this part of our compiler writing journey, I've begun the process of implementing structs into the language. Even though these are not yet functional, I've made substantial changes to the code just to get to the point where we can declare structs, and global variables of struct type.

Symbol Table Changes

As I mentioned in the last part, we need to change the symbol table structure to include a pointer to a composite type node, when the symbol is of this type. We also added a `next` pointer to support linked lists, and a `member` pointer. The `member` pointer of a function node holds the function's parameter list. We will use the `member` node for structs to hold the struct's member fields.

So, we now have:

```
struct symtable {
  char *name;                   // Name of a symbol
  int type;                     // Primitive type for the symbol
  struct symtable *ctype;       // If needed, pointer to the composite type
  ...
  struct symtable *next;        // Next symbol in one list
  struct symtable *member;      // First member of a function, struct,
};                              // union or enum
```

We also have two new lists for symbols in `data.h` :

```
// Symbol table lists
struct symtable *Globhead, *Globtail;     // Global variables and functions
struct symtable *Loclhead, *Locltail;     // Local variables
struct symtable *Parmhead, *Parmtail;     // Local parameters
struct symtable *Membhead, *Membtail;     // Temp list of struct/union members
struct symtable *Structhead, *Structtail; // List of struct types
```

Changes to `sym.c`

Throughout `sym.c` , and elsewhere in the code, we used to only receive the `int type` argument to determine the type of something. This isn't enough now that we have composite types: the P_STRUCT integer value tells us that something is a struct, not which one.

Therefore, many functions now receive an `int type` argument and also a `struct symtable *ctype` argument. When `type` is P_STRUCT, `ctype` points at the node which defines this particular struct type.

In `sym.c` , all the `addXX()` functions have been modified to have this extra argument. There is also a new `addmemb()` function and a new `addstruct()` function to add nodes to these two new lists. They function identically to the other `addXX()` functions but just on a different list. I will come back to these functions later.

A New Token

We have our first new token, P_STRUCT, in quite a while. It goes with the matching `struct` keyword. I'll omit the changes to `scan.c` as they are minor.

Parsing Structs in our Grammar

There are a bunch of places where we need to parse the `struct` keyword:

- the definition of a named struct
- the definition of an unnamed struct followed by a variable of this type
- the definition of a struct within another struct or union
- the definition of a variable of a previously defined struct type

At first, I wasn't sure where to fit in the parsing of structs. Should I assume that we are parsing a new struct definition, but bail out when I see a variable identifier, or assume a variable declaration?

In the end, I realised that, after seeing `struct <identifier>`, I had to assume that this was just the naming of a type, just as `int` is the naming of the `int` type. We had to parse the next token to determine otherwise.

Therefore, I modified `parse_type()` in `decl.c` to parse both scalar types (e.g. `int`) and composite types (e.g. `struct foo`). And now that it can return a composite type, I had to find a way to return the pointer to the node that defines this composite type:

```
// Parse the current token and return
// a primitive type enum value and a pointer
// to any composite type.
// Also scan in the next token
int parse_type(struct symtable **ctype) {
  int type;
  switch (Token.token) {
    ...         // Existing code for T_VOID, T_CHAR etc.
    case T_STRUCT:
      type = P_STRUCT;
      *ctype = struct_declaration();
      break;
    ...
```

We call `struct_declaration()` to either look up an existing struct type or to parse the declaration of the new struct type.

Refactoring The Parsing of a Variable List

In our old code, there was a function called `param_declaration()` that parsed a list of parameters separated by commas, e.g.

```
int fred(int x, char y, long z);
```

such as you would find as the parameter list for a function declaration. Well, a struct and union declaration also has a list of variables, except that they are separated by semicolons and surrounded by curly brackets, e.g.

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

It makes sense to refactor the function to parse both lists. It now is passed two tokens: the separating token, e.g. T_SEMI and the ending token, e.g. T_RBRACE. Thus, we can use it to parse both styles of lists.

```
// Parse a list of variables.
// Add them as symbols to one of the symbol table lists, and return the
// number of variables. If funcsym is not NULL, there is an existing function
// prototype, so compare each variable's type against this prototype.
static int var_declaration_list(struct symtable *funcsym, int class,
                                int separate_token, int end_token) {
    ...
    // Get the type and identifier
    type = parse_type(&ctype);
    ...
    // Add a new parameter to the right symbol table list, based on the class
    var_declaration(type, ctype, class);
}
```

When we are parsing function parameter lists, we call:

```
    var_declaration_list(oldfuncsym, C_PARAM, T_COMMA, T_RPAREN);
```

When we are parsing struct member lists, we call:

```
    var_declaration_list(NULL, C_MEMBER, T_SEMI, T_RBRACE);
```

Also note that the call to `var_declaration()` now is given the type of the variable, the composite type pointer (if it is a struct or union), and the variable's class.

Now we can parse the lists of members of a struct. So let's see how we parse the whole struct.

The `struct_declaration()` Function

Let's take this in stages.

```
static struct symtable *struct_declaration(void) {
  struct symtable *ctype = NULL;
  struct symtable *m;
  int offset;
```

```
  // Skip the struct keyword
  scan(&Token);

  // See if there is a following struct name
  if (Token.token == T_IDENT) {
    // Find any matching composite type
    ctype = findstruct(Text);
    scan(&Token);
  }
```

At this point we have seen `struct` possibly followed by an identifier. If this is an existing struct type, `ctype` now points at the existing type node. Otherwise, `ctype` is NULL.

```
  // If the next token isn't an LBRACE , this is
  // the usage of an existing struct type.
  // Return the pointer to the type.
  if (Token.token != T_LBRACE) {
    if (ctype == NULL)
      fatals("unknown struct type", Text);
    return (ctype);
  }
```

We didn't see a '{', so this has to be just the naming of an existing type. `ctype` cannot be NULL, so we check that first and then simply return the pointer to this existing struct type. This is going to go back to `parse_type()` when we did:

```
      type = P_STRUCT; *ctype = struct_declaration();
```

But, assuming we didn't return, we must have found a '{', and this signals the definition of a struct type. Let's go on...

```
  // Ensure this struct type hasn't been
  // previously defined
  if (ctype)
    fatals("previously defined struct", Text);

  // Build the struct node and skip the left brace
  ctype = addstruct(Text, P_STRUCT, NULL, 0, 0);
  scan(&Token);
```

We can't declare a struct with the same name twice, so prevent this. Then build the beginnings of the new struct type as a node in the symbol table. All we have so far is its name and that it is of P_STRUCT type.

```
// Scan in the list of members and attach
// to the struct type's node
var_declaration_list(NULL, C_MEMBER, T_SEMI, T_RBRACE);
rbrace();
```

This parses the list of members. For each one, a new symbol node is appended to the list that `Membhead` and `Membtail` point to. This list is only temporary, because the next lines of code move the member list into the new struct type node:

```
ctype->member = Membhead;
Membhead = Membtail = NULL;
```

We now have a struct type node with a name, and the list of members in the struct. What's left to do? Well, we now need to determine:

- the overall size of the struct, and
- the offset of each member from the base of the struct

Some of this is very hardware-specific due to the alignment of scalar values in memory. So I'll give the code as it stands now, and then follow the function call structure later.

```
// Set the offset of the initial member
// and find the first free byte after it
m = ctype->member;
m->posn = 0;
offset = typesize(m->type, m->ctype);
```

We now have a new function, `typesize()` to get the size of any type: scalar, pointer or composite. The first member's position is set to zero, and we use its size to determine the first possible byte where the next member could be stored. But now we need to worry about alignment.

As an example, on a 32-bit architecture where 4-byte scalar values have to be aligned on a 4-byte boundary:

```
struct {
  char x;               // At offset 0
  int y;                // At offset 4, not 1
};
```

So here is the code to calculate the offset of each successive member:

```c
  // Set the position of each successive member in the struct
  for (m = m->next; m != NULL; m = m->next) {
    // Set the offset for this member
    m->posn = genalign(m->type, offset, 1);

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

We have a new function, `genalign()` that takes a current offset and the type that we need to align, and returns the first offset that suits the alignment of this type. For example, `genalign(P_INT, 3, 1)` might return 4 if P_INTs have to be 4-aligned. I'll discuss the final 1 argument soon.

So, `genalign()` works out the correct alignment for this member, and then we add on this member's size to get the next free (unaligned) position which is available for the next member.

Once we have done the above for all the members in the list, the `offset` is the size in bytes of the overall struct. So:

```c
  // Set the overall size of the struct
  ctype->size = offset;
  return (ctype);
}
```

The `typesize()` Function

It's time to follow all the new functions to see what they do and how they do it. We'll start with `typesize()` in `types.c`:

```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)
    return(ctype->size);
  return(genprimsize(type));
}
```

If the type is a struct, return the size from the struct's type node. Otherwise it's a scalar or pointer type, so ask `genprimsize()` (which calls the hardware-specific `cgprimsize()`) to get the type's size. Nice and easy.

The `genalign()` and `cgalign()` Functions

Now we get into some not so nice code. Given a type and an existing unaligned offset, we need to know which is the next aligned position to place a value of the given type.

I also was worried that we might need to do this on the stack, which grows downwards not upwards. So there is a third argument to the function: the *direction* in which we need to find the next aligned position.

Also, the knowledge of alignment is hardware specific, so:

```
int genalign(int type, int offset, int direction) {
  return (cgalign(type, offset, direction));
}
```

and we turn our attention to `cgalign()` in `cg.c` :

```
// Given a scalar type, an existing memory offset
// (which hasn't been allocated to anything yet)
// and a direction (1 is up, -1 is down), calculate
// and return a suitably aligned memory offset
// for this scalar type. This could be the original
// offset, or it could be above/below the original
int cgalign(int type, int offset, int direction) {
  int alignment;

  // We don't need to do this on x86-64, but let's
  // align chars on any offset and align ints/pointers
  // on a 4-byte alignment
  switch(type) {
    case P_CHAR: return (offset);
    case P_INT:
    case P_LONG: break;
    default:     fatald("Bad type in calc_aligned_offset:", type);
  }

  // Here we have an int or a long. Align it on a 4-byte offset
  // I put the generic code here so it can be reused elsewhere.
  alignment= 4;
  offset = (offset + direction * (alignment-1)) & ~(alignment-1);
```

```
    return (offset);
  }
```

Firstly, yes I know that we don't have to worry about alignment in the x86-64 architecture. But I thought we should go through the exercise of dealing with alignment, so there is an example of it being done which can be borrowed for other backends that may be written.

The code returns the given offset for `char` types, as they can be stored at any alignment. But we enforce a 4-byte alignment on `int` s and `long` s.

Let's break down the big offset expression. The first `alignment-1` turns `offset` 0 into 3, 1 into 4, 2 into 5 etc. Then, at the end we AND this with the inverse of 3, i.e. ...111111100 to discard the last two bits and lower the value back down to the correct alignment.

Thus:

Offset	Add Value	New Offset
0	3	0
1	4	4
2	5	4
3	6	4
4	7	4
5	8	8
6	9	8
7	10	8

An offset of 0 stays at zero, but values 1 to 3 are pushed up to 4. 4 stays aligned at 4, but 5 to 7 get pushed up to 8.

Now the magic. A `direction` of 1 does everything that we have seen so far. A `direction` of -1 sends the offset in the opposite direction to ensure that the value's "high end" won't hit what's above it:

Offset	Add Value	New Offset
0	-3	-4
-1	-4	-4

Offset	Add Value	New Offset
-2	-5	-8
-3	-6	-8
-4	-7	-8
-5	-8	-8
-6	-9	-12
-7	-10	-12

Creating a Global Struct Variable

So now we can parse a struct type, and declare a global variable to this type. Now let's modify the code to allocate the memory space for a global variable:

```
// Generate a global symbol but not functions
void cgglobsym(struct symtable *node) {
  int size;

  if (node == NULL) return;
  if (node->stype == S_FUNCTION) return;

  // Get the size of the type
  size = typesize(node->type, node->ctype);

  // Generate the global identity and the label
  cgdataseg();
  fprintf(Outfile, "\t.globl\t%s\n", node->name);
  fprintf(Outfile, "%s:", node->name);

  // Generate the space for this type
  switch (size) {
    case 1: fprintf(Outfile, "\t.byte\t0\n"); break;
    case 4: fprintf(Outfile, "\t.long\t0\n"); break;
    case 8: fprintf(Outfile, "\t.quad\t0\n"); break;
    default:
      for (int i=0; i < size; i++)
        fprintf(Outfile, "\t.byte\t0\n");
  }
}
```

Trying The Changes Out

We don't have any new functionality apart from parsing structs, storing new nodes in the symbol table and generating storage for global struct variables.

I have this test program, `z.c` :

```
struct fred { int x; char y; long z; };
struct foo { char y; long z; } var1;
struct { int x; };
struct fred var2;
```

which should create two global variables `var1` and `var2` . We create two named struct types, `fred` and `foo` , and one unnamed struct. The third struct should cause an error (or at least a warning) because there is no variable associated with the struct, so the struct itself is useless.

I added some test code to print out the member offsets and struct sizes for the above structs, and this is the result:

```
Offset for fred.x is 0
Offset for fred.y is 4
Offset for fred.z is 8
Size of struct fred is 13

Offset for foo.y is 0
Offset for foo.z is 4
Size of struct foo is 9

Offset for struct.x is 0
Size of struct struct is 4
```

Finally, when I do `./cwj -S z.c` , I get this assembly output:

```
        .globl  var1
var1:   .byte   0       // Nine bytes
        ...

        .globl  var2    // Thirteen bytes
var2:   .byte   0
        ...
```

Conclusion and What's Next

In this part I've had to change a lot of the existing code from dealing with just an `int type` to dealing with an `int type; struct symtable *ctype` pair. I'm sure I'll have to do this in more places.

We've added the parsing of struct definitions and also declarations of struct variables, and we can generate the space for global struct variables. At the moment, we can't use the struct variables that we have created. But it's a good start. I also haven't even tried to deal with local struct variables, because that involves the stack and I'm sure that will be complicated.

In the next part of our compiler writing journey, I will try to add the code to parse the '.' token so that we can access members in a struct variable. [Next step]