Part 40: Global Variable Initialisation

In the previous part of our compiler writing journey, I started the groundwork to add variable declarations to our language. I've been able to implement this for global scalar and array variables in this part of our compiler writing journey.

At the same time, I realised that I hadn't designed the symbol table structure to properly deal with the size of a variable and the number of elements in an array variable. So half of this part is going to be a rewrite of some of the code that deals with the symbol table.

A Quick Recap for Global Variable Assignments

As a quick recap, below are a set of example global variable assignments that I want to support:

```
int x= 2;
char y= 'a';
char *str= "Hello world";
int a[10];
char b[]= { 'q', 'w', 'e', 'r', 't', 'y' };
char c[10]= { 'q', 'w', 'e', 'r', 't', 'y' };   // Zero padded
char *d[]= { "apple", "banana", "peach", "pear" };
```

I'm not going to deal with initialisation of global structs or unions. Also, for now, I'm not going to deal with putting NULL into `char *` variables. I'll come back to that later, if we need it.

Where We Go To

In the last part of the journey, I'd written this in `decl.c`:

```
static struct symtable *symbol_declaration(...) {
  ...
  // The array or scalar variable is being initialised
  if (Token.token == T_ASSIGN) {
    ...
        // Array initialisation
    if (stype == S_ARRAY)
      array_initialisation(sym, type, ctype, class);
    else {
      fatal("Scalar variable initialisation not done yet");
      // Variable initialisation
      // if (class== C_LOCAL)
      // Local variable, parse the expression
      // expr= binexpr(0);
      // else write more code!
    }
  }
  ...
}
```

i.e. I knew where to put the code but I didn't know what code to write. First up, we need to parse some literal values...

Scalar Variable Initialisation

We are going to need to parse integer and string literals, as these are the only things which we can assign to global variables. We need to ensure that the type of each literal is compatible with the variable type that we are assigning. To this end, there's a new function in `decl.c`:

```
// Given a type, check that the latest token is a literal
// of that type. If an integer literal, return this value.
// If a string literal, return the label number of the string.
// Do not scan the next token.
int parse_literal(int type) {

  // We have a string literal. Store in memory and return the label
  if ((type == pointer_to(P_CHAR)) && (Token.token == T_STRLIT))
    return(genglobstr(Text));
```

```
  if (Token.token == T_INTLIT) {
    switch(type) {
      case P_CHAR: if (Token.intvalue < 0 || Token.intvalue > 255)
                     fatal("Integer literal value too big for char type");
      case P_INT:
      case P_LONG: break;
      default: fatal("Type mismatch: integer literal vs. variable");
    }
  } else
    fatal("Expecting an integer literal value");
  return(Token.intvalue);
}
```

The first IF statement ensures that we can do:

```
char *str= "Hello world";
```

and it returns the label number of the address where the string is stored.

For integer literals, we check the range when we are assigning to a `char` variable. And for any other token type, we have a fatal error.

Changes to the Symbol Table Structure

The above function always returns an integer, regardless of what type of literal it parses. Now we need a location in each variable's symbol entry to store this. So, I've added (and/or modified) these fields in the symbol entry structure in `defs.h` :

```
// Symbol table structure
struct symtable {
  ...
  int size;             // Total size in bytes of this symbol
  int nelems;           // Functions: # params. Arrays: # elements
  ...
  int *initlist;        // List of initial values
  ...
};
```

For a scalar with one initial value, or for an array with several initial values, we store a count of elements in `nelems` and attach a list of integer values to `initlist` . Let's look at assignment to a scalar variable.

Assignment to Scalar Variables

The `scalar_declaration()` function is modified as follows:

```
static struct symtable *scalar_declaration(...) {
  ...
    // The variable is being initialised
  if (Token.token == T_ASSIGN) {
    // Only possible for a global or local
    if (class != C_GLOBAL && class != C_LOCAL)
      fatals("Variable can not be initialised", varname);
    scan(&Token);

    // Globals must be assigned a literal value
    if (class == C_GLOBAL) {
      // Create one initial value for the variable and
      // parse this value
      sym->initlist= (int *)malloc(sizeof(int));
      sym->initlist[0]= parse_literal(type);
      scan(&Token);
    }                           // No else code yet, soon
  }

  // Generate any global space
  if (class == C_GLOBAL)
    genglobsym(sym);

  return (sym);
}
```

We ensure that the assignment can only occur in global or local context, and we skip over the '=' token. We set up an `initlist` of exactly one and call `parse_literal()` with the type of this variable to get the literal value (or the label number of a string). Then we skip the literal value to get to the following token (either a ',' or a ';').

Previously, the `sym` symbol table entry was created with `addglob()` and the number of elements was set to one. I'll cover this change soon.

We now move the call to `genglobsym()` (which previously was in `addglob()` to here, and we wait until the initial value is stored in the `sym` entry. This ensures that the literal we just parsed will be put into the storage for the variable in memory.

Scalar Initialisation Examples

As a quick example:

```
int x= 5;
char *y= "Hello";
```

generates:

```
        .globl  x
x:
        .long   5

L1:
        .byte   72
        .byte   101
        .byte   108
        .byte   108
        .byte   111
        .byte   0

        .globl  y
y:
        .quad   L1
```

Changes to the Symbol Table Code

Before we get to the parsing of array intialisation, we need to detour over to the changes to the symbol table code. As I highlighted before, my original code didn't properly handle the storage of the size of a variable nor the number of elements in an array. Let's look at the changes I've made to do this.

Firstly, we have a bug fix. In `types.c`:

```
// Return true if a type is an int type
// of any size, false otherwise
int inttype(int type) {
  return (((type & 0xf) == 0) && (type >= P_CHAR && type <= P_LONG));
}
```

Previously, there was no test against P_CHAR, so a `void` type was treated as an integer type. Oops!

In `sym.c` we now deal with the fact that each variable now has a:

```
  int size;                     // Total size in bytes of this symbol
  int nelems;                   // Functions: # params. Arrays: # elements
```

Later, we will use the `size` field for the `sizeof()` operator. We now need to set up both fields when we add a symbol to the global or local symbol table.

The `newsym()` function and all of the `addXX()` functions in `sym.c` now take an `nelems` argument instead of a `size` argument. For scalar variables, this is set to one. For arrays, this is set to the number of elements in the list. For functions, this is set to the number of function parameters. And for all other symbol tables, the value is unused.

We now calculate the `size` value in `newsym()`:

```
  // For pointers and integer types, set the size
  // of the symbol. structs and union declarations
  // manually set this up themselves.
  if (ptrtype(type) || inttype(type))
    node->size = nelems * typesize(type, ctype);
```

`typesize()` consults the `ctype` pointer to get the size of a struct or union, or calls `genprimsize()` (which calls `cgprimsize()`) to get the size of a pointer or an integer type.

Note the comment about structs and unions. We can't call `addstruct()` (which calls `newsym()`) with the details of a struct's size, because:

```
struct foo {            // We call addglob() here
  int x;
  int y;                // before we know the size of the structure
  int z;
};
```

So the code in `composite_declaration()` in `decl.c` now does this:

```
static struct symtable *composite_declaration(...) {
  ...
  // Build the composite type
  if (type == P_STRUCT)
    ctype = addstruct(Text);
  else
    ctype = addunion(Text);
  ...
  // Scan in the list of members
  while (1) {
```

```
    ...
  }

  // Attach to the struct type's node
  ctype->member = Membhead;
  ...

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

So, in summary, the `size` field in a symbol table entry now holds the size of all of the elements in the variable, and `nelems` is the count of elements in the variable: one for arrays, some non-zero positive number for arrays.

Array Variable Initialisation

We can finally get to array initialisation. I want to allow three forms:

```
int a[10];                                  // Ten zeroed elements
char b[]= { 'q', 'w', 'e', 'r', 't', 'y' };     // Six elements
char c[10]= { 'q', 'w', 'e', 'r', 't', 'y' };   // Ten elements, zero padded
```

but prevent an array declared with size *N* and more than *N* initialisation values. Let's look at the changes to `array_declaration()`. Previously, I was going to call an `array_initialisation()` function, but I decided to move all of the initialisation code into `array_declaration()` in `decl.c`. We will take it in stages.

```
// Given the type, name and class of an variable, parse
// the size of the array, if any. Then parse any initialisation
// value and allocate storage for it.
// Return the variable's symbol table entry.
static struct symtable *array_declaration(...) {
  int nelems= -1;       // Assume the number of elements won't be given
  ...
  // Skip past the '['
  scan(&Token);

  // See we have an array size
  if (Token.token == T_INTLIT) {
    if (Token.intvalue <= 0)
      fatald("Array size is illegal", Token.intvalue);
```

```
    nelems= Token.intvalue;
    scan(&Token);
  }

  // Ensure we have a following ']'
  match(T_RBRACKET, "]");
```

If there's a number between the '[' ']' tokens, parse it and set `nelems` to this value. If there is no number, we leave it set to -1 to indicate this. We also check that the number is positive and non-zero.

```
    // Array initialisation
  if (Token.token == T_ASSIGN) {
    if (class != C_GLOBAL)
      fatals("Variable can not be initialised", varname);
    scan(&Token);

    // Get the following left curly bracket
    match(T_LBRACE, "{");
```

Right now I'm only dealing with global arrays.

```
#define TABLE_INCREMENT 10

    // If the array already has nelems, allocate that many elements
    // in the list. Otherwise, start with TABLE_INCREMENT.
    if (nelems != -1)
      maxelems= nelems;
    else
      maxelems= TABLE_INCREMENT;
    initlist= (int *)malloc(maxelems *sizeof(int));
```

We create an initial list of either 10 integers, or `nelems` if the array was given a fixed size. However, for arrays with no fixed size, we cannot predict how big the initialisation list will be. So we must be prepared to grow the list.

```
    // Loop getting a new literal value from the list
    while (1) {

      // Check we can add the next value, then parse and add it
      if (nelems != -1 && i == maxelems)
        fatal("Too many values in initialisation list");
```

```
      initlist[i++]= parse_literal(type);
      scan(&Token);
```

Get the next literal value and ensure we don't have more initial values that the array size if it was specified.

```
      // Increase the list size if the original size was
      // not set and we have hit the end of the current list
      if (nelems == -1 && i == maxelems) {
        maxelems += TABLE_INCREMENT;
        initlist= (int *)realloc(initlist, maxelems *sizeof(int));
      }
```

Here is where we increase the initialisation list size as necessary.

```
      // Leave when we hit the right curly bracket
      if (Token.token == T_RBRACE) {
        scan(&Token);
        break;
      }

      // Next token must be a comma, then
      comma();
    }
```

Parse the closing right curly bracket or a comma that separates values. Once out of the loop, we now have an `initlist` with values in it.

```
    // Zero any unused elements in the initlist.
    // Attach the list to the symbol table entry
    for (j=i; j < sym->nelems; j++) initlist[j]=0;
    if (i > nelems) nelems = i;
    sym->initlist= initlist;
  }
```

We may not have been given enough initialisation values to meet the specified size of the initialisation list, so zero out all the ones that were not initialised. It is here that we attach the initialisation list to the symbol table entry.

```
  // Set the size of the array and the number of elements
  sym->nelems= nelems;
  sym->size= sym->nelems * typesize(type, ctype);
  // Generate any global space
```

```
  if (class == C_GLOBAL)
    genglobsym(sym);
  return (sym);
}
```

We can finally updated `nelems` and `size` in the symbol table entry. Once this is done, we can call `genglobsym()` to create the memory storage for the array.

Changes to `cgglobsym()`

Before we look at the assembly output of an example array initialisation, we need to see how the changes of `nelems` and `size` have affected the code that generates the assembly for the memory storage.

`genglobsym()` is the front-end function which simply calls `cgglobsym()`. Let's look at this function in `cg.c`:

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

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

  // Get the size of the variable (or its elements if an array)
  // and the type of the variable
  if (node->stype == S_ARRAY) {
    size= typesize(value_at(node->type), node->ctype);
    type= value_at(node->type);
  } else {
    size = node->size;
    type= node->type;
  }
```

Right now, arrays have their `type` set to be a pointer to the underlying element type. This allows us to do:

```
char a[45];
char *b;
```

```
    b= a;          // as they are of same type
```

In terms of generating storage, we need to know the size of the elements, so we call `value_at()` to do this. For scalars, `size` and `type` are stored as-is in the symbol table entry.

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

As before. But now the code is different:

```
    // Output space for one or more elements
    for (i=0; i < node->nelems; i++) {

      // Get any initial value
      initvalue= 0;
      if (node->initlist != NULL)
        initvalue= node->initlist[i];

      // Generate the space for this type
      switch (size) {
        case 1:
          fprintf(Outfile, "\t.byte\t%d\n", initvalue);
          break;
        case 4:
          fprintf(Outfile, "\t.long\t%d\n", initvalue);
          break;
        case 8:
          // Generate the pointer to a string literal
          if (node->initlist != NULL && type== pointer_to(P_CHAR))
            fprintf(Outfile, "\t.quad\tL%d\n", initvalue);
          else
            fprintf(Outfile, "\t.quad\t%d\n", initvalue);
          break;
        default:
          for (int i = 0; i < size; i++)
          fprintf(Outfile, "\t.byte\t0\n");
      }
    }
  }
```

For every element, get its intial value from the `initlist` or use zero if no initialisation list. Based on the size of each element, output either a byte, a long or a quad.

For `char *` elements, we have the label of the string literal's base in the initialisation list, so output "L%d" (i.e. the label) instead of the integer literal value.

Array Initialisation Examples

Here is a small example of an array initialisation:

```
int x[4]= { 1, 4, 17 };
```

generates:

```
        .globl  x
x:
        .long   1
        .long   4
        .long   17
        .long   0
```

Test Programs

I won't go through the test programs, but the programs `tests/input89.c` through to `tests/input99.c` check that the compiler is generating sensible initialisation code as well as catching suitable fatal errors.

Conclusion and What's Next

So that was a lot of work! Three steps forward and one step back, as they say. I'm happy, though, because the changes to the symbol table make much more sense than what I had before.

In the next part of our compiler writing journey, we will try to add local variable initialisation to the compiler. Next step