Part 48: A Subset of `static`

In a real C compiler, there are three types of `static` things:

- static functions, whose declaration is visible only in the source file where the function appears;
- static global variables, whose declaration is visible only in the source file where the variable appears; and
- static local variables, which act like global variables except that each static local variables is only visible within the function where the variable appears.

The first two should be simple to implement:

- add them as a global variable when they are declared, and
- remove them from the global symbol table when that source code file is closed

The third one is much harder. Here's an example. Let's keep two private counters with functions to increment them:

```
int inc_counter1(void) {
  static int counter= 0;
  return(counter);
}

int inc_counter2(void) {
  static int counter= 0;
```

```
  return(counter);
}
```

Both functions see their own `counter` variable, and the value of both counters persist across function calls. The variable persistence makes them "global" (i.e. live outside of function scope) but they are only visible to one function, which makes them sort of "local".

I'll drop a reference to closures here, but the theory side is a bit out of scope, mainly because I'm *not* going to implement this third type of static things.

Why not? Mainly because it will be hard to implement something that has both global and local characteristics at the same time. But, also, I don't have any static local variables in our compiler (now that I've rewritten some code), and so there's no need for the functionality.

Instead, we can concentrate on static global functions and static global variables.

A New Keyword and Token

We have a new keyword `static` and a new token T_STATIC. As always, read through `scan.c` for the changes.

Parsing `static`

The `static` keyword gets parsed in the same place as `extern`. We also want to reject any attempt to use the `static` keyword in a local context. So in `decl.c`, we modify `parse_type()`:

```
// Parse the current token and return a primitive type enum value,
// a pointer to any composite type and possibly modify
// the class of the type.
int parse_type(struct symtable **ctype, int *class) {
  int type, exstatic = 1;

  // See if the class has been changed to extern or static
  while (exstatic) {
    switch (Token.token) {
      case T_EXTERN:
        if (*class == C_STATIC)
          fatal("Illegal to have extern and static at the same time");
        *class = C_EXTERN;
        scan(&Token);
        break;
      case T_STATIC:
        if (*class == C_LOCAL)
```

```
            fatal("Compiler doesn't support static local declarations");
          if (*class == C_EXTERN)
            fatal("Illegal to have extern and static at the same time");
          *class = C_STATIC;
          scan(&Token);
          break;
        default:
          exstatic = 0;
      }
    }
    ...
  }
```

If we see either `static` or `extern`, we firstly check if this is legal given the current declaration class. Then we update the `class` variable. If we see neither tokens, we leave the loop.

Now that we have a type which is marked as being for a static declaration, how is this added to the global symbol table? We need to change, in nearly every place in the compiler, any use of the C_GLOBAL class to also include C_STATIC. This occurs numerous times across multiple files, but you should look out for code like this:

```
if (class == C_GLOBAL || class == C_STATIC) ...
```

in `cg.c`, `decl.c`, `expr.c` and `gen.c`.

Getting Rid of `static` Declarations

Once we have finished parsing static declarations, we need to remove them from the global symbol table. In `do_compile()` in `main.c`, just after we close the input file, we now do this:

```
  genpreamble();                // Output the preamble
  global_declarations();        // Parse the global declarations
  genpostamble();               // Output the postamble
  fclose(Outfile);              // Close the output file
  freestaticsyms();             // Free any static symbols in the file
```

So let's now look at `freestaticsyms()` in `sym.c`. We walk the global symbol table. For any static node, we relink the list to remove it. I'm not a whiz at linked list code, so I wrote out all the possibly alternatives on a sheet of paper to come up with the following code:

```
// Remove all static symbols from the global symbol table
void freestaticsyms(void) {
  // g points at current node, prev at the previous one
  struct symtable *g, *prev= NULL;

  // Walk the global table looking for static entries
  for (g= Globhead; g != NULL; g= g->next) {
    if (g->class == C_STATIC) {

      // If there's a previous node, rearrange the prev pointer
      // to skip over the current node. If not, g is the head,
      // so do the same to Globhead
      if (prev != NULL) prev->next= g->next;
      else Globhead->next= g->next;

      // If g is the tail, point Globtail at the previous node
      // (if there is one), or Globhead
      if (g == Globtail) {
        if (prev != NULL) Globtail= prev;
        else Globtail= Globhead;
      }
    }
  }

  // Point prev at g before we move up to the next node
  prev= g;
}
```

The overall effect is to treat static declarations as global declarations, but to remove them from the symbol table at the end of processing an input file.

Testing the Changes

There are three programs to test the changes, `tests/input116.c` through to `tests/input118.c`. Let's look at the first one:

```
#include <stdio.h>

static int counter=0;
static int fred(void) { return(counter++); }

int main(void) {
  int i;
  for (i=0; i < 5; i++)
    printf("%d\n", fred());
```

```
  return(0);
}
```

Let's look at the assembly output for some of this:

```
        ...
        .data
counter:
        .long   0
        .text
fred:
        pushq   %rbp
        movq    %rsp, %rbp
        addq    $0,%rsp
        ...
```

Normally, `counter` and `fred` would have been decorated with a `.globl` marking. Now that they are static, they get labels but we tell the assembler not to make these globally visible.

Conclusion and What's Next

I was worried about `static`, but once I decided to not implement the really hard third alternative, it wasn't too bad. What caused me some grief was going through the code, finding all C_GLOBAL uses and ensuring that I added appropriate C_STATIC code as well.

In the next part of our compiler writing journey, I think it's time that I tackle the ternary operator. Next step