Part 29: A Bit of Refactoring

I started thinking about the design side of implementing structs, unions and enums in our compiler, and then I had a good idea on how to improve the symbol table, and that led me to doing a bit of refactoring of the compiler's code. So in this part of the journey there is no new functionality, but I feel a bit happier about some of the code in the compiler.

If you are more interested in my design ideas for structs, unions and enums, feel free to skip to the next part.

Refactoring the Symbol Table

When I started writing our compiler, I had just finished reading through the SubC compiler's code and adding my own comments. Thus, I borrowed many of my initial ideas from this code base. One of them was to have an array of elements for the symbol table, with global symbols at one end and local symbols at the other.

We've seen, for function prototypes and parameters, that we have to copy a function's prototype from the global end over to the local end so that the function has local parameter variables. And we have to worry if one end of the symbol table crashes into the other end.

So, at some point, we should convert the symbol table into a number of singly-linked lists: at least one for the global symbols and one for the local symbols. When we get to implementing enums, I might have a third one for the enum values.

Now, I haven't done this refactoring in this part of the journey as the changes look to be substantial, so I'll wait until I really need to do it. But one more change I will make is this. Each symbol node will have a `next` pointer to form the singly-linked list, but also a `param` pointer. This will allow functions to have a separate singly-linked list for their parameters which we can skip past when searching for global symbols. Then, when we need to "copy" a function's prototype to be its list of parameters, we can simply copy the pointer to the prototype list of parameters. Anyway, this change is for the future.

Types, Revisited

Another thing that I borrowed from SubC is the enumeration of types (in `defs.h`):

```
// Primitive types
enum {
  P_NONE, P_VOID, P_CHAR, P_INT, P_LONG,
  P_VOIDPTR, P_CHARPTR, P_INTPTR, P_LONGPTR
};
```

SubC only allows one level of indirection, thus the list of types above. I had the idea, why not encode the level of indirection in the primitive type value? So I've changed our code so that the bottom four bits in a `type` integer is the level of indirection, and the higher bits encode the actual type:

```
// Primitive types. The bottom 4 bits is an integer
// value that represents the level of indirection,
// e.g. 0= no pointer, 1= pointer, 2= pointer pointer etc.
enum {
  P_NONE, P_VOID=16, P_CHAR=32, P_INT=48, P_LONG=64
};
```

I've been able to completely refactor out all of the old `P_XXXPTR` references in the old code. Let's see what changes there have been.

Firstly, we have to deal with scalar and pointer types in `types.c`. The code now is actually smaller than before:

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

```
// Return true if a type is of pointer type
int ptrtype(int type) {
  return ((type & 0xf) != 0);
}

// Given a primitive type, return
// the type which is a pointer to it
int pointer_to(int type) {
  if ((type & 0xf) == 0xf)
    fatald("Unrecognised in pointer_to: type", type);
  return (type + 1);
}

// Given a primitive pointer type, return
// the type which it points to
int value_at(int type) {
  if ((type & 0xf) == 0x0)
    fatald("Unrecognised in value_at: type", type);
  return (type - 1);
}
```

And `modify_type()` hasn't changed whatsoever.

In `expr.c`, when dealing with literal strings, I was using `P_CHARPTR` but now I can write:

```
    n = mkastleaf(A_STRLIT, pointer_to(P_CHAR), id);
```

One other substantial area where the `P_XXXPTR` values were used is the code in the hardware-dependent code in `cg.c`. We start by rewriting `cgprimsize()` to use `ptrtype()`:

```
// Given a P_XXX type value, return the
// size of a primitive type in bytes.
int cgprimsize(int type) {
  if (ptrtype(type)) return (8);
  switch (type) {
    case P_CHAR: return (1);
    case P_INT:  return (4);
    case P_LONG: return (8);
    default: fatald("Bad type in cgprimsize:", type);
  }
  return (0);                   // Keep -Wall happy
}
```

With this code, the other functions in `cg.c` can now call `cgprimsize()`, `ptrtype()`, `inttype()`, `pointer_to()` and `value_at()` as required, instead of referring to specific types. Here's an example from `cg.c`:

```
// Dereference a pointer to get the value it
// pointing at into the same register
int cgderef(int r, int type) {

  // Get the type that we are pointing to
  int newtype = value_at(type);

  // Now get the size of this type
  int size = cgprimsize(newtype);

  switch (size) {
  case 1:
    fprintf(Outfile, "\tmovzbq\t(%s), %s\n", reglist[r], reglist[r]);
    break;
  case 2:
    fprintf(Outfile, "\tmovslq\t(%s), %s\n", reglist[r], reglist[r]);
    break;
  case 4:
  case 8:
    fprintf(Outfile, "\tmovq\t(%s), %s\n", reglist[r], reglist[r]);
    break;
  default:
    fatald("Can't cgderef on type:", type);
  }
  return (r);
}
```

Have a quick read through `cg.c` and look for the calls to `cgprimsize()`.

An Example Use of Double Pointers

Now that we have up to sixteen levels of indirection, I wrote a test program to confirm that they work, `tests/input55.c`:

```
int printf(char *fmt);

int main(int argc, char **argv) {
  int i;
  char *argument;
  printf("Hello world\n");

  for (i=0; i < argc; i++) {
```

```
    argument= *argv; argv= argv + 1;
    printf("Argument %d is %s\n", i, argument);
  }
  return(0);
}
```

Note that `argv++` doesn't yet work, and `argv[i]` also doesn't yet work. But we can work around these missing features as shown above.

Changes to the Symbol Table Structure

While I didn't refactor the symbol table into lists, I did tweak the symbol table structure itself, now that I realised that I can use unions and not have to give the union a variable name:

```
// Symbol table structure
struct symtable {
  char *name;                   // Name of a symbol
  int type;                     // Primitive type for the symbol
  int stype;                    // Structural type for the symbol
  int class;                    // Storage class for the symbol
  union {
    int size;                   // Number of elements in the symbol
    int endlabel;               // For functions, the end label
  };
  union {
    int nelems;                 // For functions, # of params
    int posn;                   // For locals, the negative offset
                                // from the stack base pointer
  };
};
```

I used to have a `#define` for `nelems`, but the above is the same result and prevents the global definition of `nelems` from polluting the namespace. I also realised that `size` and `endlabel` could occupy the same position in the structure, and added that union. There are a few cosmetic changes to the parameters to `addglob()`, but not much else.

Changes to the AST Structure

Similarly, I've modified the AST node structure so that the union doesn't have a variable name:

```
// Abstract Syntax Tree structure
struct ASTnode {
  int op;                       // "Operation" to be performed on this tree
  int type;                     // Type of any expression this tree generates
  int rvalue;                   // True if the node is an rvalue
  struct ASTnode *left;         // Left, middle and right child trees
  struct ASTnode *mid;
  struct ASTnode *right;
  union {                       // For A_INTLIT, the integer value
    int intvalue;               // For A_IDENT, the symbol slot number
    int id;                     // For A_FUNCTION, the symbol slot number
    int size;                   // For A_SCALE, the size to scale by
  };                            // For A_FUNCCALL, the symbol slot number
};
```

and this means that I can, e.g., write the second line instead of the first one:

```
    return (cgloadglob(n->left->v.id, n->op));    // Old code
    return (cgloadglob(n->left->id,   n->op));    // New code
```

Conclusion and What's Next

That's about it for this part of our compiler writing journey. I might have done a few more small code changes here and there, but I can't think of anything else that was major.

I will get to changing the symbol table to be a linked list; this will probably happen in the part where we implement enumerated values.

In the next part of our compiler writing journey, I'll get back to what I wanted to cover in this part: the design side of implementing structs, unions and enums in our compiler. Next step