Part 46: Void Function Parameters and Scanning Changes

In this part of our compiler writing journey, I've made several changes which involve the scanner and the parser.

Void Function Parameters

We start with this common C construct to indicate that a function has no parameters:

```
int fred(void);         // Void means no parameters, but
int fred();             // No parameters also means no parameters
```

It does seem strange that we already have a way of indicating no parameters but, anyway, it's a common thing so we need to support it.

The problem is that, once we hit the left parenthesis, we fall into the `declaration_list()` function in `decl.c`. This has been set up to parse a type with a definite following identifier. It's not going to be easy to alter it to deal with a type and *no* identifier. So we need to go back to the `param_declaration_list()` function and parse the 'void' ')' tokens there.

I already have a function in the scanner called `reject_token()` in `scan.c`. We should be able to scan a token, look at it, decide we don't want it, and reject it. Then, the next scanned token will be the one we reject.

I've never used this function and, as it turns out, it was broken. Anyway, I took a step back and decided that it would be easier to *peek* at the next token. If we decide we like it, we can scan it in as per normal. If we don't like it, we don't have to do anything: it will get scanned in on the next real token scan.

Now, why do we need this? It's because our pseudo-code for dealing with 'void' in the parameter list will be:

```
  parse the '('
  if the next token is 'void' {
    peek at the one after it
    if the one after 'void' is ')',
    then return zero parameters
  }
  call declaration_list() to get the real parameters
  so that 'void' is still the current token
```

We need to do the peek because both of the following are legal:

```
int fred(void);
int jane(void *ptr, int x, int y);
```

If we scan and parse the next token after 'void' and see it is the asterisk, then we have lost the 'void' token. When we then call `declaration_list()`, the first token it will see is the asterisk and it will get upset. Thus, we need the ability to peek beyond the current token while keeping the current token intact.

New Scanner Code

In `data.h` we have a new token variable:

```
extern_ struct token Token;             // Last token scanned
extern_ struct token Peektoken;         // A look-ahead token
```

and `Peektoken.token` is intialised to zero by code in `main.c`. We modify the main `scan()` function in `scan.c` as follows:

```
// Scan and return the next token found in the input.
// Return 1 if token valid, 0 if no tokens left.
int scan(struct token *t) {
  int c, tokentype;
```

```
    // If we have a lookahead token, return this token
    if (Peektoken.token != 0) {
      t->token = Peektoken.token;
      t->tokstr = Peektoken.tokstr;
      t->intvalue = Peektoken.intvalue;
      Peektoken.token = 0;
      return (1);
    }
    ...
  }
```

If `Peektoken.token` remains zero, we get the next token. But once something is stored in `Peektoken`, then that will be the next token we return.

Declaration Modifications

Now that we can peek ahead at the next token, let's put it into action. We modify the code in `param_declaration_list()` as follows:

```
    // Loop getting any parameters
    while (Token.token != T_RPAREN) {

      // If the first token is 'void'
      if (Token.token == T_VOID) {
        // Peek at the next token. If a ')', the function
        // has no parameters, so leave the loop.
        scan(&Peektoken);
        if (Peektoken.token == T_RPAREN) {
          // Move the Peektoken into the Token
          paramcnt= 0; scan(&Token); break;
        }
      }
      ...
      // Get the type of the next parameter
      type = declaration_list(&ctype, C_PARAM, T_COMMA, T_RPAREN, &unused);
      ...
    }
```

Assume that we have scanned in the 'void'. We now `scan(&Peektoken);` to see what's up next without altering the current `Token`. If that's a right parenthesis, we can leave with `paramcnt` set to zero after skipping the 'void' token.

But if the next token wasn't a right parenthesis, we still have `Token` set to 'void' and we can now call `declaration_list()` to get the actual list of parameters.

Hex and Octal Integer Constants

I found the above problem because I've started to feed the compiler's source code to itself. Once I had fixed the 'void' parameter issue, the next thing that I found was that the compiler is unable to parse hex and octal constants like `0x314A` and `0073` .

Luckily, the SubC compiler written by Nils M Holm has code to do this, and I can borrow it wholesale to add to our compiler. We need to modify the `scanint()` function in `scan.c` to do this:

```
// Scan and return an integer literal
// value from the input file.
static int scanint(int c) {
  int k, val = 0, radix = 10;

  // NEW CODE: Assume the radix is 10, but if it starts with 0
  if (c == '0') {
    // and the next character is 'x', it's radix 16
    if ((c = next()) == 'x') {
      radix = 16;
      c = next();
    } else
      // Otherwise, it's radix 8
      radix = 8;

  }

  // Convert each character into an int value
  while ((k = chrpos("0123456789abcdef", tolower(c))) >= 0) {
    if (k >= radix)
      fatalc("invalid digit in integer literal", c);
    val = val * radix + k;
    c = next();
  }

  // We hit a non-integer character, put it back.
  putback(c);
  return (val);
}
```

We already had the `k= chrpos("0123456789")` code in the function to deal with decimal literal values. The new code above this now scans for a leading '0' digit. If it sees this, it checks the following character. If it's an 'x', the radix is 16; if not, the radix is 8.

The other change is that we multiply the previous value by the radix instead of the constant 10. It's a very elegant way to solve this problem, and many thanks to Nils for writing the code.

More Character Constants

The next problem I hit was code in our compiler that says:

```
  if (*posn == '\0')
```

That's a character literal which our compiler doesn't recognise. We will need to modify `scanch()` in `scan.c` to deal with character literals which are specified as octal values. But character literals which are specified as hexadecimal values are also possible, e.g. '\0x41'. Again, the code from SubC comes to our rescue:

```
// Read in a hexadecimal constant from the input
static int hexchar(void) {
  int c, h, n = 0, f = 0;

  // Loop getting characters
  while (isxdigit(c = next())) {
    // Convert from char to int value
    h = chrpos("0123456789abcdef", tolower(c));
    // Add to running hex value
    n = n * 16 + h;
    f = 1;
  }
  // We hit a non-hex character, put it back
  putback(c);
  // Flag tells us we never saw any hex characters
  if (!f)
    fatal("missing digits after '\\x'");
  if (n > 255)
    fatal("value out of range after '\\x'");
  return n;
}

// Return the next character from a character
// or string literal
static int scanch(void) {
  int i, c, c2;

  // Get the next input character and interpret
  // metacharacters that start with a backslash
  c = next();
  if (c == '\\') {
    switch (c = next()) {
      ...
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':                 // Code from SubC
        for (i = c2 = 0; isdigit(c) && c < '8'; c = next()) {
          if (++i > 3)
            break;
          c2 = c2 * 8 + (c - '0');
        }
        putback(c);             // Put back the first non-octal char
        return (c2);
      case 'x':
        return hexchar();       // Code from SubC
      default:
        fatalc("unknown escape sequence", c);
    }
  }
  return (c);                   // Just an ordinary old character!
}
```

Again, it's nice and elegant code. However, we now have two code fragments to do hex conversion and three code fragments to do radix conversion, so there is still some potential refactoring here.

Conclusion and What's Next

We mostly made changes to the scanner in this part of the journey. They were not earth shattering changes, but they are some of the little things that we need to get done to have the compiler be self-compiling.

Two big things that we will need to tackle are static functions and variables, and the `sizeof()` operator.

In the next part of our compiler writing journey, I will probably work on the `sizeof()` operator because `static` still scares me a bit! [Next step]