Part 35: The C Pre-Processor

In this part of our compiler writing journey, I've added support for an external C pre-processor, and I also added the `extern` keyword to our language.

We've reached the point where we can write header files for our programs, and also put comments in them. I must admit, this feels good.

The C Pre-Processor

I don't want to write about the C pre-processor itself, even though is a very important part of any C environment. Instead, I'll point you at these two articles to read:

- C Preprocessor at *Wikipedia*
- C Preprocessor and Macros at *www.programiz.com*

Integrating the C Pre-Processor

In other compilers like SubC, the pre-processor is built right into the language. Here I've decided to use the external system C pre-processor which is usually the Gnu C pre-processor.

Before I show you how I've done this, firstly we need to look at the lines that the pre-processor inserts as part of its operation.

Consider this short program (with lines numbered):

```c
#include <stdio.h>

int main() {
  printf("Hello world\n");
  return(0);
}
```

Here is what we (the compiler) might receive from the pre-processor after it processes this file:

```
# 1 "z.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "z.c"
# 1 "include/stdio.h" 1
# 1 "include/stddef.h" 1

typedef long size_t;
# 5 "include/stdio.h" 2

typedef char * FILE;

FILE *fopen(char *pathname, char *mode);
...
# 2 "z.c" 2

int main() {
  printf("Hello world\n");
  return(0);
}
```

Each pre-processor line starts with a '#', then the number of the following line, then the name of the file from where this line comes from. The numbers at the end of some of the lines I don't really know what they are. I suspect, when one file includes another, they represent the line number of the file that did the including.

Here is how I'm going to integrate the pre-processor with our compiler. I'm going to use `popen()` to open up a pipe from a process which is the pre-processor, and we will tell the pre-processor to work on our input file. Then we will modify the lexical scanner to identify the pre-processor lines and set the current line number and name of the file being processed.

Modifications to `main.c`

We have a new global variable, `char *Infilename`, defined in `data.h`. In the `do_compile()` function in `main.c` we now do this:

```
// Given an input filename, compile that file
// down to assembly code. Return the new file's name
static char *do_compile(char *filename) {
  char cmd[TEXTLEN];
  ...
  // Generate the pre-processor command
  snprintf(cmd, TEXTLEN, "%s %s %s", CPPCMD, INCDIR, filename);

  // Open up the pre-processor pipe
  if ((Infile = popen(cmd, "r")) == NULL) {
    fprintf(stderr, "Unable to open %s: %s\n", filename, strerror(errno));
    exit(1);
  }
  Infilename = filename;
```

which I think is a straight-forward piece of code, except that I haven't explained where `CPPCMD` and `INCDIR` come from.

`CPPCMD` is defined as the name of the pre-processor command in `defs.h`:

```
#define CPPCMD "cpp -nostdinc -isystem "
```

This tells the Gnu pre-processor to not use the standard include directory `/usr/include`: instead, `-isystem` tells the pre-processor to use the next thing on the command line which is `INCDIR`.

`INCDIR` is actually defined in the `Makefile`, as this is a common place to put things that can be changed at configuration time:

```
# Define the location of the include directory
# and the location to install the compiler binary
INCDIR=/tmp/include
BINDIR=/tmp
```

The compiler binary is now compiled with this `Makefile` rule:

```
cwj: $(SRCS) $(HSRCS)
        cc -o cwj -g -Wall -DINCDIR=\"$(INCDIR)\" $(SRCS)
```

and this passes the `/tmp/include` value in to the compilation as `INCDIR`. Now, when does `/tmp/include` get created, and what gets put there?

Our First Set of Header Files

In the `include/` directory in this area, I've made a start on some header files that are plain enough for our compiler to digest. We can't use the real system header files, as they contain lines like:

```
extern int _IO_feof (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__));
extern int _IO_ferror (_IO_FILE *__fp) __attribute__ ((__nothrow__ , __leaf__)
```

which would cause our compiler to have a fit! There is now a rule in the `Makefile` to copy our own header files to the `INCDIR` directory:

```
install: cwj
        mkdir -p $(INCDIR)
        rsync -a include/. $(INCDIR)
        cp cwj $(BINDIR)
        chmod +x $(BINDIR)/cwj
```

Scanning the Pre-Processor Input

So now we are reading the pre-processor output from working on the input file, and not reading from the file directly. We now need to recognise pre-processor lines and set the number of the next line and the file's name where the line came from.

I've modified the scanner to do this, as this already deals with incrementing the line number. So in `scan.c`, I've made this change to the `scan()` function:

```
// Get the next character from the input file.
static int next(void) {
  int c, l;

  if (Putback) {                        // Use the character put
    c = Putback;                        // back if there is one
```

```
      Putback = 0;
      return (c);
    }

    c = fgetc(Infile);                        // Read from input file

    while (c == '#') {                        // We've hit a pre-processor statement
      scan(&Token);                           // Get the line number into l
      if (Token.token != T_INTLIT)
        fatals("Expecting pre-processor line number, got:", Text);
      l = Token.intvalue;

      scan(&Token);                           // Get the filename in Text
      if (Token.token != T_STRLIT)
        fatals("Expecting pre-processor file name, got:", Text);

      if (Text[0] != '<') {                   // If this is a real filename
        if (strcmp(Text, Infilename))         // and not the one we have now
          Infilename = strdup(Text);          // save it. Then update the line num
        Line = l;
      }

      while ((c = fgetc(Infile)) != '\n'); // Skip to the end of the line
      c = fgetc(Infile);                      // and get the next character
    }

    if ('\n' == c)
      Line++;                                 // Increment line count
    return (c);
  }
```

We use a 'while' loop because there can be successive pre-processor lines. We are fortunate that we can call `scan()` recursively to scan in both the line number as a T_INTLIT and the file's name as a T_STRLIT.

The code ignores filenames that are enclosed in '<' ... '>', as these don't represent real filenames. We do have to `strdup()` the file's name as it is in the global `Text` variable which will get overwritten. However, if the name in `Text` is already what's in `Infilename` , we don't need to duplicate it.

Once we have the line number and filename, we read up to and one character past the end of the line, then go back to our original character scanning code.

And that turned out to be all that was needed to integrate the C pre-processor with our compiler. I had worried that it would be complex to do this, but it wasn't.

Preventing Unwanted Function/Variable Redeclarations

Many header files include other header files, so there is a strong chance that one header file might get included multiple times. This would cause redeclarations of the same function and/or global variable.

To prevent this, I'm using the normal header mechanism of defining a header-specific macro the first time a header file is included. This then prevents the contents of the header file being included a second time.

As an example, here is what is currently in `include/stdio.h` :

```
#ifndef _STDIO_H_
# define _STDIO_H_

#include <stddef.h>

// This FILE definition will do for now
typedef char * FILE;

FILE *fopen(char *pathname, char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);
int printf(char *format);
int fprintf(FILE *stream, char *format);

#endif  // _STDIO_H_
```

Once `_STDIO_H_` is defined, it prevents this file's contents from being included a second time.

The `extern` Keyword

Now that we have a working pre-processor, I thought it would be time to add the `extern` keyword to the language. This would allow us to define a global variable but not generate any storage for it: the assumption is that the variable has been declared global in another source file.

The addition of `extern` actually has an impact across several files. Not a big impact, but a widespread impact. Let's see this.

A New Token and Keyword

So, we have a new keyword `extern` and a new token T_EXTERN in `scan.c`. As always, the code is there for you to read.

A New Class

In `defs.h` we have a new storage class:

```
// Storage classes
enum {
  C_GLOBAL = 1,                 // Globally visible symbol
  ...
  C_EXTERN,                     // External globally visible symbol
  ...
};
```

The reason I put this in is because we already have this code for global symbols in `sym.c`:

```
// Create a symbol node to be added to a symbol table list.
struct symtable *newsym(char *name, int type, struct symtable *ctype,
                        int stype, int class, int size, int posn) {
  // Get a new node
  struct symtable *node = (struct symtable *) malloc(sizeof(struct symtable));
  // Fill in the values
  ...
    // Generate any global space
  if (class == C_GLOBAL)
    genglobsym(node);
```

We want `extern` symbols added to the global list, but we don't want to call `genglobsym()` to create the storage for them. So, we need to call `newsym()` with a class that isn't C_GLOBAL.

Changes to `sym.c`

To this end, I've modified `addglob()` to take a `class` argument which is passed to `newsym()`:

```
// Add a symbol to the global symbol list
struct symtable *addglob(char *name, int type, struct symtable *ctype,
                         int stype, int class, int size) {
  struct symtable *sym = newsym(name, type, ctype, stype, class, size, 0);
  appendsym(&Globhead, &Globtail, sym);
```

```
    return (sym);
  }
```

This means that, everywhere that we call `addglob()` in the compiler, we now must pass in a `class` value. Before, `addglob()` would explicitly pass C_GLOBAL to `newsym()`. Now, we must pass the `class` value we want to `addglob()`.

The `extern` Keyword and Our Grammar

In terms of the grammar of our language, I'm going to enforce the rule that the `extern` keyword must come before any other words in a type description. Later on, I'll add `static` to the list of words. The BNF Grammar for C that we saw in past parts has these production rules:

```
storage_class_specifier
        : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;

declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        | type_qualifier
        | type_qualifier declaration_specifiers
        ;
```

which I think allows `extern` to come anywhere in the type specification. Oh well, we are building a subset of the C language here!

Parsing the `extern` Keyword

As with the last five or six parts of this journey, I've made changes to `parse_type()` in `decl.c` again:

```
int parse_type(struct symtable **ctype, int *class) {
  int type, exstatic=1;

  // See if the class has been changed to extern (later, static)
  while (exstatic) {
    switch (Token.token) {
      case T_EXTERN: *class= C_EXTERN; scan(&Token); break;
      default: exstatic= 0;
    }
  }
  ...
}
```

Note now that `parse_type()` has a second parameter, `int *class`. This allows the caller to pass in the initial storage class for the type (probably C_GLOBAL, G_LOCAL or C_PARAM). If we see the `extern` keyword in `parse_type()`, we can change to become T_EXTERN. Also apologies, I couldn't think of a good name for the boolean flag that controls the 'while' loop.

The `parse_type()` and `addglob()` Callers

So we've modified the arguments to both `parse_type()` and `addglob()`. Now we have to find everywhere in the compiler where both functions are called, and ensure we pass a suitable `class` value to both of them.

In `var_declaration_list()` in `decl.c` where we are parsing a list of variables or parameters, we already get the storage class for these variables:

```
static int var_declaration_list(struct symtable *funcsym, int class,
                                int separate_token, int end_token);
```

So we can pass the `class` to `parse_type()` which may change it, then call `var_declaration()` with the actual class:

```
    ...
    // Get the type and identifier
    type = parse_type(&ctype, &class);
    ident();
    ...
    // Add a new parameter to the right symbol table list, based on the class
    var_declaration(type, ctype, class);
```

And in `var_declaration()` :

```
      switch (class) {
        case C_EXTERN:
        case C_GLOBAL:
          sym = addglob(Text, type, ctype, S_VARIABLE, class, 1);
        ...
      }
```

For local variables, we need to turn our attention to `single_statement()` in `stmt.c` . I also should, at this point, say that I'd previously forgot to add the cases for structs, unions, enums and typedefs here.

```
// Parse a single statement and return its AST
static struct ASTnode *single_statement(void) {
  int type, class= C_LOCAL;
  struct symtable *ctype;

  switch (Token.token) {
    case T_IDENT:
      // We have to see if the identifier matches a typedef.
      // If not do the default code in this switch statement.
      // Otherwise, fall down to the parse_type() call.
      if (findtypedef(Text) == NULL)
        return (binexpr(0));
    case T_CHAR:
    case T_INT:
    case T_LONG:
    case T_STRUCT:
    case T_UNION:
    case T_ENUM:
    case T_TYPEDEF:
      // The beginning of a variable declaration.
```

```
        // Parse the type and get the identifier.
        // Then parse the rest of the declaration
        // and skip over the semicolon
        type = parse_type(&ctype, &class);
        ident();
        var_declaration(type, ctype, class);
        semi();
        return (NULL);           // No AST generated here
        ...
      }
      ...
  }
```

Note that we start with `class= C_LOCAL`, but it might get modified by `parse_type()` before being passed to `var_declaration()`. This allows us to write code that looks like:

```
int main() {
  extern int foo;
  ...
}
```

Testing the Code

I've got one test program, `test/input70.c` which uses one of our new header files to confirm that the pre-processor works:

```
#include <stdio.h>

typedef int FOO;

int main() {
  FOO x;
  x= 56;
  printf("%d\n", x);
  return(0);
}
```

I was hoping that `errno` was still an ordinary integer so that I could declare `extern int errno;` in `include/errno.h`. But, apparently, `errno` is now a function and not a global integer variable. I think this tells you a) how old I am and b) how long it is since I've written C code.

Conclusion and What's Next

I feel like we have hit another milestone here. We now have external variables and header files. This also means that, *finally*, we can put comments into our source files. That really makes me happy.

We are up to just over 4,100 lines of code, of which about 2,800 lines are not comments and not whitespace. I have no idea exactly how many more lines of code we'll need to make the compiler self-compiling, but I'm going to hazard a guess of between 7,000 to 9,000 lines. We'll see!

In the next part of our compiler writing journey, we will add the `break` and `continue` keywords to our loop constructs. Next step