Part 34: Enums and Typedefs

I decided to implement both enums and typedefs in this part of our compiler writing journey, as each one was quite small.

We already covered the design aspects of enums back in part 30. To revise briefly, enums are just named integer literals. There were two issues to deal with:

- we cannot redefine an enum type name, and
- we cannot redefine a named enum value

As examples of the above:

```
enum fred { x, y, z };
enum fred { a, b };          // fred is redefined
enum jane { x, y };          // x and y are redefined
```

As you can see above, a list of enumerated values only has identifier names and no types: it means we can't reuse our existing variable declaration parsing code. We will have to write our own parsing code here.

New Keywords and Tokens

I've added two new keywords, 'enum' and 'typedef' to the grammar along with two tokens, T_ENUM and T_TYPEDEF. Browse through the code in `scan.c` for details.

Symbol Table Lists for Enums and Typedefs

We need to record the details of the declared enums and typedefs, so there are two new symbol table lists in `data.h` :

```
extern_ struct symtable *Enumhead,  *Enumtail;    // List of enum types and va
extern_ struct symtable *Typehead,  *Typetail;    // List of typedefs
```

and in `sym.c` there are associated functions to add entries to each list and to search each list for specific names. Nodes in these lists are marked as being one of (from `defs.h`):

```
  C_ENUMTYPE,                   // A named enumeration type
  C_ENUMVAL,                    // A named enumeration value
  C_TYPEDEF                     // A named typedef
```

OK, so two lists but three node classes, what's going on? It turns out that enum values (like `x` and `y` in the examples at the top) don't belong to any specific enum type. Also, enum type names (like `fred` and `jane` in the examples at the top) don't really do anything, but we do have to prevent redefinitions of them.

I'm using the one enum symbol table list to hold both the C_ENUMTYPE and the C_ENUMVAL nodes in the same lists. Using the examples near the top, we would have:

```
    fred           x            y            z
C_ENUMTYPE -> C_ENUMVAL -> C_ENUMVAL -> C_ENUMVAL
                   0            1            2
```

This also means that, when we are searching the enum symbol table list, we need the ability to search for C_ENUMTYPEs or for C_ENUMVALs.

Parsing Enum Declarations

Before I give the code to do this, let's just look at some examples of what we need to parse:

```
enum fred { a, b, c };              // a is 0, b is 1, c is 2
enum foo  { d=2, e=6, f };          // d is 2, e is 6, f is 7
enum bar  { g=2, h=6, i } var1;     // var1 is really an int
enum      { j, k, l }     var2;     // var2 is really an int
```

Firstly, where does enum parsing get attached to our existing parsing code? As with structs and unions, in the code that parses types (in `decl.c`):

```
// Parse the current token and return
// a primitive type enum value and a pointer
// to any composite type.
// Also scan in the next token
int parse_type(struct symtable **ctype) {
  int type;
  switch (Token.token) {

      // For the following, if we have a ';' after the
      // parsing then there is no type, so return -1
      ...
    case T_ENUM:
      type = P_INT;             // Enums are really ints
      enum_declaration();
      if (Token.token == T_SEMI)
        type = -1;
      break;
  }
  ...
}
```

I've changed the return value of `parse_type()` to help identify when it was a declaration of a struct, union, enum or typedef and not an actual type (followed by an identifier).

Let's now look at the `enum_declaration()` code in stages.

```
// Parse an enum declaration
static void enum_declaration(void) {
  struct symtable *etype = NULL;
  char *name;
  int intval = 0;

  // Skip the enum keyword.
  scan(&Token);

  // If there's a following enum type name, get a
  // pointer to any existing enum type node.
  if (Token.token == T_IDENT) {
    etype = findenumtype(Text);
    name = strdup(Text);        // As it gets tromped soon
    scan(&Token);
  }
```

We only have one global variable, `Text`, to hold a scanned-in word, and we have to be able to parse `enum foo var1`. If we scan in the token after the `foo`, we will lose the `foo` string. So we need to `strdup()` this.

```
// If the next token isn't a LBRACE, check
// that we have an enum type name, then return
if (Token.token != T_LBRACE) {
  if (etype == NULL)
    fatals("undeclared enum type:", name);
  return;
}
```

We've hit a declaration like `enum foo var1` and not `enum foo { ...`. Therefore `foo` must already exist as a known enum type. We can return with no value, as the type of every enum is P_INT, which is set in the code that calls `enum_declaration()`.

```
// We do have an LBRACE. Skip it
scan(&Token);

// If we have an enum type name, ensure that it
// hasn't been declared before.
if (etype != NULL)
  fatals("enum type redeclared:", etype->name);
else
  // Build an enum type node for this identifier
  etype = addenum(name, C_ENUMTYPE, 0);
```

Now we are parsing something like `enum foo { ...`, so we must check that `foo` has not already been declared as an enum type.

```
// Loop to get all the enum values
while (1) {
  // Ensure we have an identifier
  // Copy it in case there's an int literal coming up
  ident();
  name = strdup(Text);

  // Ensure this enum value hasn't been declared before
  etype = findenumval(name);
  if (etype != NULL)
    fatals("enum value redeclared:", Text);
```

Again, we `strdup()` the enum value identifier. We also check that this enum value identifier hasn't already been defined.

```
      // If the next token is an '=', skip it and
      // get the following int literal
      if (Token.token == T_ASSIGN) {
        scan(&Token);
        if (Token.token != T_INTLIT)
          fatal("Expected int literal after '='");
        intval = Token.intvalue;
        scan(&Token);
      }
```

This is why we had to `strdup()` as the scanning of an integer literal will walk over the `Text` global variable. We scan in the '=' and integer literal tokens here and set the `intval` variable to be the integer literal value.

```
      // Build an enum value node for this identifier.
      // Increment the value for the next enum identifier.
      etype = addenum(name, C_ENUMVAL, intval++);

      // Bail out on a right curly bracket, else get a comma
      if (Token.token == T_RBRACE)
        break;
      comma();
    }
    scan(&Token);                 // Skip over the right curly bracket
  }
```

We now have the enum value's name and its value in `intval`. We can add this to the enum symbol table list with `addenum()`. We also increment `intval` to be ready for the next enum value identifier.

Accessing Enum Names

We now have the code to parse the list of enum value names and store their integer literal values in the symbol table. How and when do we search for them and use them?

We have to do this at the point where we could be using a variable name in an expression. If we find an enum name, we convert it into an A_INTLIT AST node with a specific value. The location to do this is `postfix()` in `expr.c`

```
// Parse a postfix expression and return
// an AST node representing it. The
// identifier is already in Text.
static struct ASTnode *postfix(void) {
  struct symtable *enumptr;

  // If the identifier matches an enum value,
  // return an A_INTLIT node
  if ((enumptr = findenumval(Text)) != NULL) {
    scan(&Token);
    return (mkastleaf(A_INTLIT, P_INT, NULL, enumptr->posn));
  }
  ...
}
```

Testing the Functionality

All done! There are several test programs that confirm we are spotting redefined enum types and names, but the `test/input63.c` code demonstrates enums working:

```
int printf(char *fmt);

enum fred { apple=1, banana, carrot, pear=10, peach, mango, papaya };
enum jane { aple=1, bnana, crrot, par=10, pech, mago, paaya };

enum fred var1;
enum jane var2;
enum fred var3;

int main() {
  var1= carrot + pear + mango;
  printf("%d\n", var1);
  return(0);
```

which adds `carrot + pear + mango` (i.e. 3+10+12) and prints out 25.

Typedefs

That's enums done. Now we look at typedefs. The basic grammar of a typedef declaration is:

```
typedef_declaration: 'typedef' identifier existing_type
                   | 'typedef' identifier existing_type variable_name
```

```
                    ;
```

Thus, once we parse the `typedef` keyword, we can parse the following type and build a C_TYPEDEF symbol node with the name. We can store the `type` and `ctype` of the actual type in this symbol node.

The parsing code is nice and simple. We hook into `parse_type()` in `decl.c` :

```
    case T_TYPEDEF:
      type = typedef_declaration(ctype);
      if (Token.token == T_SEMI)
        type = -1;
      break;
```

Here is the `typedef_declaration()` code. Note that it returns the actual `type` and `ctype` in case the declaration is followed by a variable name.

```
// Parse a typedef declaration and return the type
// and ctype that it represents
int typedef_declaration(struct symtable **ctype) {
  int type;

  // Skip the typedef keyword.
  scan(&Token);

  // Get the actual type following the keyword
  type = parse_type(ctype);

  // See if the typedef identifier already exists
  if (findtypedef(Text) != NULL)
    fatals("redefinition of typedef", Text);

  // It doesn't exist so add it to the typedef list
  addtypedef(Text, type, *ctype, 0, 0);
  scan(&Token);
  return (type);
}
```

The code should be straight-forward but note the recursive call back to `parse_type()` : we already have the code to parse the type definition after the name of the typedef.

Searching and Using Typedef Definitions

We now have a list of typedef definitions in a symbol table list. How do we use these definitions? We effectively have added new type keywords to our grammar, e.g.

```
FILE    *zin;
int32_t cost;
```

It just means that when we are parsing a type and we hit a keyword that we don't recognise, we can look that work up in the typedef list. So, we get to modify `parse_type()` again:

```
    case T_IDENT:
      type = type_of_typedef(Text, ctype);
      break;
```

Both `type` and `ctype` are returned by `type_of_typedef()` :

```
// Given a typedef name, return the type it represents
int type_of_typedef(char *name, struct symtable **ctype) {
  struct symtable *t;

  // Look up the typedef in the list
  t = findtypedef(name);
  if (t == NULL)
    fatals("unknown type", name);
  scan(&Token);
  *ctype = t->ctype;
  return (t->type);
}
```

Note that, as yet, I haven't written the code to be "recursive". For example, the current code won't parse this example:

```
typedef int FOO;
typedef FOO BAR;
BAR x;                  // x is of type BAR -> type FOO -> type int
```

But it does compile `tests/input68.c` :

```
int printf(char *fmt);

typedef int FOO;
FOO var1;

struct bar { int x; int y} ;
typedef struct bar BAR;
BAR var2;

int main() {
  var1= 5; printf("%d\n", var1);
  var2.x= 7; var2.y= 10; printf("%d\n", var2.x + var2.y);
  return(0);
}
```

with both `int` redefined as type `FOO` and a struct redefined as type `BAR` .

Conclusion and What's Next

In this part of our compiler writing journey, we added support for both enums and typedefs. Both were relatively easy to do, even though we did have to write a fair bit of parsing code for the enums. I guess I was spoiled when I could reuse the same parsing code for variable lists, struct member lists and union member lists!

The code to add typedefs was really nice and simple. I do need to add to code to follow typedefs of typedefs: that also should be simple.

In the next part of our compiler writing journey, I think it's time we bring in the C pre-processor. Now that we have structs, unions, enums and typedefs, we should be able to write a bunch of *header files* with definitions of some of the common Unix/Linux library functions. Then we will be able to include them in our source files and write some really useful programs. Next step