Part 15: Pointers, part 1

In this part of our compiler writing journey, I want to begin the work to add pointers to our language. In particular, I want to add this:

- Declaration of pointer variables
- Assignment of an address to a pointer
- Dereferencing a pointer to get the value it points at

Given that this is a work in progress, I'm sure I will implement a simplistic version that works for now, but later on I will have to change or extend it for to be more general.

New Keywords and Tokens

There are no new keywords this time, only two new tokens:

- '&', T_AMPER, and
- '&&', T_LOGAND

We don't need T_LOGAND yet, but I might as well add this code to `scan()` now:

```
    case '&':
      if ((c = next()) == '&') {
        t->token = T_LOGAND;
      } else {
        putback(c);
```

```
        t->token = T_AMPER;
      }
      break;
```

New Code for Types

I've added some new primitive types to the language (in `defs.h`):

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

We will have new unary prefix operators:

- '&' to get the address of an identifier, and
- '*' to dereference a pointer and get the value it points at.

The type of expression that each operator produces is different to the type that each works on. We need a couple of functions in `types.c` to make the type change:

```
// Given a primitive type, return
// the type which is a pointer to it
int pointer_to(int type) {
  int newtype;
  switch (type) {
    case P_VOID: newtype = P_VOIDPTR; break;
    case P_CHAR: newtype = P_CHARPTR; break;
    case P_INT:  newtype = P_INTPTR;  break;
    case P_LONG: newtype = P_LONGPTR; break;
    default:
      fatald("Unrecognised in pointer_to: type", type);
  }
  return (newtype);
}

// Given a primitive pointer type, return
// the type which it points to
int value_at(int type) {
  int newtype;
  switch (type) {
    case P_VOIDPTR: newtype = P_VOID; break;
    case P_CHARPTR: newtype = P_CHAR; break;
    case P_INTPTR:  newtype = P_INT;  break;
```

```
    case P_LONGPTR: newtype = P_LONG; break;
    default:
      fatald("Unrecognised in value_at: type", type);
  }
  return (newtype);
}
```

Now, where are we going to use these functions?

Declaring Pointer Variables

We want to be able to declare scalar variables and pointer variables, e.g.

```
char  a; char *b;
int   d; int  *e;
```

We already have a function `parse_type()` in `decl.c` that converts the type keyword to a type. Let's extend it to scan the following token and change the type if the next token is a '*':

```
// Parse the current token and return
// a primitive type enum value. Also
// scan in the next token
int parse_type(void) {
  int type;
  switch (Token.token) {
    case T_VOID: type = P_VOID; break;
    case T_CHAR: type = P_CHAR; break;
    case T_INT:  type = P_INT;  break;
    case T_LONG: type = P_LONG; break;
    default:
      fatald("Illegal type, token", Token.token);
  }

  // Scan in one or more further '*' tokens
  // and determine the correct pointer type
  while (1) {
    scan(&Token);
    if (Token.token != T_STAR) break;
    type = pointer_to(type);
  }

  // We leave with the next token already scanned
```

```
  return (type);
}
```

This will allow the programmer to try to do:

```
char *****fred;
```

This will fail because `pointer_to()` can't convert a P_CHARPTR to a P_CHARPTRPTR (yet). But the code in `parse_type()` is ready to do it!

The code in `var_declaration()` now quite happily parses pointer variable declarations:

```
// Parse the declaration of a variable
void var_declaration(void) {
  int id, type;

  // Get the type of the variable
  // which also scans in the identifier
  type = parse_type();
  ident();
  ...
}
```

Prefix Operators '*' and '&'

With declarations out of the road, let's now look at parsing expressions where '*' and '&' are operators that come before an expression. The BNF grammar looks like this:

```
prefix_expression: primary
    | '*' prefix_expression
    | '&' prefix_expression
    ;
```

Technically this allows:

```
x= ***y;
a= &&&b;
```

To prevent impossible uses of the two operators, we add in some semantic checking. Here's the code:

```
// Parse a prefix expression and return
// a sub-tree representing it.
struct ASTnode *prefix(void) {
  struct ASTnode *tree;
  switch (Token.token) {
    case T_AMPER:
      // Get the next token and parse it
      // recursively as a prefix expression
      scan(&Token);
      tree = prefix();

      // Ensure that it's an identifier
      if (tree->op != A_IDENT)
        fatal("& operator must be followed by an identifier");

      // Now change the operator to A_ADDR and the type to
      // a pointer to the original type
      tree->op = A_ADDR; tree->type = pointer_to(tree->type);
      break;
    case T_STAR:
      // Get the next token and parse it
      // recursively as a prefix expression
      scan(&Token); tree = prefix();

      // For now, ensure it's either another deref or an
      // identifier
      if (tree->op != A_IDENT && tree->op != A_DEREF)
        fatal("* operator must be followed by an identifier or *");

      // Prepend an A_DEREF operation to the tree
      tree = mkastunary(A_DEREF, value_at(tree->type), tree, 0);
      break;
    default:
      tree = primary();
  }
  return (tree);
}
```

We're still doing recursive descent, but we also put error checks in to prevent input mistakes. Right now, the limitations in `value_at()` will prevent more than one '*' operator in a row, but later on when we change `value_at()`, we won't have to come back and change `prefix()`.

Note that `prefix()` also calls `primary()` when it doesn't see a '*' or '&' operator. That allows us to change our existing code in `binexpr()`:

```
struct ASTnode *binexpr(int ptp) {
  struct ASTnode *left, *right;
  int lefttype, righttype;
  int tokentype;

  // Get the tree on the left.
  // Fetch the next token at the same time.
  // Used to be a call to primary().
  left = prefix();
  ...
}
```

New AST Node Types

Up in `prefix()` I introduced two new AST node types (declared in `defs.h`):

- A_DEREF: Dereference the pointer in the child node
- A_ADDR: Get the address of the identifier in this node

Note that the A_ADDR node isn't a parent node. For the expression `&fred`, the code in `prefix()` replaces the A_IDENT in the "fred" node with the A_ADDR node type.

Generating the New Assembly Code

In our generic code generator, `gen.c`, there are only a few new lines to `genAST()`:

```
    case A_ADDR:
      return (cgaddress(n->v.id));
    case A_DEREF:
      return (cgderef(leftreg, n->left->type));
```

The A_ADDR node generates the code to load the address of the `n->v.id` identifier into a register. The A_DEREF node take the pointer address in `lefreg`, and its associated type, and returns a register with the value at this address.

x86-64 Implementation

I worked out the following assembly output by reviewing the assembly code generated by other compilers. It might not be correct!

```
// Generate code to load the address of a global
// identifier into a variable. Return a new register
int cgaddress(int id) {
  int r = alloc_register();

  fprintf(Outfile, "\tleaq\t%s(%%rip), %s\n", Gsym[id].name, reglist[r]);
  return (r);
}

// Dereference a pointer to get the value it
// pointing at into the same register
int cgderef(int r, int type) {
  switch (type) {
    case P_CHARPTR:
      fprintf(Outfile, "\tmovzbq\t(%s), %s\n", reglist[r], reglist[r]);
      break;
    case P_INTPTR:
    case P_LONGPTR:
      fprintf(Outfile, "\tmovq\t(%s), %s\n", reglist[r], reglist[r]);
      break;
  }
  return (r);
}
```

The `leaq` instruction loads the address of the named identifier. In the section function, the `(%r8)` syntax loads the value that register `%r8` points to.

Testing the New Functinality

Here's our new test file, `tests/input15.c` and the result when we compile it:

```
int main() {
  char  a; char *b; char  c;
  int   d; int  *e; int   f;

  a= 18; printint(a);
  b= &a; c= *b; printint(c);

  d= 12; printint(d);
  e= &d; f= *e; printint(f);
  return(0);
}
```

```
$ make test15
cc -o comp1 -g -Wall cg.c decl.c expr.c gen.c main.c misc.c
   scan.c stmt.c sym.c tree.c types.c
./comp1 tests/input15.c
cc -o out out.s lib/printint.c
./out
18
18
12
12
```

I decided to change our test files to end with the `.c` suffix, now that they are actually C programs. I also changed the `tests/mktests` script to generate the *correct* results by using a "real" compiler to compile our test files.

Conclusion and What's Next

Well, we have the start of pointers implemented. They are not completely correct yet. For example, if I write this code:

```
int main() {
  int x; int y;
  int *iptr;
  x= 10; y= 20;
  iptr= &x + 1;
  printint( *iptr);
}
```

it should print 20 because `&x + 1` should address one `int` past `x`, i.e. `y`. This is eight bytes away from `x`. However, our compiler simply adds one to the address of `x`, which is incorrect. I'll have to work out how to fix this.

In the next part of our compiler writing journey, we will try to fix this problem. Next step