Part 19: Arrays, part 1

> *My lecturer for the first year of university was a Scotsman with a very heavy accent. Around the third or fourth week of first term, he began saying "Hurray!" a lot in class. It took me about twenty minutes to work out he was saying "array".*

So, we begin the work to add arrays to the compiler in this part of the journey. I sat down and wrote a small C program to see what sort of functionality I should try to implement:

```
int ary[5];               // Array of five int elements
int *ptr;                 // Pointer to an int

ary[3]= 63;               // Set ary[3] (lvalue) to 63
ptr   = ary;              // Point ptr to base of ary
// ary= ptr;              // error: assignment to expression with array type
ptr   = &ary[0];          // Also point ptr to base of ary, ary[0] is lvalue
ptr[4]= 72;               // Use ptr like an array, ptr[4] is an lvalue
```

Arrays are *like* pointers in that we can dereference both a pointer and an array with the "[ ]" syntax to get access to a specific element. We can use the array's name as a "pointer" and save the array's base into a pointer. We can get the address of an element in the array. But one thing we can't do is "overwrite" the base of an array with a pointer: the elements of the array are mutable, but the base address of the array is not mutable.

In this part of the journey, I'll add in:

- declarations of array with a fixed size but no initialisation list
- array indexes as rvalues in an expression
- array indexes as an lvalue in an assignment

I also won't implement more than one dimension in each array.

Parentheses in Expressions

At some point I want to try this out: `*(ptr + 2)` which should end up being the same as `ptr[2]`. But we haven't allowed parentheses in expressions yet, so now it's time to add them.

C Grammar in BNF

On the web there is a page with the BNF Grammar for C written by Jeff Lee in 1985. I like to reference it to give me ideas and to confirm that I'm not making too many mistakes.

One thing to note is that, instead of implemnting the priority of the binary expression operators in C, the grammar uses recursive definitions to make the priorities explicit. Thus:

```
additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        ;
```

shows that we descend into "multiplicative_expression" while we are parsing an "additive_expression", thus giving the '*' and '/' operators a higher precedence than the '+' and '-' operators.

Right at the top of the expression precedence hierarchy is:

```
primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;
```

We already have a `primary()` function which is called to find T_INTLIT and T_IDENT tokens, and this conforms to Jeff Lee's C grammar. It's thus the perfect place to add the parsing of parentheses in expressions.

We already have T_LPAREN and T_RPAREN as tokens in our language, so there is no work to be done in the lexical scanner.

Instead, we simply modify `primary()` to do the extra parsing:

```
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  ...

  switch (Token.token) {
  case T_INTLIT:
  ...
  case T_IDENT:
  ...
  case T_LPAREN:
    // Beginning of a parenthesised expression, skip the '('.
    // Scan in the expression and the right parenthesis
    scan(&Token);
    n = binexpr(0);
    rparen();
    return (n);

    default:
    fatald("Expecting a primary expression, got token", Token.token);
  }

  // Scan in the next token and return the leaf node
  scan(&Token);
  return (n);
}
```

And that's it! Just a few extra lines to add parentheses in expression. You'll notice that I explicitly call `rparen()` in the new code and return instead of breaking out of the switch statement. If the code had left the switch statement, the `scan(&Token);` before the final return would not strictly enforce the requirement for a ')' token to match the opening '(' token.

The `test/input19.c` test checks that parentheses are working:

```
  a= 2; b= 4; c= 3; d= 2;
  e= (a+b) * (c+d);
  printint(e);
```

and it should print out 30, i.e. `6 * 5`.

Symbol Table Changes

We have scalar variables (with only one value) and functions in our symbol table. It's time to add arrays. Later on, we'll want to get the number of elements in each array with the `sizeof()` operator. Here are the changes in `defs.h`:

```
// Structural types
enum {
  S_VARIABLE, S_FUNCTION, S_ARRAY
};

// 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 endlabel;                 // For S_FUNCTIONs, the end label
  int size;                     // Number of elements in the symbol
};
```

For now, we will treat arrays as pointers, and so the type for an array is "pointer to" something, e.g. "pointer to int" if the elements in the array are `int` s. We also need to add one more argument to `addglob()` in `sym.c`:

```
int addglob(char *name, int type, int stype, int endlabel, int size) {
  ...
}
```

Parsing Array Declarations

For now, I'm only going to allow declarations of arrays with a size. The BNF grammar for variable declarations is now:

```
 variable_declaration: type identifier ';'
        | type identifier '[' P_INTLIT ']' ';'
        ;
```

So we need to see what token is next in `var_declaration()` in `decl.c` and process either a scalar variable declaration or an array declaration:

```
// Parse the declaration of a scalar variable or an array
// with a given size.
// The identifier has been scanned & we have the type
void var_declaration(int type) {
  int id;

  // Text now has the identifier's name.
  // If the next token is a '['
  if (Token.token == T_LBRACKET) {
    // Skip past the '['
    scan(&Token);

    // Check we have an array size
    if (Token.token == T_INTLIT) {
      // Add this as a known array and generate its space in assembly.
      // We treat the array as a pointer to its elements' type
      id = addglob(Text, pointer_to(type), S_ARRAY, 0, Token.intvalue);
      genglobsym(id);
    }

    // Ensure we have a following ']'
    scan(&Token);
    match(T_RBRACKET, "]");
  } else {
    ...      // Previous code
  }

    // Get the trailing semicolon
  semi();
}
```

I think that's pretty straight-forward code Later on, we'll add initialisation lists to the declaration of arrays.

Generating the Array Storage

Now that we know the size of the array, we can modify `cgglobsym()` to allocate this space in the assembler:

```
void cgglobsym(int id) {
  int typesize;
  // Get the size of the type
  typesize = cgprimsize(Gsym[id].type);

  // Generate the global identity and the label
```

```
    fprintf(Outfile, "\t.data\n" "\t.globl\t%s\n", Gsym[id].name);
    fprintf(Outfile, "%s:", Gsym[id].name);

    // Generate the space
    for (int i=0; i < Gsym[id].size; i++) {
      switch(typesize) {
        case 1: fprintf(Outfile, "\t.byte\t0\n"); break;
        case 4: fprintf(Outfile, "\t.long\t0\n"); break;
        case 8: fprintf(Outfile, "\t.quad\t0\n"); break;
        default: fatald("Unknown typesize in cgglobsym: ", typesize);
      }
    }
  }
```

With this in place, we can now declare arrays such as:

```
    char a[10];
    int  b[25];
    long c[100];
```

Parsing Array Indexes

In this part I don't want to get too adventurous. I only want to get basic array indexes as rvalues and lvalues to work. The `test/input20.c` program has the functionality that I want to achieve:

```
int a;
int b[25];

int main() {
  b[3]= 12; a= b[3];
  printint(a); return(0);
}
```

Back in the BNF grammar for C, we can see that array indexes have *slightly* lower priority than parentheses:

```
primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;
```

```
postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
          ...
```

But for now, I'll parse array indexes also in the `primary()` function. The code to do the semantic analysis ended up being big enough to warrant a new function:

```
static struct ASTnode *primary(void) {
  struct ASTnode *n;
  int id;


  switch (Token.token) {
  case T_IDENT:
    // This could be a variable, array index or a
    // function call. Scan in the next token to find out
    scan(&Token);

    // It's a '(', so a function call
    if (Token.token == T_LPAREN) return (funccall());

    // It's a '[', so an array reference
    if (Token.token == T_LBRACKET) return (array_access());
```

And here is the `array_access()` function:

```
// Parse the index into an array and
// return an AST tree for it
static struct ASTnode *array_access(void) {
  struct ASTnode *left, *right;
  int id;

  // Check that the identifier has been defined as an array
  // then make a leaf node for it that points at the base
  if ((id = findglob(Text)) == -1 || Gsym[id].stype != S_ARRAY) {
    fatals("Undeclared array", Text);
  }
  left = mkastleaf(A_ADDR, Gsym[id].type, id);

  // Get the '['
  scan(&Token);

  // Parse the following expression
  right = binexpr(0);
```

```
    // Get the ']'
    match(T_RBRACKET, "]");

    // Ensure that this is of int type
    if (!inttype(right->type))
      fatal("Array index is not of integer type");

    // Scale the index by the size of the element's type
    right = modify_type(right, left->type, A_ADD);

    // Return an AST tree where the array's base has the offset
    // added to it, and dereference the element. Still an lvalue
    // at this point.
    left = mkastnode(A_ADD, Gsym[id].type, left, NULL, right, 0);
    left = mkastunary(A_DEREF, value_at(left->type), left, 0);
    return (left);
  }
```

For the array `int x[20];` and the array index `x[6]`, we need to scale the index (6) by the size of `int` s (4), and add this to the address of the array base. Then this element has to be dereferenced. We leave it marked as an lvalue, because we could be trying to do:

```
x[6] = 100;
```

If it does become an rvalue, then `binexpr()` will set the `rvalue` flag in the A_DEREF AST node.

The Generated AST Trees

Going back to our test program `tests/input20.c`, the code that will produce AST trees with array indexes are:

```
b[3]= 12; a= b[3];
```

Running `comp1 -T tests/input20.c`, we get:

```
  A_INTLIT 12
A_WIDEN
    A_ADDR b
      A_INTLIT 3    # 3 is scaled by 4
    A_SCALE 4
  A_ADD             # and then added to b's address
```

```
    A_DEREF             # and derefenced. Note, stll an lvalue
  A_ASSIGN

        A_ADDR b
          A_INTLIT 3    # As above
        A_SCALE 4
      A_ADD
    A_DEREF rval        # but the dereferenced address will be an rvalue
    A_IDENT a
  A_ASSIGN
```

Other Minor Parse Changes

There are a couple of minor changes to the parser in `expr.c` which took me a while to debug. I needed to be more stringent with the input to the operator precedence lookup function:

```
// Check that we have a binary operator and
// return its precedence.
static int op_precedence(int tokentype) {
  int prec;
  if (tokentype >= T_VOID)
    fatald("Token with no precedence in op_precedence:", tokentype);
  ...
}
```

Until I got the parsing right, I was sending a token not in the precedence table, and `op_precedence()` was reading past the end of the table. Oops! Don't you just love C?!

The other change is that, now that we can use expressions as array indexes (e.g. `x[ a+2 ]`), we have to expect the ']' token can end an expression. So, at the end of `binexpr()`:

```
    // Update the details of the current token.
    // If we hit a semicolon, ')' or ']', return just the left node
    tokentype = Token.token;
    if (tokentype == T_SEMI || tokentype == T_RPAREN
        || tokentype == T_RBRACKET) {
      left->rvalue = 1;
      return (left);
    }
  }
```

Changes to the Code Generator

There are none. We had all the necessary components in our compiler already: scaling integer values, obtaining the address of a variable etc. For our test code:

```
b[3]= 12; a= b[3];
```

we generate the x86-64 assembly code:

```
        movq    $12, %r8
        leaq    b(%rip), %r9    # Get b's address
        movq    $3, %r10
        salq    $2, %r10        # Shift 3 by 2, i.e. 3 * 4
        addq    %r9, %r10       # Add to b's address
        movq    %r8, (%r10)     # Save 12 into b[3]

        leaq    b(%rip), %r8    # Get b's address
        movq    $3, %r9
        salq    $2, %r9         # Shift 3 by 2, i.e. 3 * 4
        addq    %r8, %r9        # Add to b's address
        movq    (%r9), %r9      # Load b[3] into %r9
        movl    %r9d, a(%rip)   # and store in a
```

Conclusion and What's Next

The parsing changes to add basic array declarations and array expressions (in terms of dealing with the syntax) were quite easy to do. What I found difficult was getting the AST tree nodes correct to scale, add to the base address, and set as lvalue/rvalue. Once this was right, the existing code generator produces the right assembly output.

In the next part of our compiler writing journey, we'll add character and string literals to our language and find a way to print them out. Next step