Part 62: Code Cleanup

This version of the compiler is essentially the same as in part 60. I am using this part to fix up comments, fix up bugs, do a bit of code cleanup, rename some functions and variables etc.

Some Small Bugfixes

For the changes to the compiler that I'm planning, I need to be able to put structs into structs. Therefore, I should be able to do:

```
printf("%d\n", thing.member1.age_in_years);
```

where `thing` is a struct, but it has a `member1` which is of type struct. To do this, we need to find the offset of `member1` from the base of `thing`, then find the offset of `age_in_years` from the previous offset.

However, the code to do this expects the things on the left-hand side of the '.' token to be a variable which has a symbol table entry and thus a fixed location in memory. We need to fix this to deal with the situation where the left-hand side of the '.' token is an offset that has already been calculated.

Fortunately, this was quite easy to do. We don't have to change the parser code, but let's look at what is already there. In `member_access()` in `expr.c`:

```
// Check that the left AST tree is a struct or union.
// If so, change it from an A_IDENT to an A_ADDR so that
// we get the base address, not the value at this address.
if (!withpointer) {
  if (left->type == P_STRUCT || left->type == P_UNION)
    left->op = A_ADDR;
```

We mark the left-hand AST tree as A_ADDR (instead of A_IDENT) to say that we need the base address of it, not the value at this address.

Now we need to fix the code generation. When we get an A_ADDR AST node, we either have a variable whose address we need (e.g. `thing` in `thing.member1`), or our child tree has the pre-calculated offset (e.g. the offset of `member1` in `member1.age_in_years). So in` genAST() `in` gen.c`, we do:

```
case A_ADDR:
  // If we have a symbol, get its address. Otherwise,
  // the left register already has the address because
  // it's a member access
```

That should be all, but we have one more fix. The code to work out the alignment of types doesn't deal with structs inside structs, only scalar types inside structs. So, I've modified `cgalign()` in `cg.c` as follows:

```
// Given a scalar type, an existing memory offset
// (which hasn't been allocated to anything yet)
// and a direction (1 is up, -1 is down), calculate
// and return a suitably aligned memory offset
// for this scalar type. This could be the original
// offset, or it could be above/below the original
int cgalign(int type, int offset, int direction) {
  int alignment;

  // We don't need to do this on x86-64, but let's
  // align chars on any offset and align ints/pointers
  // on a 4-byte alignment
  switch (type) {
  case P_CHAR:
    break;
  default:
      // Align whatever we have now on a 4-byte alignment.
      // I put the generic code here so it can be reused elsewhere.
      alignment = 4;
      offset = (offset + direction * (alignment - 1)) & ~(alignment - 1);
    }
    return (offset);
  }
```

Everything but P_CHAR gets aligned on a 4-byte alignment, including structs and unions.

Known but Unfixed Bugs

Now that this Github repository is up and has gained some attention, several people have reported bugs and misfeatures. The list of open and closed issues is here: https://github.com/DoctorWkt/acwj/issues. If you spot any bugs or misfeatures, feel free to report them. However, I can't promise I'll get time to fix them all!

What's Next

I've been reading up on register allocation, and I think I'll add a linear scan register allocation mechanism to the compiler. To do this, though, I need to add an intermediate representation stage. This will be the goal for the next few stages, but so far I haven't done anything concrete. Next step