Part 23: Local Variables

I've just implemented local variables on the stack following the design ideas I described in the previous part of our compiler writing journey, and it all went fine. Below, I will outline the actual code changes.

Symbol Table Changes

We start with the changes to the symbol table as these are central to having two variable scopes: global and local. The structure of the symbol table entries is now (in `defs.h`):

```
// Storage classes
enum {
        C_GLOBAL = 1,           // Globally visible symbol
        C_LOCAL                 // Locally visible symbol
};

// 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 class;                    // Storage class for the symbol
  int endlabel;                 // For functions, the end label
  int size;                     // Number of elements in the symbol
  int posn;                     // For locals,the negative offset
```

```
                                // from the stack base pointer
};
```

with the `class` and `posn` fields added. As described in the last part, the `posn` is negative and holds an offset from the stack base pointer, i.e. the local variable is stored on the stack. In this part, I've only implemented local variables, not parameters. Also note that we now have symbols marked C_GLOBAL or C_LOCAL.

The symbol table's name has also changed, along with the indexed into it (in `data.h`):

```
extern_ struct symtable Symtable[NSYMBOLS];     // Global symbol table
extern_ int Globs;                              // Position of next free globa
extern_ int Locls;                              // Position of next free local
```

Visually, the global symbols are stored in the left-hand side of the symbol table with `Globs` pointing at the next free global symbol slot and `Locls` pointing at the next free local symbol slot.

```
0xxxx.....................................xxxxxxxxxxxxxNSYMBOLS-1
     ^                                     ^
     |                                     |
   Globs                                 Locls
```

In `sym.c` as well as the existing `findglob()` and `newglob()` functions to find or allocate a global symbol, we now have `findlocl()` and `newlocl()`. They have code to detect a collision between `Globs` and `Locls`:

```
// Get the position of a new global symbol slot, or die
// if we've run out of positions.
static int newglob(void) {
  int p;

  if ((p = Globs++) >= Locls)
    fatal("Too many global symbols");
  return (p);
}

// Get the position of a new local symbol slot, or die
// if we've run out of positions.
static int newlocl(void) {
  int p;
```

```
  if ((p = Locls--) <= Globs)
    fatal("Too many local symbols");
  return (p);
}
```

There is now a generic function `updatesym()` to set all the fields in a symbol table entry. I won't give the code because it simply sets each field one at a time.

The `updatesym()` function is called by `addglobl()` and `addlocl()`. These first try to find an existing symbol, allocate a new one if not found, and call `updatesym()` to set the values for this symbol. Finally, there is a new function, `findsymbol()`, that searches for a symbol in both local and global sections of the symbol table:

```
// Determine if the symbol s is in the symbol table.
// Return its slot position or -1 if not found.
int findsymbol(char *s) {
  int slot;

  slot = findlocl(s);
  if (slot == -1)
    slot = findglob(s);
  return (slot);
}
```

Throughout the rest of the code, the old calls to `findglob()` have been replaced with calls the `findsymbol()`.

Changes to Declaration Parsing

We need to be able to parse both global and local variable declarations. The code to parse them is (for now) the same, so I added a flag to the function:

```
void var_declaration(int type, int islocal) {
    ...
      // Add this as a known array
      if (islocal) {
        addlocl(Text, pointer_to(type), S_ARRAY, 0, Token.intvalue);
      } else {
        addglob(Text, pointer_to(type), S_ARRAY, 0, Token.intvalue);
      }
    ...
    // Add this as a known scalar
    if (islocal) {
      addlocl(Text, type, S_VARIABLE, 0, 1);
```

```
    } else {
      addglob(Text, type, S_VARIABLE, 0, 1);
    }
    ...
}
```

There are two calls to `var_declaration()` in our compiler at present. This one in `global_declarations()` in `decl.c` parses global variable declarations:

```
void global_declarations(void) {
      ...
      // Parse the global variable declaration
      var_declaration(type, 0);
      ...
}
```

This one in `single_statement()` in `stmt.c` parses local variable declarations:

```
static struct ASTnode *single_statement(void) {
  int type;

  switch (Token.token) {
    case T_CHAR:
    case T_INT:
    case T_LONG:

      // The beginning of a variable declaration.
      // Parse the type and get the identifier.
      // Then parse the rest of the declaration.
      type = parse_type();
      ident();
      var_declaration(type, 1);
   ...
  }
  ...
}
```

Changes to the x86-64 Code Generator

As always, many of the `cgXX()` functions in the platform-specific code in `cg.c` are exposed to the rest of the compiler as `genXX()` functions in `gen.c`. That's going to be the case here. So while I only mention the `cgXX()` functions, don't forget that there are often matching `genXX()` functions.

For each local variable, we need to allocate a position for it and record this in the symbol table's `posn` field. Here is how we do it. In `cg.c` we have a new static variable and two functions to manipulate it:

```
// Position of next local variable relative to stack base pointer.
// We store the offset as positive to make aligning the stack pointer easier
static int localOffset;
static int stackOffset;

// Reset the position of new local variables when parsing a new function
void cgresetlocals(void) {
  localOffset = 0;
}

// Get the position of the next local variable.
// Use the isparam flag to allocate a parameter (not yet XXX).
int cggetlocaloffset(int type, int isparam) {
  // Decrement the offset by a minimum of 4 bytes
  // and allocate on the stack
  localOffset += (cgprimsize(type) > 4) ? cgprimsize(type) : 4;
  return (-localOffset);
}
```

For now, we allocate all local variables on the stack. They are aligned with a minimum of 4 bytes between each one. For 64-bit integers and pointers, that's 8-bytes for each variable, though.

> I know, in the past, that multi-byte data items had to be properly aligned in memory or the CPU would fault. It seems that, at least for x86-64, there is no need to align data items.

> However, the stack pointer on the x86-64 *does* have to be properly aligned before a function call. In "Optimizing Subroutines in Assembly Language" by Agner Fog, page 30, he notes that "The stack pointer must be aligned by 16 before any CALL instruction, so that the value of RSP is 8 modulo 16 at the entry of a function."

> This means that, as part of the function preamble, we need to set `%rsp` to a correctly aligned value.

`cgresetlocals()` is called in `function_declaration()` once we have added the function's name to the symbol table but before we start parsing the local variable declarations. This sets `localOffset` back to zero.

We saw that `addlocl()` is called with a new local scalar or local array is parsed. `addlocl()` calls `cggetlocaloffset()` with the type of the new variable. This decrements the offset from the stack base pointer by an approriate amount, and this offset is stored in the `posn` field for the symbol.

Now that we have the symbol's offset from the stack base pointer, we now need to modify the code generator so that, when we are accessing a local variable instead of a global variable, we output an offset to `%rbp` instead of naming a global location.

Thus, we now have a `cgloadlocal()` function which is nearly identical to `cgloadglob()` except that all `%s(%%rip)` format strings to print `Symtable[id].name` are replaced with `%d(%%rbp)` format strings to print `Symtable[id].posn`. In fact, if you search for `Symtable[id].posn` in `cg.c`, you will spot all of these new local variable references.

Updating the Stack Pointer

Now that we are using locations on the stack, we had better move the stack pointer down below the area which holds our local variables. Thus, we need to modify the stack pointer in our function preamble and postamble:

```
// Print out a function preamble
void cgfuncpreamble(int id) {
  char *name = Symtable[id].name;
  cgtextseg();

  // Align the stack pointer to be a multiple of 16
  // less than its previous value
  stackOffset= (localOffset+15) & ~15;

  fprintf(Outfile,
          "\t.globl\t%s\n"
          "\t.type\t%s, @function\n"
          "%s:\n" "\tpushq\t%%rbp\n"
          "\tmovq\t%%rsp, %%rbp\n"
          "\taddq\t$%d,%%rsp\n", name, name, name, -stackOffset);
}

// Print out a function postamble
void cgfuncpostamble(int id) {
  cglabel(Symtable[id].endlabel);
  fprintf(Outfile, "\taddq\t$%d,%%rsp\n", stackOffset);
  fputs("\tpopq %rbp\n" "\tret\n", Outfile);
}
```

Remember that `localOffset` is negative. So we add a negative value in the function preamble, and add a negative negative value in the function postamble.

Testing the Changes

I think that is the bulk of the changes to add local variables to our compiler. The test program `tests/input25.c` demonstrates the storage of local variables on the stack:

```
int a; int b; int c;

int main()
{
  char z; int y; int x;
  x= 10;  y= 20; z= 30;
  a= 5;   b= 15; c= 25;
}
```

Here is the annotated assembly output:

```
        .data
        .globl  a
a:      .long   0                       # Three global variables
        .globl  b
b:      .long   0
        .globl  c
c:      .long   0

        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        addq    $-16,%rsp               # Lower stack pointer by 16
        movq    $10, %r8
        movl    %r8d, -12(%rbp)         # z is at offset -12
        movq    $20, %r8
        movl    %r8d, -8(%rbp)          # y is at offset -8
        movq    $30, %r8
        movb    %r8b, -4(%rbp)          # x is at offfset -4
        movq    $5, %r8
        movl    %r8d, a(%rip)           # a has a global label
        movq    $15, %r8
        movl    %r8d, b(%rip)           # b has a global label
        movq    $25, %r8
```

```
        movl    %r8d, c(%rip)           # c has a global label
        jmp     L1
L1:
        addq    $16,%rsp                # Raise stack pointer by 16
        popq    %rbp
        ret
```

Finally, a `$ make test` demonstrates that the compiler passes all previous tests.

Conclusion and What's Next

I thought implementing local variables was going to be tricky, but after doing some thinking about the design of a solution, it turned out to be easier than I expected. Somehow I suspect the next step will be the tricky one.

In the next part of our compiler writing journey, I will attempt to add function arguments and parameters to our compiler. Wish me luck! Next step