Part 22: Design Ideas for Local Variables and Function Calls

This is going to be first first part of our compiler writing journey where I don't introduce any new code. This time, I need to step back from the coder's keyboard and take a big-picture view. This will give me a chance to think about how I'm going to implement local variables (in one part) and then function arguments & parameters (in the next part).

Both of these steps are going to involve some significant additions and changes to our existing compiler. We also have to deal with new concepts like *stack frames* and *register spills*, which so far I've omitted.

Let's start by identifying what new functionality we want to add to the compiler.

What Functionality Do We Want

Local and Global Variable Scopes

Right now, all our variables are globally visible to all functions. We want to add a local scope for variables, so that each function has its own variables that cannot be seen by other functions. Moreover, in the case of recursive functions, each instance of the same function gets its own local variables.

However, I only want to add two scopes: *local* and *global*. C actually creates a new scope for every compound statement. In the following example, there are three different `a` variables in three different scopes:

```
#include <stdio.h>
int a = 2;              // Global scope

int main()
{
  int a= 5;             // Local scope
  if (a > 2) {
    int a= 17;          // Third scope
    printf("%d\n", a);  // Print 17
  }
  printf("%d\n", a);    // Print 5
  return(0);
}
```

I'm not going to support the third, inner, scope. Two will be enough!

Function Parameters as Local Variables

We also need to support the declaration of zero or more *parameters* to a function, and these need to be treated as variables local to the instance of that function.

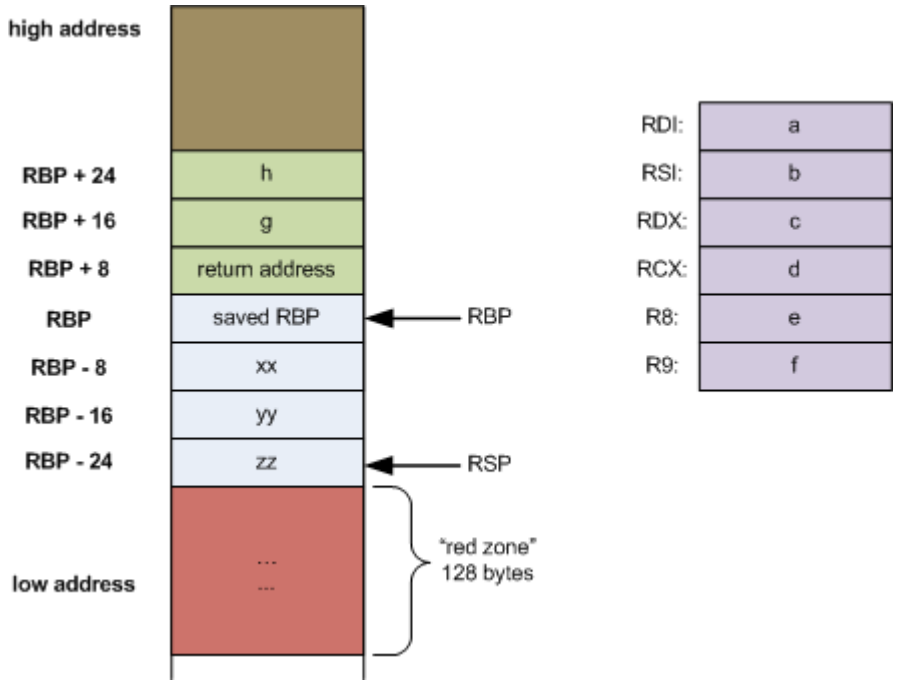
C functions are "call by value": the argument values in the caller of a function are copied into the function's parameters so that the called function can use and modify them.

Introducing the Stack

To create a local scope for multiple instances of the same function, and to provide a place to store the function's parameters, we need a *stack*. At this point, if you don't know much about stacks, you should do a bit of background reading on them. I'd start with this Wikipedia article on call stacks.

Given that one of the hardware architectures that we support is the Intel x86-64 architecture running Linux, we are going to have to implement the function call mechanism on this architecture. I found this great article by Eli Bendersky on the stack frame layout on x86-64. This is a document that you will definitely need to read before continuing on with this document! As Eli's article is in the public domain, I'm reproducing his picture of the stack frame and the parameters in registers below for the function

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx, yy, zz;
    ...
}
```



Essentially, on the x86-64 architecture, the values of some parameters will be passed in registers, and some parameter values will be pushed onto the stack. All our local variables will be on the stack but below the stack base pointer.

At the same time, we want our compiler to be portable to different architectures. So, we will need to support a general function parameter framework for different architectures which use the only the stack, only registers or a combination of both.

Spilling Registers

Something that I have ignored so far and not implemented yet is register spilling. We need to spill some or all the registers that we have allocated for several reasons:

- We have run out of registers to allocate as there is only a fixed number of registers. We can spill a register onto the stack so that it is free to allocate.
- We need to spill all our allocated register, and all registers with parameters, onto the stack before a function call. This frees them up so they can be used by the called function.

On a function call return, we will need to unspill the registers to get the values that we need back. Similarly, if we've spilled a register to make it free, then we need to unspill its old value and reallocate it when it becomes free again.

Static Variables

While not on the list of things to implement immediately, at some point I'll need to allocate static variables. There will be some naming issues here for local static variables, but I'll try to keep this in the back of my mind as I implement all of the immediate ideas.

Initialising Variables

We should allow variables to be initialised when they are declared. For global variables, we can definitely initialise them to a constant value, e.g. `int x= 7;` but not to an expression as we don't have a function context to run the initialisation code in.

However, we should be able to do local variable initialisation, e.g. `int a= 2, b= a+5;` as we can insert the initialisation code for the variable at the start of the function code.

Ideas and Implementation

OK, so these are the ideas and issues that are bubbling around in my designer's mind at this time. Here's how I think I'm going to implement some of them.

Local Symbols

Let's start with the differentiation between local and global variables. The globals have to be visible to all functions, but the locals are only visible to one function.

SubC uses the one symbol table to store information about both local and global variables. The global variables are allocated at one end and the local variables are stored at the other. There is code to ensure there is no collision between the two ends in the middle. I like this idea, as we then have a single set of unique symbol slot numbers for every symbol, regardless of its scope.

In terms of prioritising local symbols over global symbols, we can search the local end of the symbol table first and, if we don't find a symbol, we can then search through the global end. And, once we finish parsing a function, we can simply wipe the local end of the symbol table.

Storage Classes

C has the concept of storage classes, and we'll have to implement at least some of these classes. SubC implements several of the storage classes:

```
/* storage classes */
enum {
        CPUBLIC = 1,            // publicly visible symbol
        CEXTERN,                // extern symbol
        CSTATIC,                // static symbols in global context
        CLSTATC,                // static symbols in local context
        CAUTO,                  // non-static local identifiers
        CSPROTO,                // function prototype
        CMEMBER,                // field of a struct/union
        CSTCDEF                 // unused
};
```

for each symbol in the symbol table. I think I can modify and use this. But I'll probably support fewer storage class types.

Function Prototypes

Every function has a *prototype*: the number and type of each parameter that it has. We need these to ensure the arguments to a function call matches the types and number of function parameters.

Somewhere I will need to record the parameter list and types for each function. We can also support the declaration of a function's prototype before the actual declaration of the function itself.

Now, where are we going to store this? I could create a separate data structure for function prototypes. I don't want to support two-dimensional arrays in our language, but we will need a list of primitive types for each function.

So, my idea is this. We already have S_FUNCTION as the type for our existing symbol table elements. We can have a "number of parameters" field in each symbol table entry to store the number of parameters that the function has. We can then immediately follow this symbol with the symbol table entries for each function parameter.

When we are parsing the function's parameter list, we can add the parameters in the global symbol section to record the function's prototype. At the same time, we can also add the parameters as entries in the local symbol section, as they will be used as local variables by the function itself.

When we need to determine if the list of arguments to a function call matches the function's prototype, we can find the function's global symbol table entry and then compare the following entries in the symbol table to the argument list.

Finally, when doing a search for a global symbol, we can easily skip past the parameter entries for a function by loading the function's "number of parameters" field and skip this many symbol table entries.

Keeping Parameters in Registers: Not Possible

I'm actually writing this section after trying to implement the above, so I've come back to revisit the design a bit. I thought that we would be able to keep the parameters passed as registers in their registers: this would make access to them faster and keep the stack frame smaller. But this isn't always possible for this reason. Consider this code:

```
void myfunction(int a) {        // a is a parameter in a register
  int b;                        // b is a local variable on the stack

  // Call a function to update a and b
  b= function2(&a);
}
```

If the `a` parameter is in a register, we won't be able to get its address with the `&` operator. Therefore, we'll have to copy it into memory somewhere. And, given that parameters are variables local to the function, we will need to copy it to the stack.

For a while I had ideas of walking the AST looking for which parameters in the tree needed to have real addresses, but then I remembered that I'm following the KISS principle: keep it simple, stupid! So I will copy all parameters out of registers and onto the stack.

Location of Local Variables

How are we going to determine where a parameter or local variable is on the stack, once they have been copied or placed there? To do this, I will add a `posn` field into each local symbol table entry. This will indicate the offset of the variable below the frame base pointer.

Looking at the BNF Grammar for C, the function declaration list (i.e. the list of function parameters) comes before the declaration list for the local variables, and this comes before the statement list.

This means that, as we parse the parameters and then the local variables, we can determine at what position they will be on the stack before we get to parse the statement list.

Conclusion and What's Next

I think that's about all I want to do in terms of design before I start on the next parts of our compiler writing journey. I'll tackle local variables by themselves in the next part, and try to add in function calls and parameters in the following part. But it might take three or more steps to get all of the new proposed features implemented. We'll see. Next step