General Links

- Dennis Ritchie
- C6T

External declarations

Overview

External declarations are derived from the abstract syntax tree (AST) build-up during parsing of C-code. The nodes are several and complex in hierarchy. The implementation of the AST is in **CCast.py**. Nodes dealing with declarations have a **decl** method, that will call connected nodes decl-methods and aggregate information into a **CCDecl**-object (or several). The CCDecl-class is implemented in **CCconf.py** and is used as a intermediary object between the AST and a symbol table. The attributes to a CCDecl-object is added dynamically when traversing the AST. An important attribute is **ctx**, that helps to identify which type of declaration we are dealing with.

Nevertheless, declarations can be complex beasts with many permutations of 'basic' declarations. The various types of declarations and how to handle them are described below as reference to the implementations made in **CCSymbols.py**

These types of declarations are identified:

- Variables, such as **int a**
- Arrays, such as **int a[10]**
- Structures, such as **struct a { int b; int c; } d;**
- Combinations and additional attributes to above declaration type, such as
    - initializers: **int a 1;**
    - array of structures: __struct a { int b; int c; } d[10];::
    - etc

A symbol table have different entries depending on which variant of declaration we are dealing with. At the end of processing a declaration through a CCDecl-object, we should have added a symbol using its **name** as key. We will also have allocated memory for the declaration (implementation in **CCMemory.py**), accurate in position and size in memory (using a PDP 11/40 architecture; little endian, 2 bytes integers etc **-** see **CCMemory.py**). Memory positions is needed when we deal with declarations using pointers.

We distinguish between various types of declarations using the ctx-attribute in the CCDecl-object:

- Variables: **ctx is ['id']**
- Arrays: **ctx is ['id', 'array']**
- Structures: **ctx is ['id', 'array', 'struct_specifier']**

Below are details on the logic for different types of declarations.

Symbolic constans

No declaration as such, simple **#define CONST 1**, any **CONST** is replaced by 1 during compile time. Note that **#define** can have constant expressions, such as **#define XTYPE (03<<3)**, which needs to be evaluated to a constant during compilation.

Constants are stored in symbol table using its symbolic name as key.

Variables

A variable declaration has:

- **type**
  - basic type (int, long, char, float, double)
  - reference/pointer type (int *a[1] - array with one element, pointer to int)
- **name**
- **initializer** (optional, see below)

Note that no check is made of duplicate names, an identifier might be declared several times. The last occurrence will overwrite previous entries in symbol table.

An initializer of a variable can be different kinds, depending on the type of variable:

Basic type

1. **int a 1;**
   - a is 2 bytes in memory, initialized to 1
2. **char c 'a';**
   - c is 1 byte in memory, initialized to 97 (ord(a))
3. **char c 97;**
   - c is 1 byte in memory, initialized to 97, same as above

Pointer type

4. **int *a 1;**
   - a is 2 bytes in memory (type 'pointer') initialized to memory position 1
5. **char *c 'a';**
   - c is 2 bytes in memory (type 'pointer') initialized to a memory position 'a' (97)
6. **char *c 97;**
   - same as above

Cases 4 - 6 is semantically meaningless, but syntactically Ok, but we can't distinguish them from case 8.

7. **char *c "abc";**
   - c is 2 bytes in memory (type 'pointer') referring to a memory position for 'abc' (97 98 99 0 - 4 bytes). A null byte (\0) is added at the end of the string. Thus, the extent for the string is 3 + 1 = 4 bytes in memory.
8. **int *a &b;**
   - a is 2 bytes in memory (type 'pointer') referring to a memory position for for b. b has been previously declared and should be of type int, a is initialized to the memory position for b.

9. **int b[2] { 1, 2 }; int *a &b[1];**
   - a is a pointer that is initialized to the address of the second element of the array b. NOT IMPLEMENTED yet, see CCast::Initializer, need to implement PostfixExpression.eval()

List

When initializer is **{ constant expression }**

10. **int a { 1 + 1 };**
    - During parsing the constant expression will be a BinOpExpression in AST. When evaluating this expression (when creating declaration object), it will be evaluated to 2. Hence, the initializer will be 2. Same thing applies if the operators would be a constant or variable (previously declared), such as { 1 + b }, evaluation of BinOpExpression will resolve b into a value. Hence, such expression will be a constant. Thus, the initializer will be a one element list with a value.
11. **int *b { a };**
    - same case as 2.e above, different syntax. initializer will be a list with one element, mempos for 'a'. Thus, 'a' is already resolved to its address/mempos by the AST Initializer object and will be a number, not a symbolic constant.
12. **char *c { "ABC" };**
    - same case as 2.d above, different syntax. c is a pointer to the string ABC

Arrays

An array object has:

Type

- basic type (**int, char, float, double**)
- reference/pointer type (int ***a[1] -** array with one element, pointer to int)
- composite type (**struct**)

Note that an array can have structs as elements, for example **struct a b[10]**. This will have ctx **['id', 'array', 'struct_specifier']**, so this combination is not handled by _add_array but in _add_struct

Subscript and rank

- one-dimensional (for example, **int a[1] -** rank one)
- multi-dimensions (for example, **int a[1][1] -** rank two)

From C Reference Manual §8.3:

```
A subscript can be a constant-expression "whose value is determinable at compile time,
and whose type is int." If there is no subscript, "the constant 1 is used".
```

Thus, **int a[b + c];** is valid. a and c should be constants (#define) and not variables.

Initializer (optional)

For example, **int a[1] {0}**, initialize a one-dimensional array with 0

From C Reference Manual §10,2:

```
An initializer for an array may contain a comma-separated list of compile-time expressions.
The length of the array is taken to be the maximum number of expressions in the list
[initializer list] and the square-bracketed constant in the array's declarator.
```

Thus, **int a[1] { 1, 2 }**, size of array is 2, not 1.

The initializer list may include symbolic constants and expressions that can be calculated at compile time and need to be compatible with the array type.

A single string may be given as the initializer for an array of char's; in this case, the characters in the string are taken as the initializing values.

Thus, **char a[] "ABC";** is valid and identical to **char *a "abc";**

char *a[] {"abc", "def", 0};

This is an array of pointers to character strings. The size of the array is 6 bytes; 2 pointers to the character strings and one pointer to 0. Hence, we need to read the strings, store them into memory, then initialize the array with the addresses to these strings.

Structures

In this setting, structures can be seen as a collection of variables, and can be declared as an array. The semantics of structures is somewhat unclear, and the combinations of different types of declarations are many.

A struct declaration can have an optional tag name in front of the declaration of its members. It can also optionally declare a variable, which may be of array type. If no variable is declared with the struct, its members are still declared and will occupy memory space. A struct declaration might be lacking both tag and variable, which therefore be seen just as a number of variables kept together. See more below.

The CCDecl.ctx list-attribute will always end with **'struct_specifier'**, but may contain additional elements.

Below, we try to sort out some of the cases, and the end result in memory.

Struct-symbol cases:

1. **with/without struct_tag** (examples with struct tag):
    - ctx has 1 element ['struct_specifier']: struct a { int b; }
    - ctx has 2 elements ['id', 'struct_specifier']: struct a c; or struct a { int b; } c;
    - ctx has 3 elements ['id', 'array', 'struct_specifier']: struct a c[10];
2. **with initializer**
    - struct a { char *id; int i; } c[] { "abc", 1, "def", 2 }; Initialize c with 2 elements
    - struct a { char *id; int i; }; struct b { struct a c[] { "abc", 1, "def", 2 }; int d { 0 } } e; e is a composite structure with initialization of its members.
3. **as an array**, see 1.c above.
4. **self-referential**, struct a { struct a *ptr1, *ptr2; };
5. **member is of type array**, struct a { char b[10]; } c;

Note

- that if a struct-variable is a pointer, this is indicated by the attribute symbol.pointer (list type).
- the struct tag (struct a) in the attribute symbol.struct_tag

Struct Members

For structs the following logic applies. According to C Reference Manual, §8.5:

```
The same member name can appear in different structures only if the two members are of the same type a
their origin with respect to their structure is the same; thus separate structures can share a common
initial segment.
```

This is an example, what is possible (even if not explained by above paragraph):

```
struct {int a1; char a2}
struct b {int a1; int a3;}
struct b *c;
```

This would declare 3 members: a1, a2 and a3, with unique memory positions (2 bytes each). The following operation is possible from above declarations: **c->a2++;**

Thus, c can reference member a2, even if it is not part for struct b declaration(!)

In the original compiler, it looks like members of structures are treated as variables, but with a "." prepended to the name. Thus, struct {int a1; char a2}; would become ".a1" and ".a2" variables. In this implementation we are not using this naming, but it is an indication that memebrs are seen as variables.

How about arrays of structures? For example, **struct a {int a1; char a2;} b[10];**

This declares an array of 10 a1- and a2-members. In memory, it would like so (memory positions to the left): 0: first a1 (2 bytes) 2: first a2 (1 byte) 3: padding (1 byte) 4: second a1 (2 bytes) 6: second a2 (1 byte) 7: padding (1 byte) and so on... 40 bytes in total reserved in memory. **b[1].a1** would refer to the second a1 at memory position 4. Syntactically, this is the same as ***(b+1).a1**

How does that work with the following example? **struct a {int a1; char a2} b[10]; struct c {int a1; int a3;} d[10];**

Would d be of same size as b? Yes!

But would **d[1].a2** be possible? Would that refer to the 1-byte at offset 6 in memory? **d[1].a3** would refer to the 2-bytes at offset 6 in memory.

Let's assume this is the intended logic, it is not clearly stated in manuals.

As a consequence, if a member name is already declared in the symbol table, we need to check that type and position within the struct is the same.

Initialization

Example, **struct a { char *id; int i; } c[] { "abc", 1, "def", 2 };**

First loop all struct members to know

- type (and size) for each member
- how many members, then we know the total size for the struct, 'struct size'

Then check if there is a variable (symbol.name) and check if it is of array type (**len(ctx) == 3** and **ctx[1] == 'array'**).

Check if there is subscript(s), if so calculate the array size. If there is no subscript, but still of array type, it is either of size 1 or the size of the initializer.

Now we need to loop the initializer (if existing) and map each value (which might be a constant expression) vs the members (check type vs member type). We count the number of initializer and make sure that the total count match count of members (**initializer count % member count == 0**).

If all is well so far, we can calculate the array size through **'array size' = 'initializer count' / 'member count'.**

If there has been a subscript, the 'array size' should match the subscript. If no subscript, 'array size' is used.

Now we can calculate the 'total struct size' as **'struct size' * 'array size'**.

Then reiterate the initializers and store them in memory, sequentially after each other. Then c (the struct variable) should be stored in memory as a pointer to the first element. Special attention must be given if a member is a pointer as the example above (*id), then it should be initialized to the memory position of the character-string.

A memory layout for above example can look as follows:

```
0:  abc0 (first initializer, 4 bytes)
4:  def0 (second initializer, 4 bytes)
8:  0 (*id, first struct member, first array element, 2 bytes)
10: 1 (i, second struct member, first array element, 2 bytes)
12: 4 (*id, first struct member, second array element, 2 bytes)
14: 2 (i, second struct member, second array element, 2 bytes)
16: 8 (c[], struct/array variable referring to the first struct member if the first array element, 2 b
```