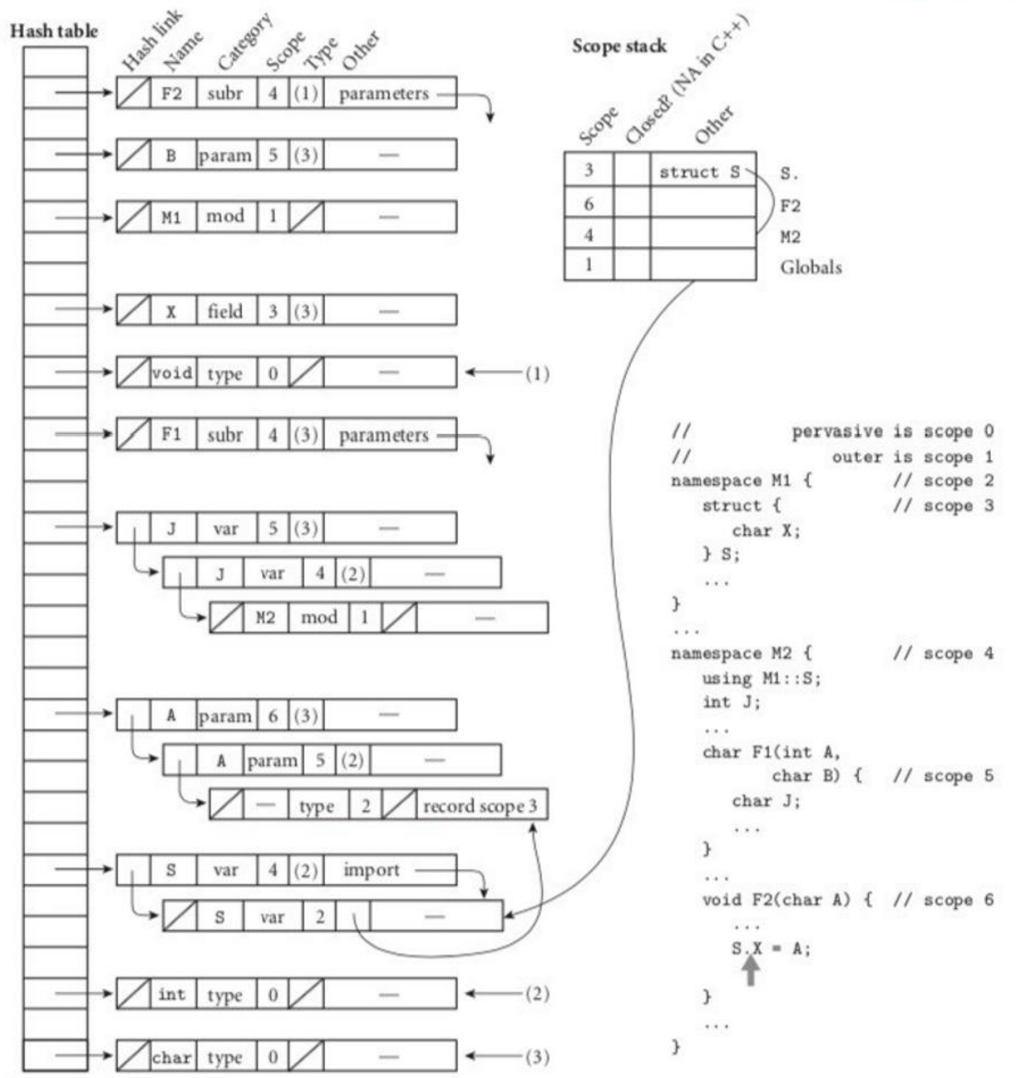
Hash Tables

An Advanced Introduction to Unix/C Programming

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands

Symbol Table Implementation - Example

- In a language with static scoping, the compiler uses an **insert operation** to place a name-to-entity binding into the symbol table for each newly encountered declaration
- When it encounters the use of a name that should already have been declared, the compiler uses a **lookup operation** to search for an existing binding
- Most compilers never delete anything from the symbol table. Instead, they manage visibility using **enter scope** and **leave scope** operations.
- Usually implemented as hash tables
- Return closest lexical declaration to handle nested lexical scoping



```
//              pervasive is scope 0
//                  outer is scope 1
namespace M1 {          // scope 2
   struct {             // scope 3
      char X;
   } S;
   ...
}
...
namespace M2 {          // scope 4
   using M1::S;
   int J;
   ...
   char F1(int A,
           char B) {    // scope 5
      char J;
      ...
   }
   ...
   void F2(char A) {    // scope 6
      ...
      S.X = A;
   }
   ...
}
```

Symbol Tables

You are a compiler writer and you are assigned to create the symbol table to store the identifiers, subroutines, and parameters found in a program.

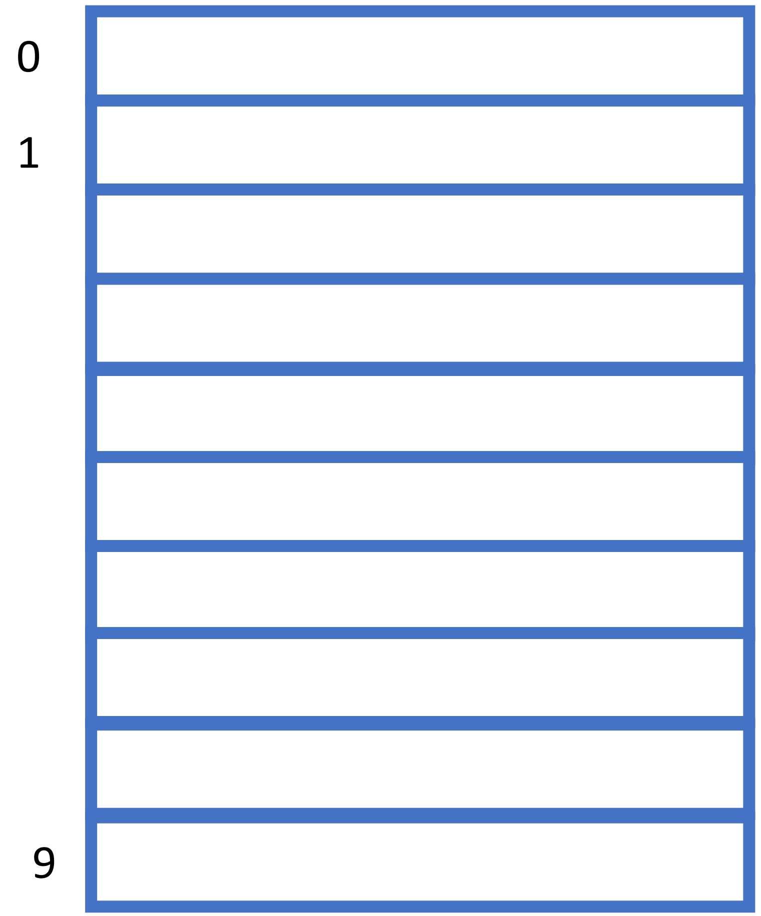
How would you store the symbol with their attributes?

What data structure could you use?

Hash Table

A hash table is fixed size array.

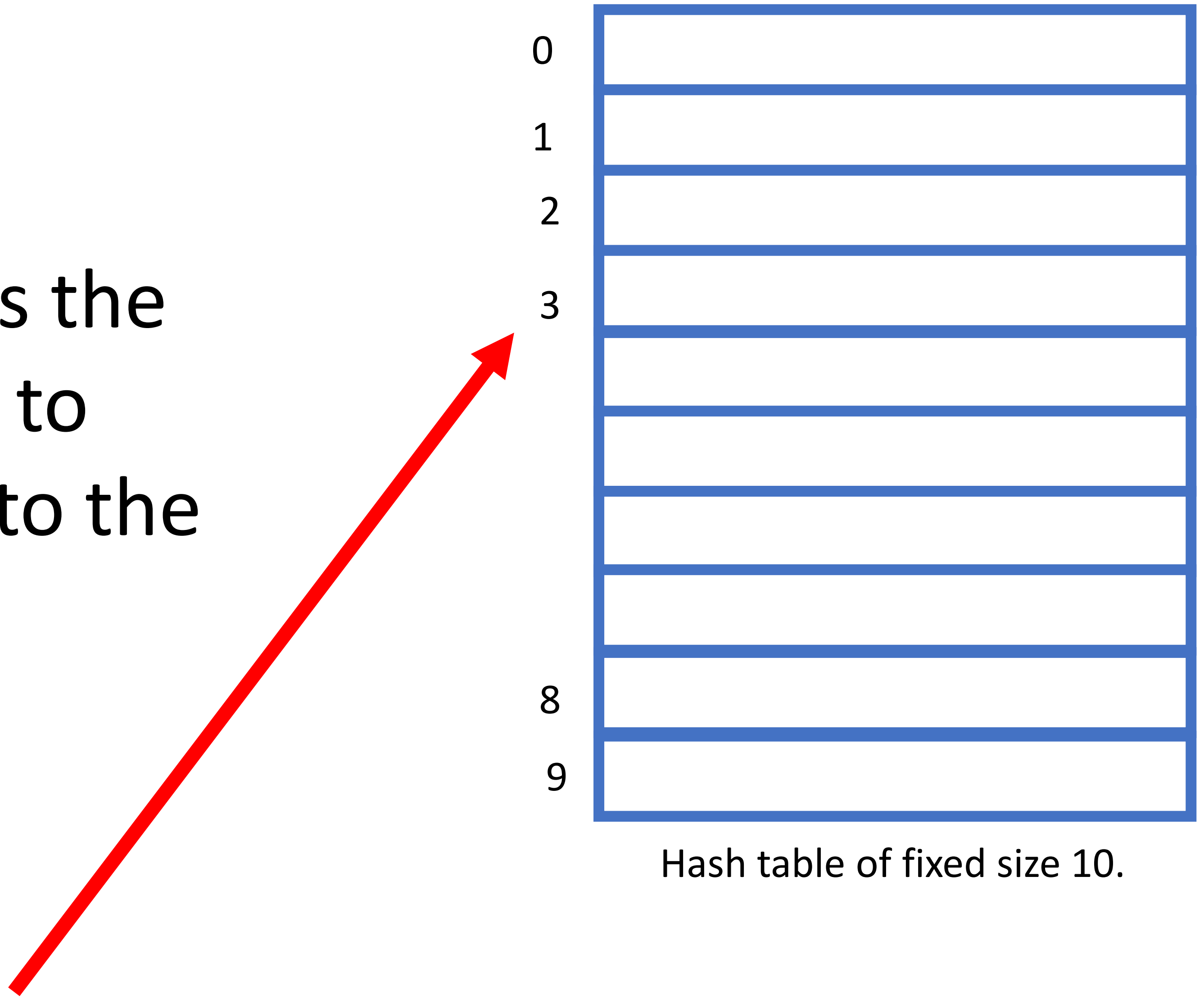
You use a hash function to compute the index into the hash table array.



Hash table of fixed size 10.

Hash Function

A hash function takes the identifier and uses it to compute an index into the hash table.



Hash table of fixed size 10.

index_into_hash_table = hash_function("identifier_token");

Hash Table

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

#define MAX_HASH_TABLE_SIZE     100

struct node {
    char    function_name[62];
    int     calls_count;
    char    calls[30][62];
} hash_table[MAX_HASH_TABLE_SIZE];
```

Hash Function

```
int hash_function(char *name)
{
   int     hash_index;
   int     i;

   hash_index = 0;
   for (i=0; i<strlen(name); i++)
      hash_index = (hash_index*23) + name[i];
   hash_index = hash_index % MAX_HASH_TABLE_SIZE;

   return(hash_index);
}
```

Insert Data Into Hash Table

```
int insert(int count, ...)
{
   int     hash_index;
   int     i;
   char    function_name[62];
   va_list vlist;

   va_start(vlist, count);

   strcpy(function_name, va_arg(vlist, char*));

   hash_index = hash_function(function_name);

   strcpy(hash_table[hash_index].function_name,
            function_name);

   hash_table[hash_index].calls_count = count-1;

   for (i=0; i<count-1; i++)
       strcpy(hash_table[hash_index].calls[i], va_arg(vlist, char*));

   printf("Insert(): Function %s (hash index = %d):\n",
            function_name, hash_index);

   for (i=0; i<count-1; i++)
      printf("\t\tCalls: %s\n", hash_table[hash_index].calls[i]);
   printf("\n");

   va_end(vlist);
}
```

Select From Hash Table / Main Program

```
int select(char *function_name)
{
   int hash_index = hash_function(function_name);

   printf("Select(): Function %s (hash_index = %d):\n",
            function_name, hash_index);

   for (int i=0; i<hash_table[hash_index].calls_count; i++)
      printf("\t\tCalls: %s\n", hash_table[hash_index]. calls[i]);
   printf("\n");
}
```

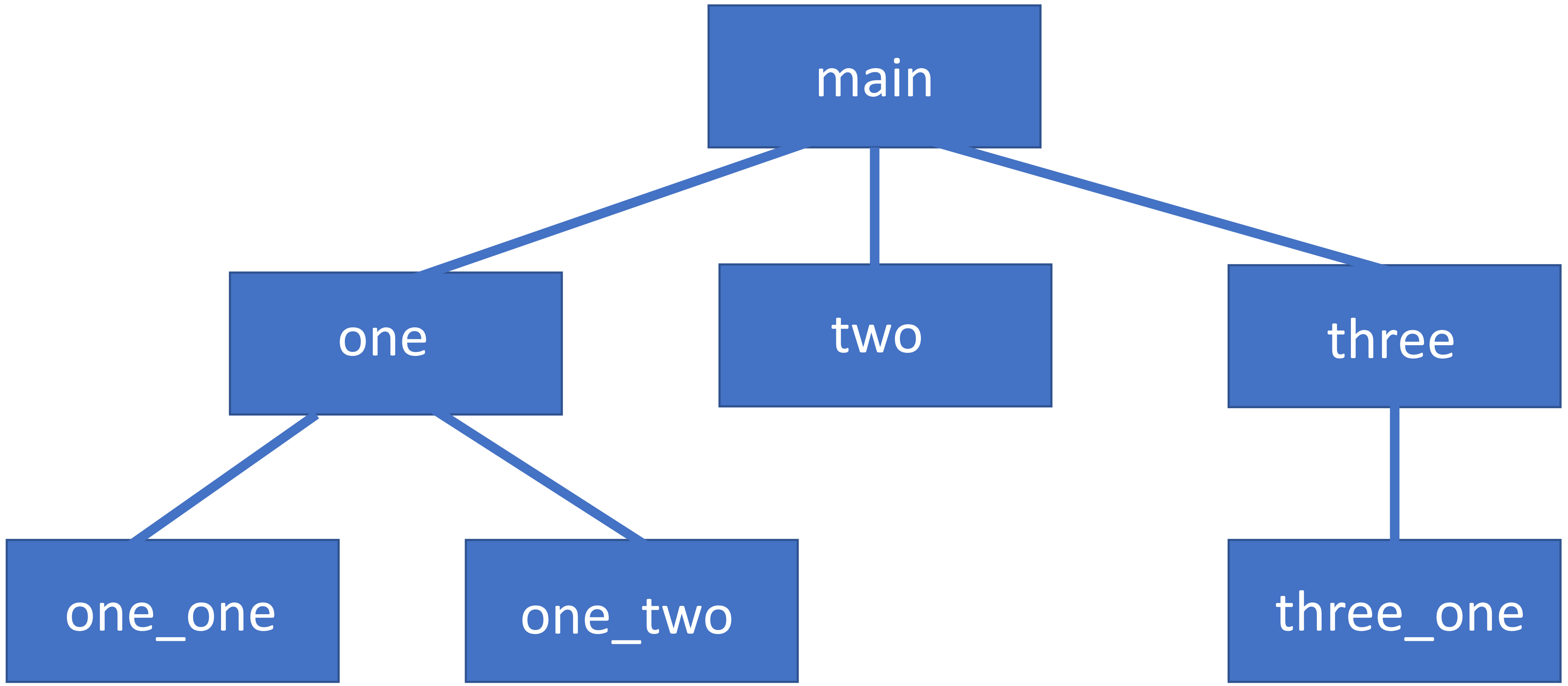
```
int main()
{
   insert(4, "main", "one", "two", "three");
   insert(3, "one", "one_one", "one_two");
   insert(1, "two");
   insert(2, "three", "three_one");

   select("main");
   select("three");
}
```

Select From Hash Table / Main Program

```
john@oho:~$ gcc hash.c; a.out
Insert(): Function main (hash index = 41):
        Calls: one
        Calls: two
        Calls: three

Insert(): Function one (hash index = 50):
        Calls: one_one
        Calls: one_two

Insert(): Function two (hash index = 12):

Insert(): Function three (hash index = 54):
        Calls: three_one
```

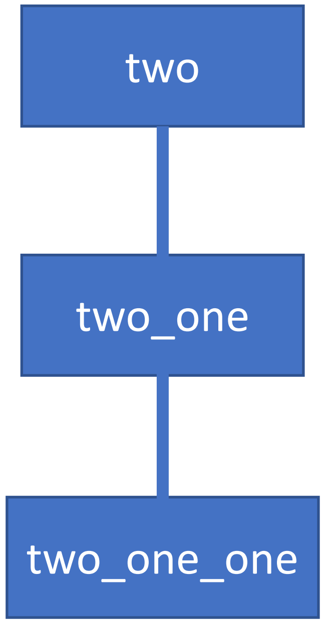
```
Select(): Function main (hash_index = 41):
        Calls: one
        Calls: two
        Calls: three

Select(): Function three (hash_index = 54):
        Calls: three_one
```

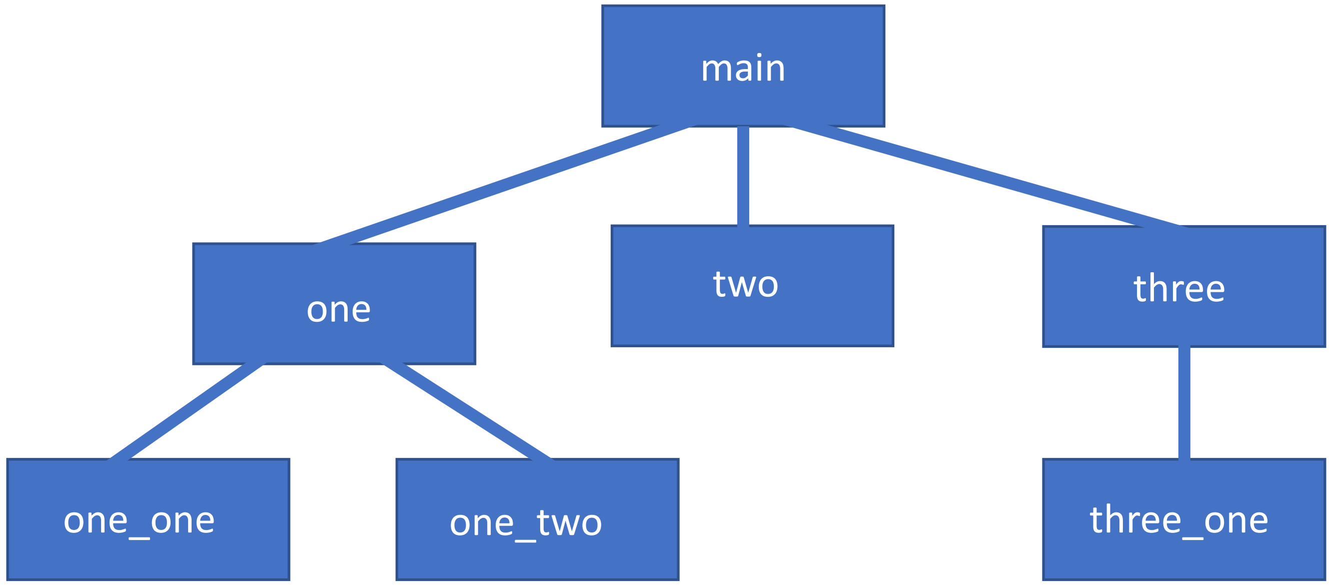
Hash Table Represents Calling Structure



Hash Table Represents Calling Structure

Viewpath: /home/john/ABC_Working_Dir /home/baselines/Project_ABC





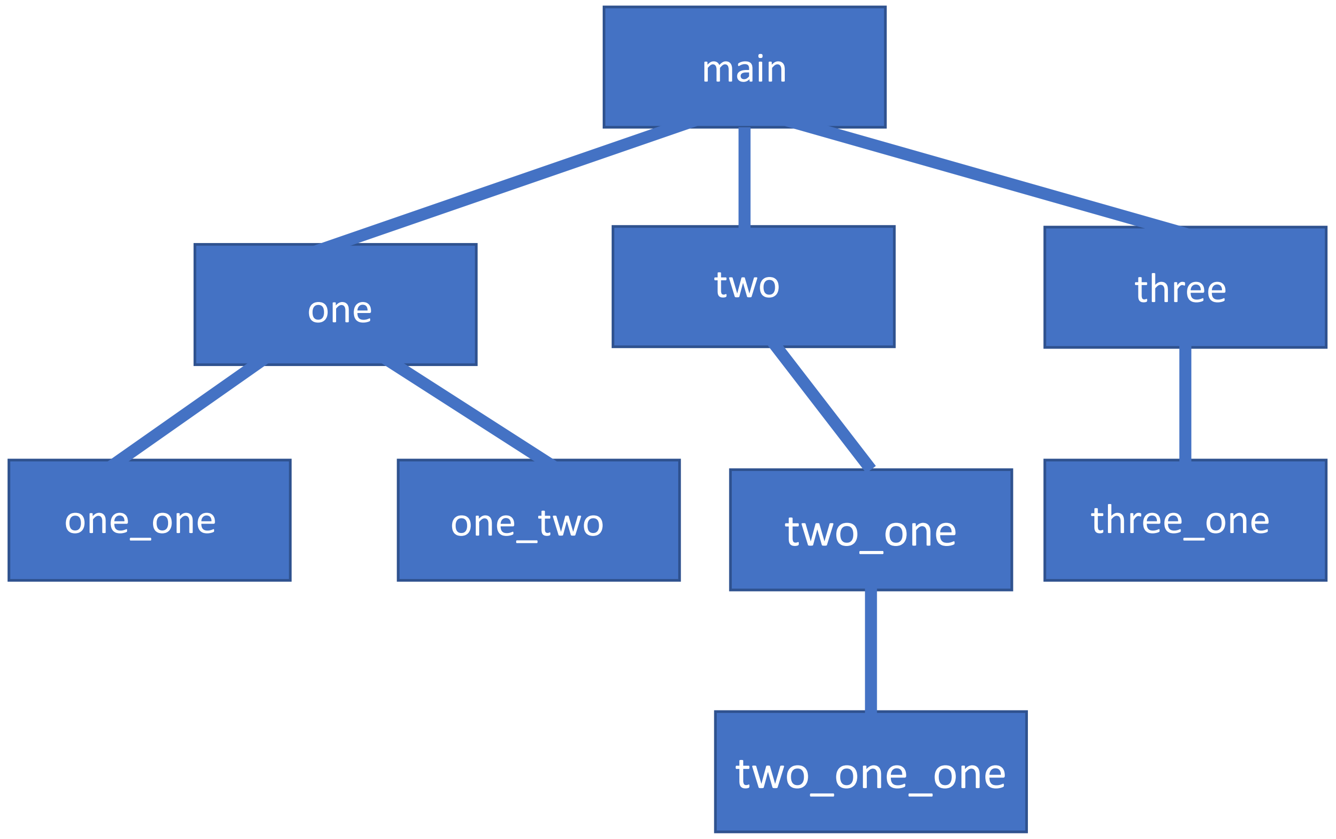
ABC_Working_Directory Hash Table: 100 Entries

Project_ABC Hash Table: 500 Entries

Hash Table Represents Calling Structure

Viewpath: /home/john/ABC_Working_Dir /home/baselines/Project_ABC

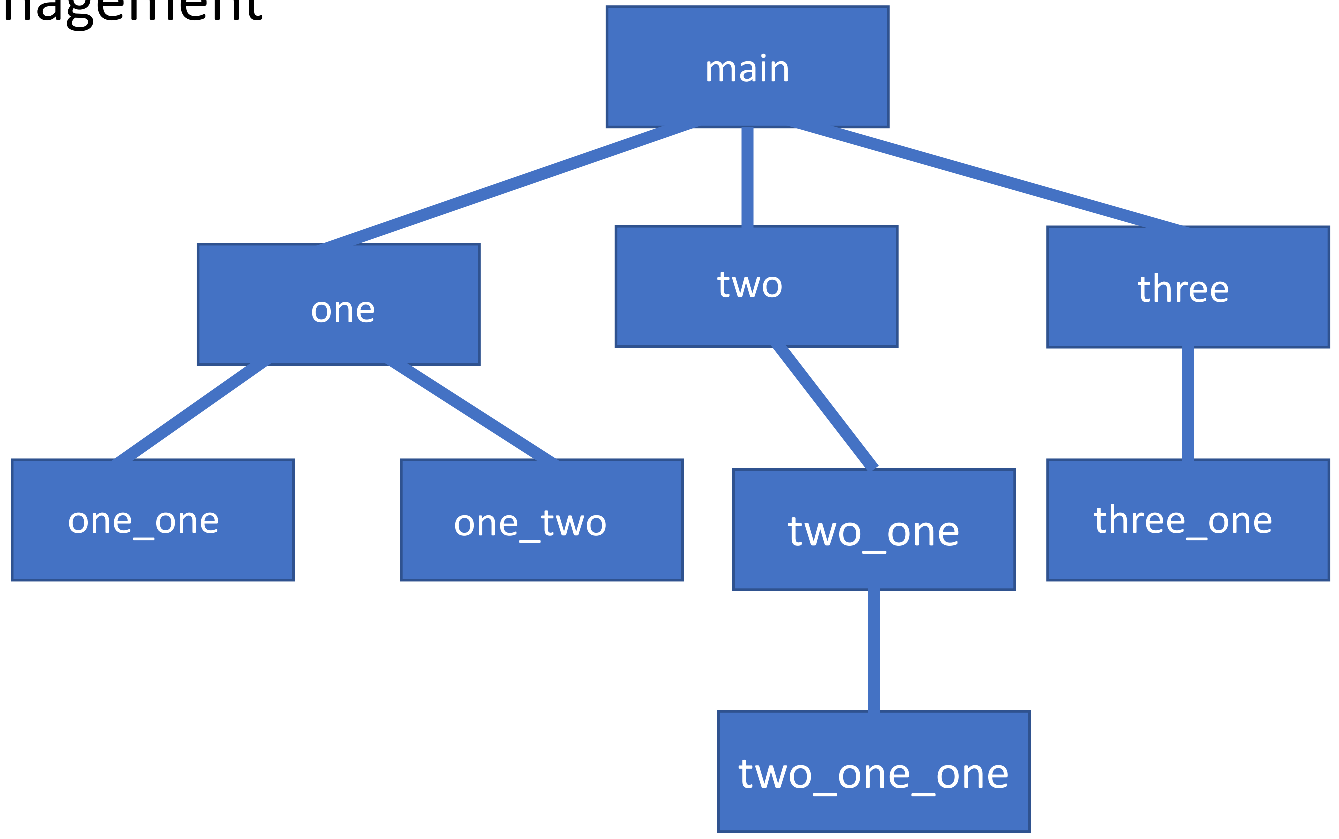
View from /home/john/ABC_Working_Dir



Why Do This?

Supports:

Automated Configuration Management
Software Reusability
Unit and Regression Testing
Requirement Tracking
User Workspaces
and much more!


main
one
two
three
one_one
one_two
two_one
three_one
two_one_one