Symbol Table

Symbol Table

- The data structure that is created and maintained by the compilers for information storing regarding the occurrence of various entities like names of variables, functions, objects, classes
- Symbol table is used by both the analysis and the synthesis parts of a compiler

Symbol Table

- A symbol table may serve the following purposes depending upon the language in hand:
  - To store the names of all entities in a structured form at one place
  - To verify if a variable has been declared
  - To implement type checking, by verifying assignments and expressions in the source code are semantically correct
  - To determine the scope of a name (scope resolution)

Information Stored in Symbol Table

- The following possible information about identifiers are stored in symbol table
  - The name (as a string)
  - Attribute: Reserved word, Variable name, Type name, Procedure name, Constant name
  - The data type
  - The block level
  - Its scope (global, local, or parameter)
  - Its offset from the base pointer (for local variables and parameters only)

Implementation

- Symbol table can be implemented as
  - Unordered List
  - Linear (sorted or unsorted) list
  - Binary Search Tree
  - Hash table
- Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Entry Format

- A symbol table maintains an entry for each name in the following format:

  <symbol name, type, attribute>

- For example, if a symbol table has to store information about the following variable declaration:

  static int interest;

- then it should store the entry such as:

  <interest, int, static>

Operations

- A symbol table, either linear or hash, should provide the following operations.
  - insert()
    - This operation is more frequently used by analysis phase where tokens are identified and names are stored in the table.
    - This operation is used to add information in the symbol table about unique names occurring in the source code.
    - The format or structure in which the names are stored depends upon the compiler in hand.

Operations

- An attribute for a symbol in the source code is the information associated with that symbol.
  - This information contains the value, state, scope, and type about the symbol.
- The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.
- For example:

  int a;

  should be processed by the compiler as:

  insert(a, int);

Operations

- lookup()
  - lookup() operation is used to search a name in the symbol table to determine:
    - if the symbol exists in the table.
    - if it is declared before it is being used.
    - if the name is used in the scope.
    - if the symbol is initialized.
    - if the symbol declared multiple times.
- The basic format should match the following:

  lookup(symbol)

Operations

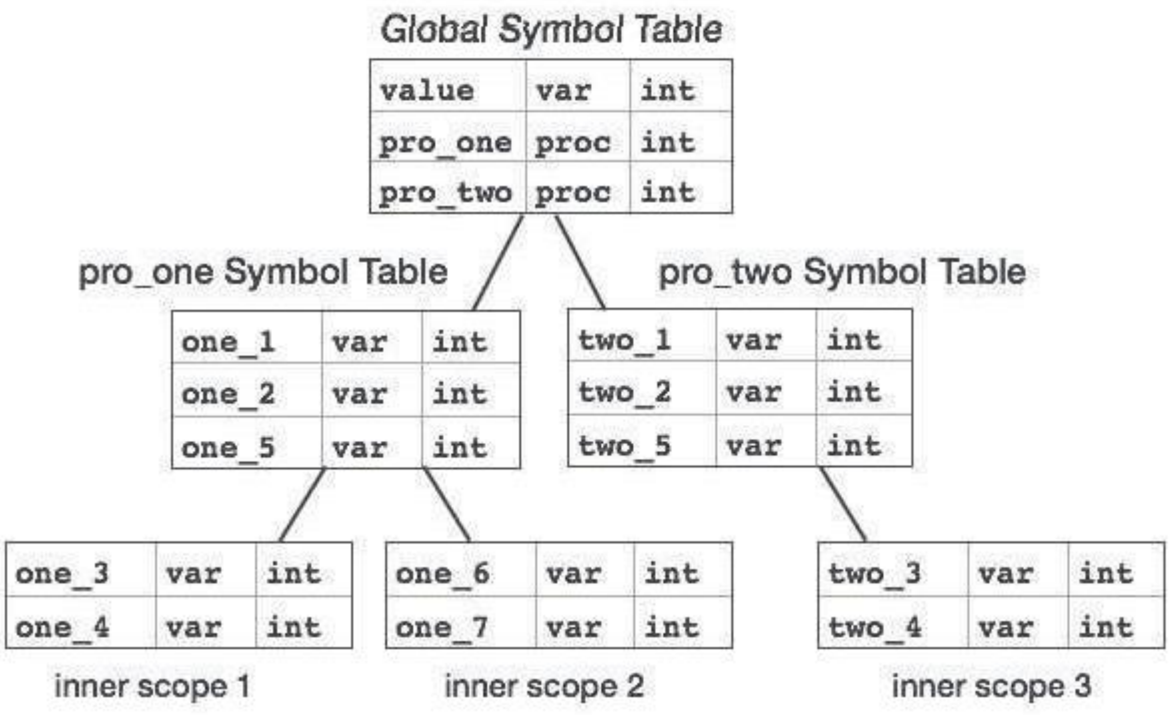
- This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

- A compiler maintains multiple block levels of symbol tables:
  - **Level 0:** A null hash table at level 0
  - **Level 1:** Keyword in the hash table at level 1
  - **Level 2: Global symbol table** which can be accessed by all the procedures
  - **Level 4: Scope symbol tables** that are created for each scope in the program

Scope Management

- Symbol tables are arranged in hierarchical structure as shown in the example below:



Global Symbol Table

value	var	int
pro_one	proc	int
pro_two	proc	int

pro_one Symbol Table

one_1	var	int
one_2	var	int
one_5	var	int

pro_two Symbol Table

two_1	var	int
two_2	var	int
two_5	var	int

inner scope 1

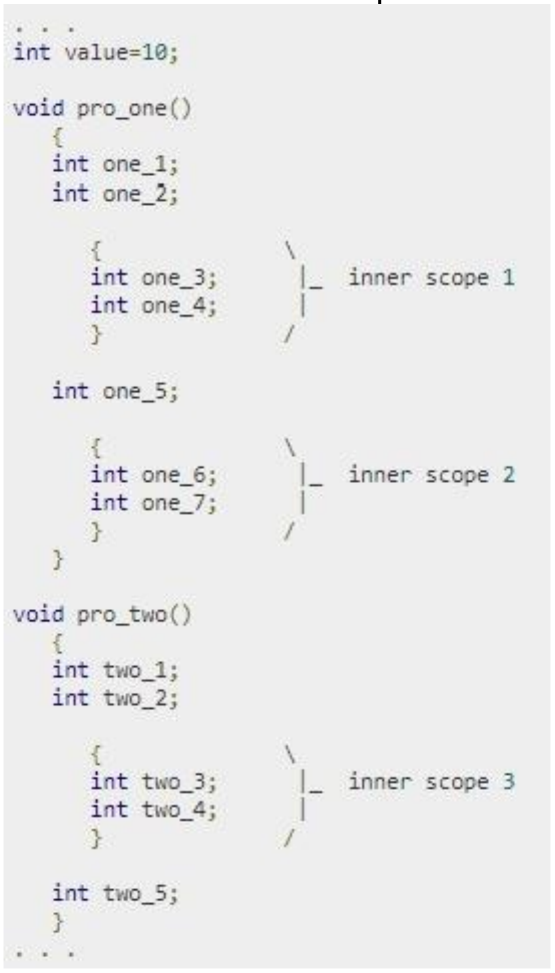
one_3	var	int
one_4	var	int

inner scope 2

one_6	var	int
one_7	var	int

inner scope 3

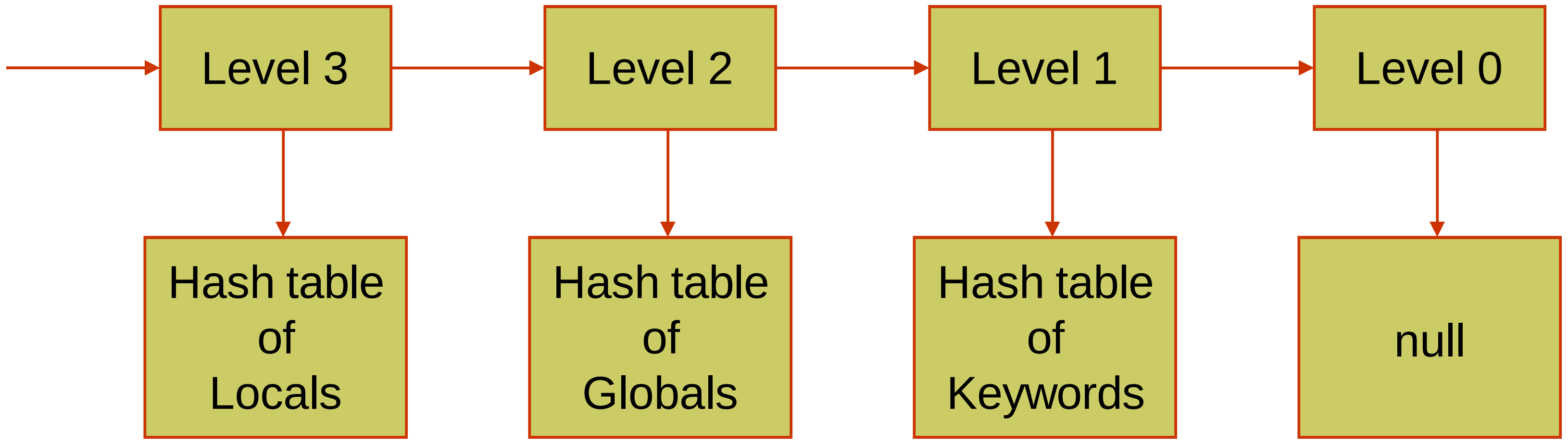
two_3	var	int
two_4	var	int



```
. . .
int value=10;

void pro_one()
   {
   int one_1;
   int one_2;

      {              \
      int one_3;      |_  inner scope 1
      int one_4;      |
      }              /

   int one_5;

      {              \
      int one_6;      |_  inner scope 2
      int one_7;      |
      }              /
   }

void pro_two()
   {
   int two_1;
   int two_2;

      {              \
      int two_3;      |_  inner scope 3
      int two_4;      |
      }              /

   int two_5;
   }
. . .
```

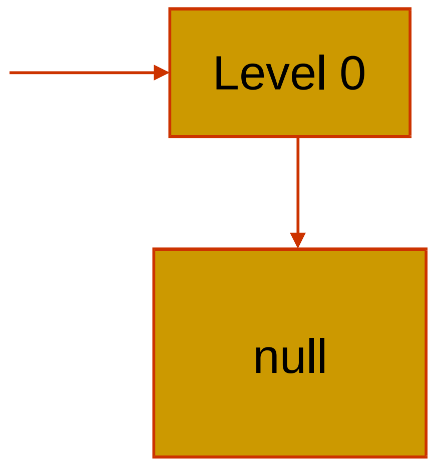
Structure of the Symbol Table

- We will implement the symbol table as a linked list of hash tables, one hash table for each block level.



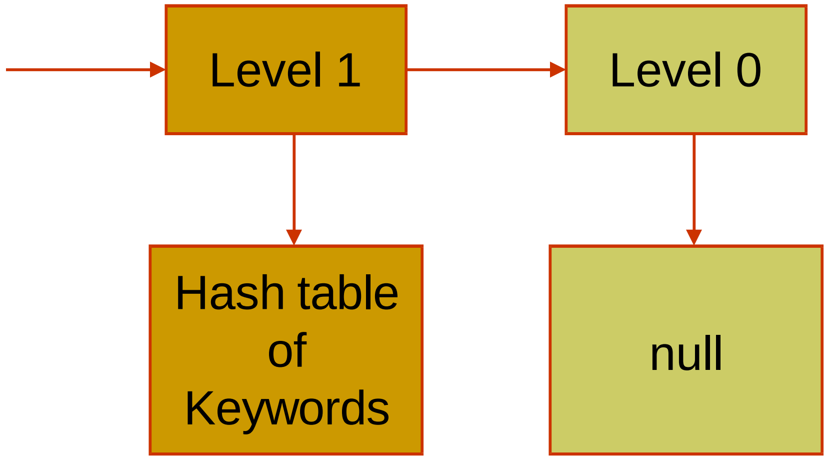
Structure of the Symbol Table

- Initially, we create a null hash table at level 0.



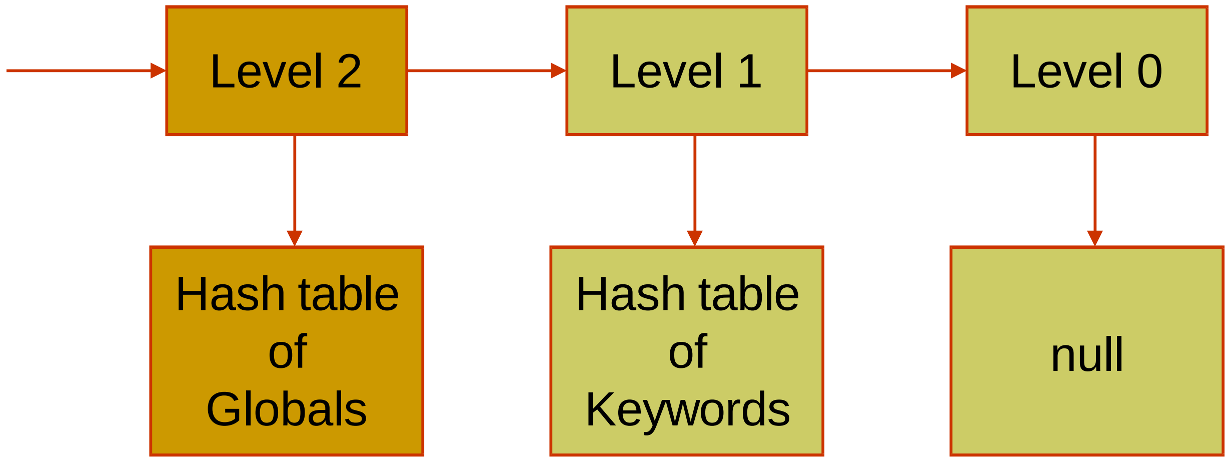
Structure of the Symbol Table

- Then we increase the block level and install the keywords in the symbol table at level 1.



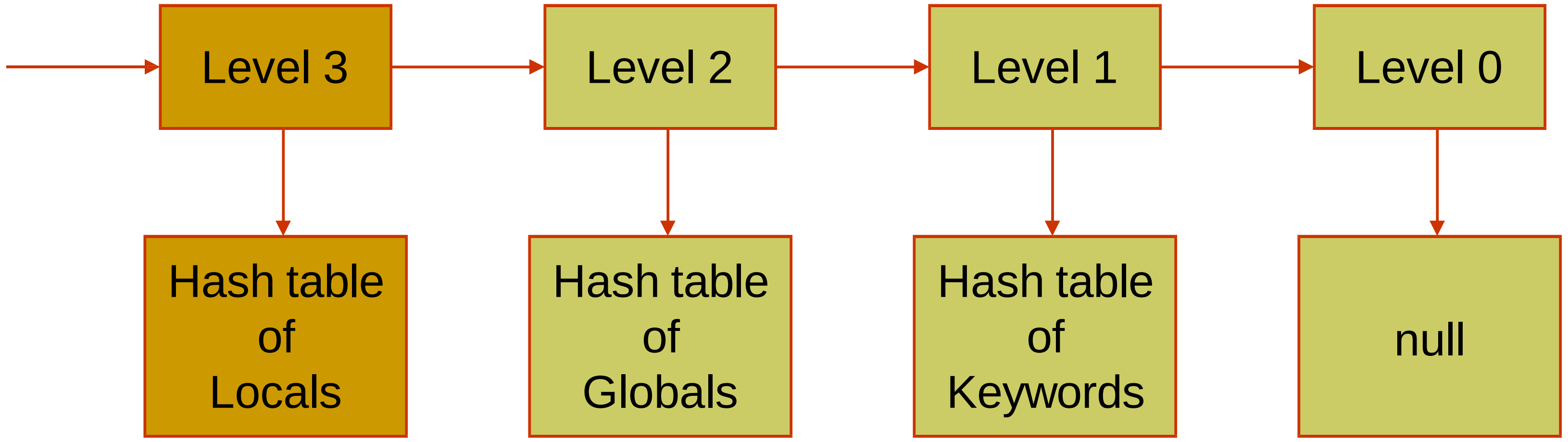
Structure of the Symbol Table

- Then we increase the block level and install the globals at level 2.



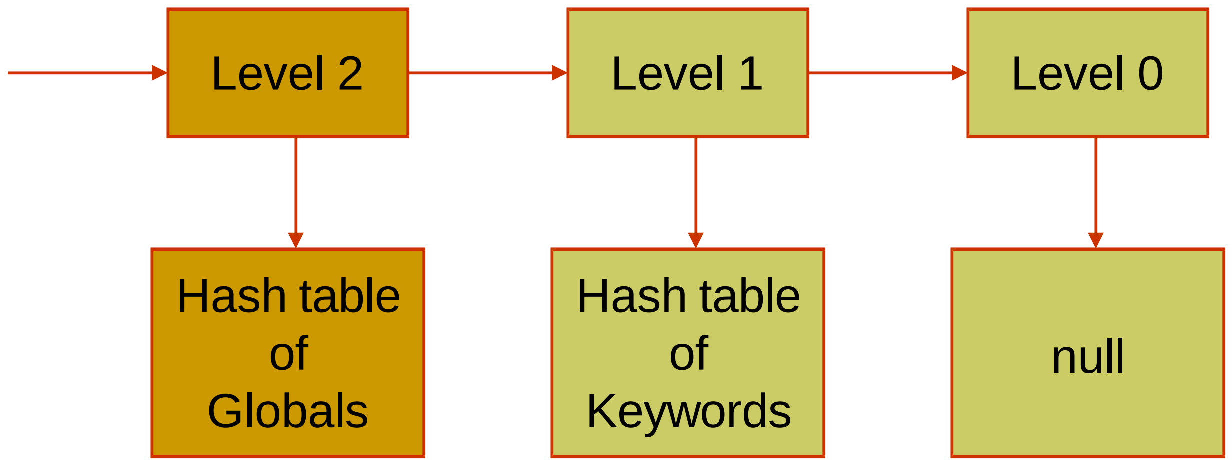
Structure of the Symbol Table

- When we enter a function, we create a level 3 hash table and store parameters and local variables there.



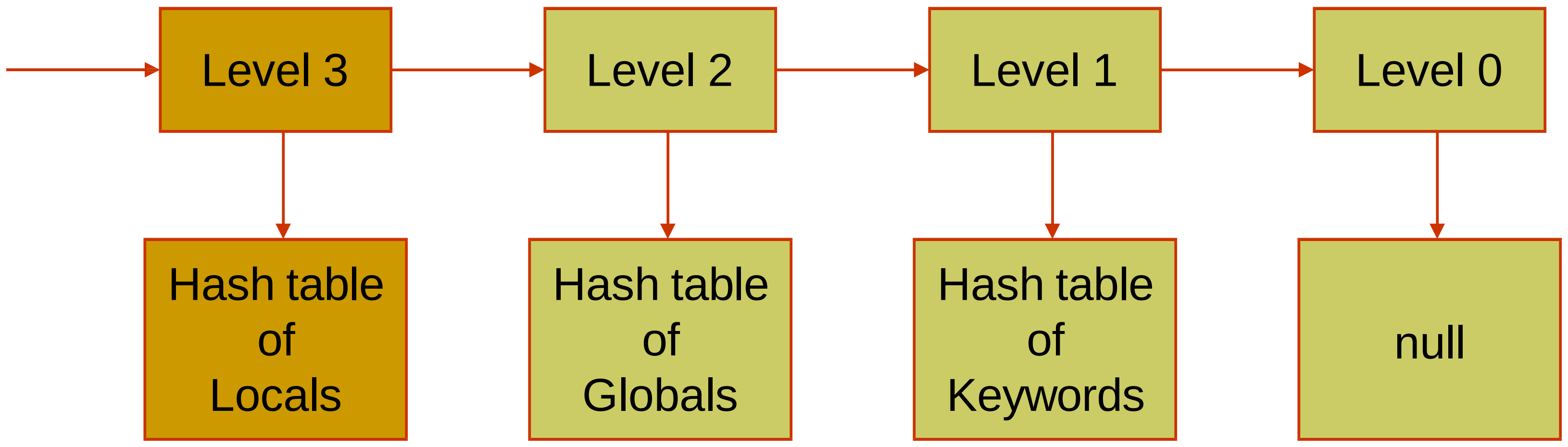
Structure of the Symbol Table

- When we leave the function, the hash table of local variables is deleted from the list.



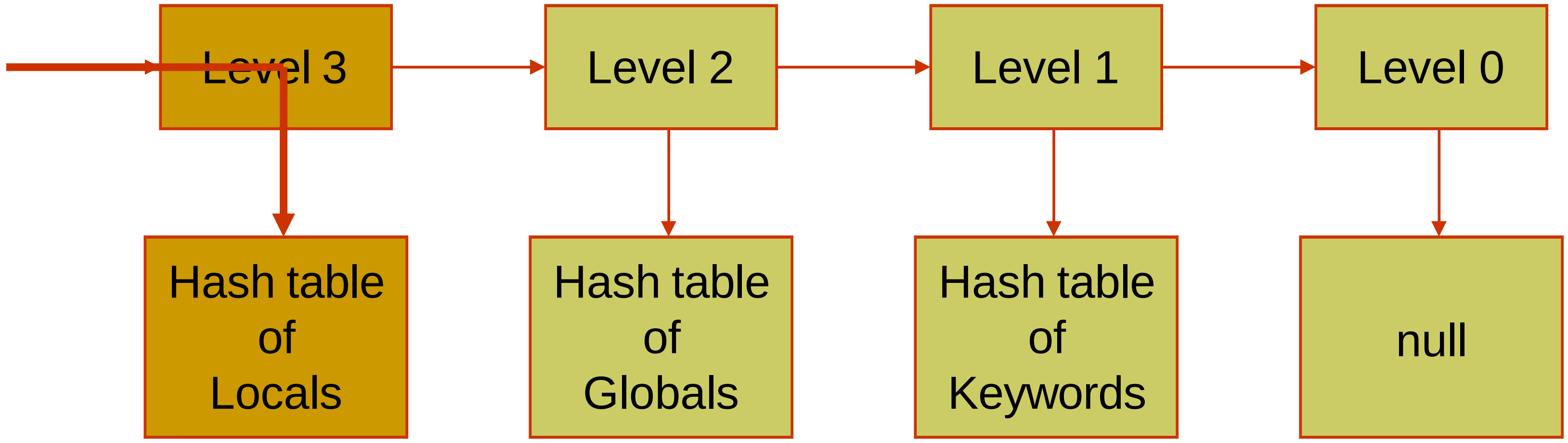
Locating a Symbol

- If we enter another function, a new level 3 hash table is created.



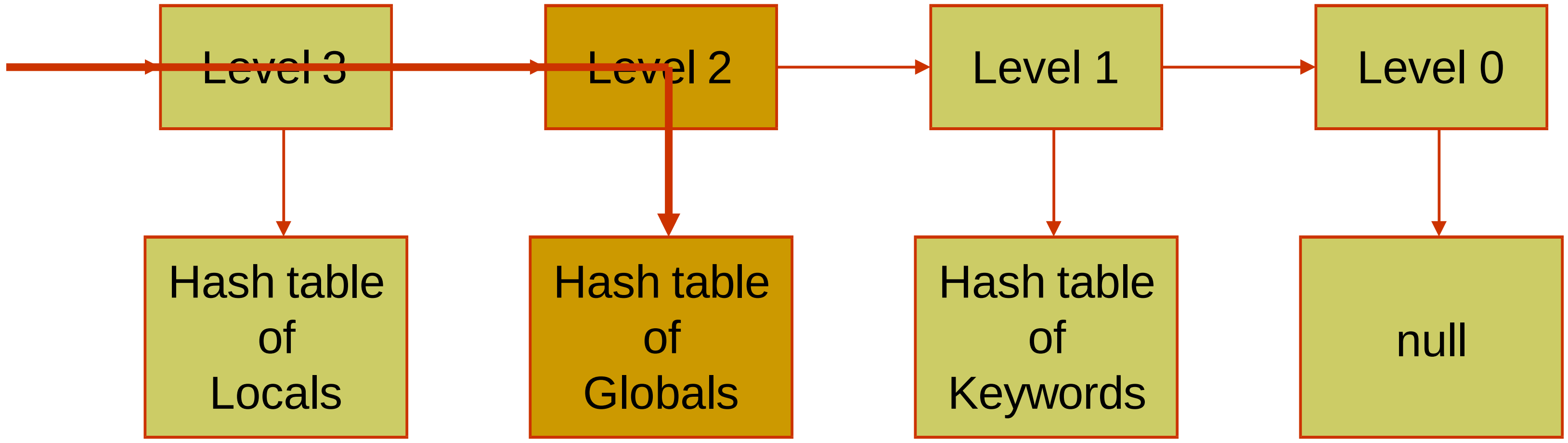
Locating a Symbol

- When we look up an identifier, we begin the search at the head of the list.



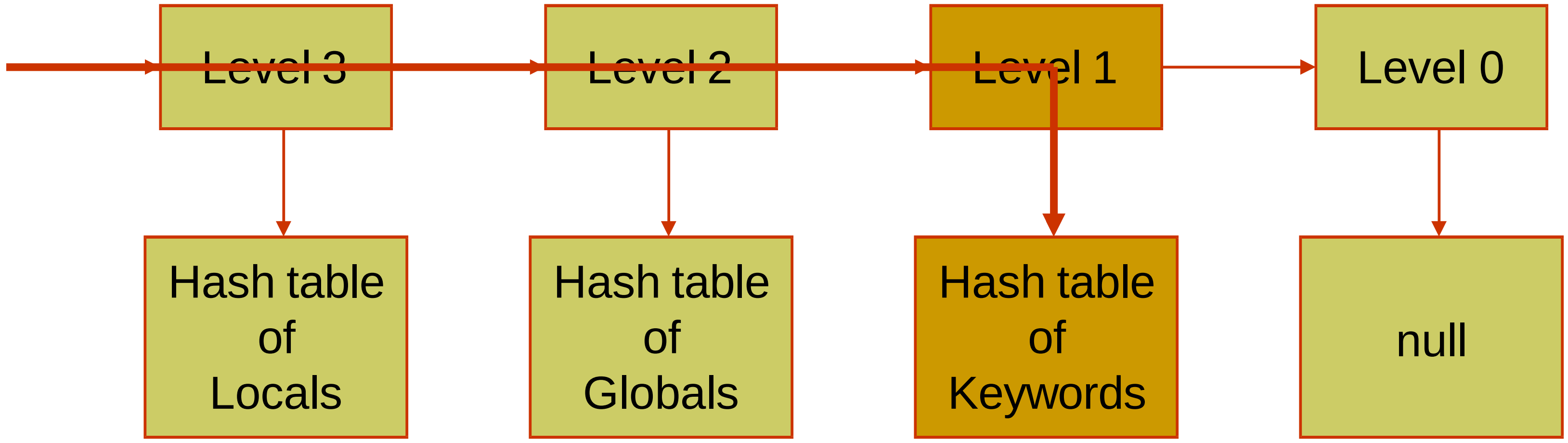
Locating a Symbol

- If it is not found there, then the search continues at the lower levels.



Locating a Symbol

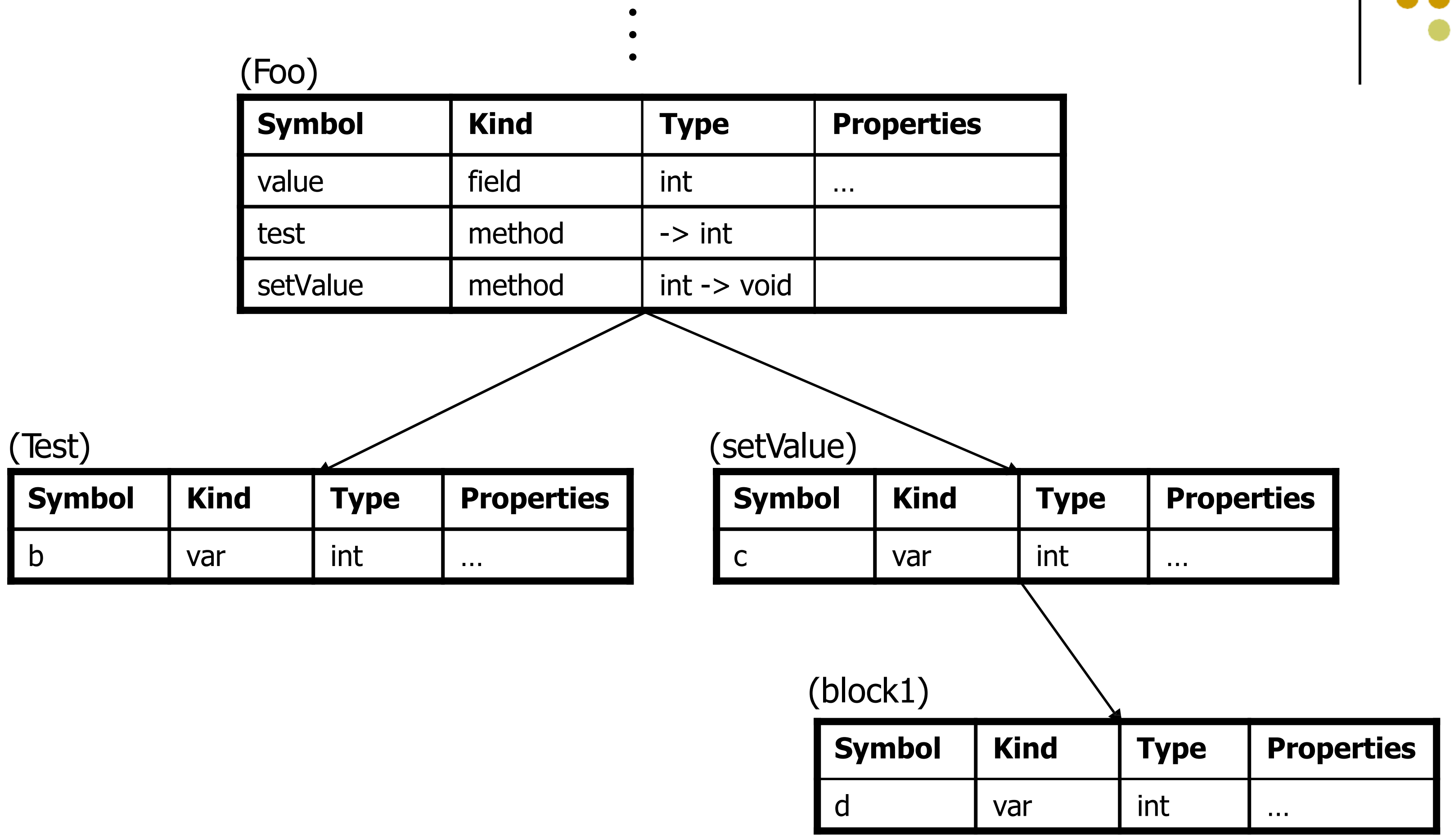
- Keywords are found in the level 1 hash table.



Symbol table example

```
        class Foo {
          int value;
          int test() {
          int b = 3;                              scope of b
            return value + b;
          }
          void setValue(int c) {
            value = c;                                                              scope of value
            { int d = c;
              c = c + d;                                         scope of c
block1        value = c;                scope of d
            }
          }
        }

        class Bar {
          int value;
          void setValue(int c) {
                  value = c;
          }                                                      scope of c  scope of value
        }
```

Symbol table example cont.



⋮

(Foo)

Symbol	Kind	Type	Properties
value	field	int	…
test	method	-> int	
setValue	method	int -> void	

(Test)

Symbol	Kind	Type	Properties
b	var	int	…

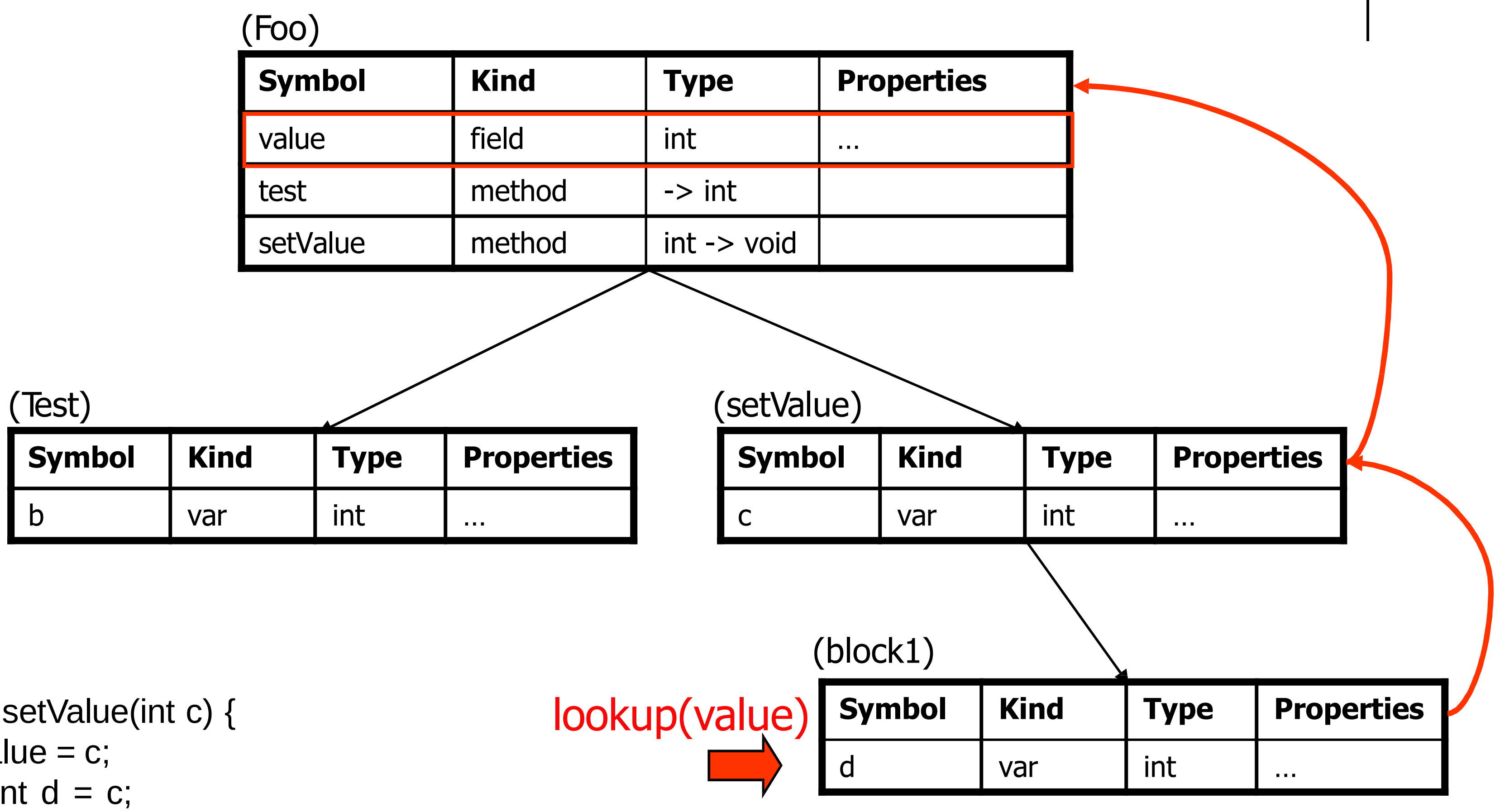
(setValue)

Symbol	Kind	Type	Properties
c	var	int	…

(block1)

Symbol	Kind	Type	Properties
d	var	int	…

Checking scope rules



(Foo)

Symbol	Kind	Type	Properties
value	field	int	…
test	method	-> int	
setValue	method	int -> void	

(Test)

Symbol	Kind	Type	Properties
b	var	int	…

(setValue)

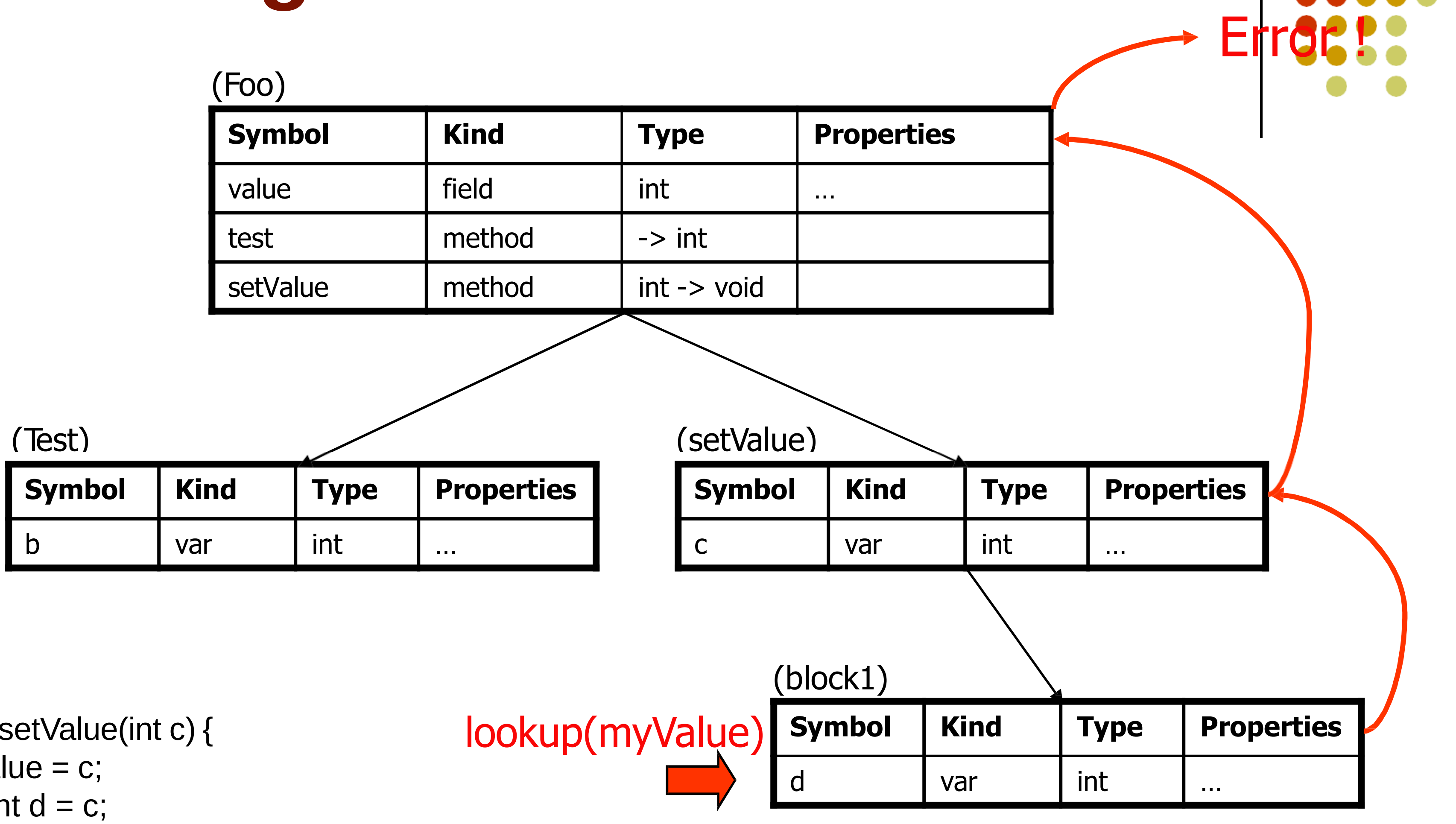
Symbol	Kind	Type	Properties
c	var	int	…

(block1)

Symbol	Kind	Type	Properties
d	var	int	…

```
void setValue(int c) {
   value = c;
   { int d = c;
     c = c + d;
     value = c;
   }
}
```

Catching semantic errors



(Foo)

Symbol	Kind	Type	Properties
value	field	int	…
test	method	-> int	
setValue	method	int -> void	

(Test)

Symbol	Kind	Type	Properties
b	var	int	…

(setValue)

Symbol	Kind	Type	Properties
c	var	int	…

(block1)

Symbol	Kind	Type	Properties
d	var	int	…

```
void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      myValue = c;
    }
}
```

Hash Tables

- A *hash table* is a list in which each member is accessed through a *key*.
- The key is used to determine where to store the value in the table.
- The function that produces a location from the key is called the *hash* function.
- For example, if it were a hash table of strings, the hash function might compute the sum of the ASCII values of the first 5 characters of the string, modulo the size of the table.

Hash Tables

- The numerical value of the hashed key gives the location of the member.
- Thus, there is no need to search for the member; the hashed key tells where it is located.
- For example, if the string were `"return"`, then the key would be (114 + 101 + 116 + 117 + 114) % 100 = 62.
- Thus, `"return"` would be located in position 62 of the hash table.

Clashes and Buckets

- Clearly, there is the possibility of a clash: two members have the same hashed key.
- In that case, the hash table creates a list, called a “bucket,” of those values in the table with that same location.
- When that location comes up, the list is searched.
- However, it is generally a very short list, especially if the table size has been chosen well.

Hash Table Efficiency

- The two parameters that determine how efficiently the hash table performs are
  - The capacity of the table, i.e., the total amount of memory allocated.
  - The number of buckets, or equivalently, the size of a bucket.
- Clearly, the size of a bucket times the number of buckets equals the capacity of the table.

Hash Table Efficiency

- For a given hash table capacity,
  - If there are too many buckets, then many buckets will not be used, leading to space inefficiency.
  - If there are too few buckets, then there will be many clashes, causing the searches to degenerate into predominately sequential searches, leading to time inefficiency.

- End of Chapter # 13