CS-4031 Compiler Construction

Fall 2025

Symbol Table

Course Instructor: Syed Zain Ul Hassan

National University of Computer and Emerging Sciences, Karachi

Email: zain.hassan@nu.edu.pk

Symbol Table

- ❑ The symbol table is used to store essential information about every symbol contained within the program

- ❑ This includes
  - ❑Keywords
  - ❑Data types
  - ❑Operators
  - ❑Functions
  - ❑Variables
  - ❑Constants
  - ❑Literals

Symbol Table

- The symbol table is a repository of all information stored within a compiler
- The symbol table maps names into attributes
- It stores the following:
  - **For type names**: Their type definitions
  - **For variables**: Their types
  - **For arrays**: Their types and dimension
  - **For constants**: Their type and value
  - **For functions**: Their formal parameter list and output type

Contents of Symbol Table

- ❑ The symbol table contains the following information:
  - ❑ Lexeme
  - ❑ Token class
  - ❑ Semantic component (e.g., variable, operator, function, constant, etc.)
  - ❑ Data type
  - ❑ Scope information
  - ❑ Pointer to other entries (if necessary)

Key Operations on Symbol Table

- ❑ Insert
  - ❑Add new identifiers when first discovered during compilation, with initial attribute assignment

- ❑ Lookup (search)
  - ❑Efficiently find existing entries to verify declarations and retrieve semantic information

- ❑ Delete
  - ❑Remove identifiers when their scope ends, managing memory and preventing conflicts

Key Operations on Symbol Table

❑ Insert

❑The reserved words, standard identifiers and operators are placed in the Symbol Table during its initialization

❑New lexemes are added when the scanner encountersthem, and they are assigned a token class

❑Similarly, the semantic analyzer adds the appropriateproperties and attributes that belong to the lexeme

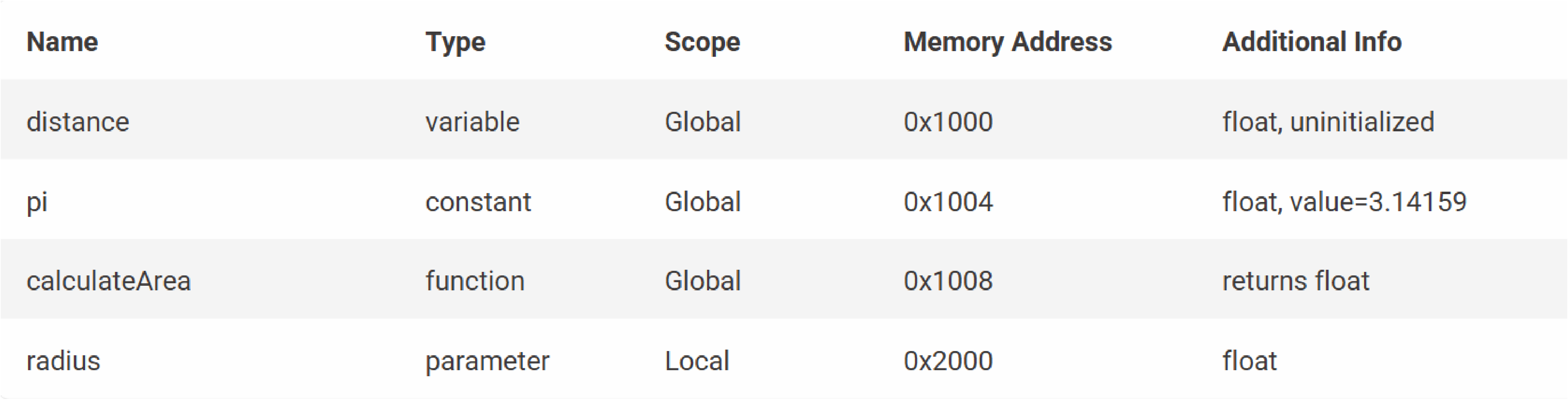
Key Operations on Symbol Table

- ❑ Delete
    - ❑ When the compiler is finished with a given scope of the program, all the symbols belonging to that scope must be effectively removed before beginning to process a different scope of the program
    - ❑ The data regarding these variables may be hidden from the compiler's view or dumped into a temporary file

Implementation Approaches

- ❑ Hash Table
  - ❑Fast O(1) average lookup time, widely used in modern compilers like GCC and LLVM
- ❑ Linear List
  - ❑Easy to implement but inefficient *O(n)* lookup for large programs
- ❑ Binary Search Tree
  - ❑Maintain sorted order with *O(log n)* operations, moderate efficiency

Example: Symbol Table Entries



Name	Type	Scope	Memory Address	Additional Info
distance	variable	Global	0x1000	float, uninitialized
pi	constant	Global	0x1004	float, value=3.14159
calculateArea	function	Global	0x1008	returns float
radius	parameter	Local	0x2000	float

Constructing Symbol Table

- ❑ Consider the following code snippet:

```
int x;
float y;
void foo(int a, float b) {
int x;
x = a + 1;
{
float a;
a = b;
}
}
```

Step 0: Initialize

- ❑ Create scope level 0 (global scope)

Step 1: read x

❑ scope level = 0

❑Create symbol as:

name = "x", kind = var, type = int, scope_level = 0, decl_line = 1

size = 4, offset = assign_global_address()

❑Insert into table

Step 2: read y

❑ scope level = 0

❑Create symbol as:

name = “y", kind = var, type = float, scope_level = 0, decl_line = 2

size = 8, offset = next_global_address()

❑Insert into table

Step 3: read void foo(int a, float b)

❑ scope level = 0

❑Insert function as:

name = “foo", kind = function, return_type = void, n_params=2, params_type=[int,float], scope_level = 0, decl_line = 4

❑Call *push_scope()* to enter scope level 1 (foo’s scope)

Step 4: process parameters a, b

❑ For a:

❑scope level = 1

❑*name = “a", kind = parameter, type = int, scope_level = 1, decl_line = 4, offset = param_offset(a)*

❑Insert into table at scope level 1

❑ For b:

❑scope level = 1

❑*name = “b", kind = parameter, type = float, scope_level = 1, decl_line = 4, offset = param_offset(b)*

❑Insert into table at scope level 1

Step 5: read x (inside foo)

❑ scope level = 1

❑Create symbol as:

name = "x", kind = var, type = int, scope_level = 1, decl_line = 5,

offset = local_offset(x), size = 4

❑Insert into table

Step 6: process x=a+1

- ❑ Lookup(x) (found in scope 1 as local)
- ❑ Lookup(a) (found in scope 1 as parameter)

Step 7: handle opening scope {

❑Call push_scope() to enter scope level 2

Step 8: read a

❑ scope level = 2

❑Create symbol as:

name = "a", kind = var, type = float, scope_level = 2, decl_line = 8

❑Insert into table

Step 9: process a=b

- ❑ Lookup(a) (found in scope 2)
- ❑ Lookup(b) (not found in scope 2)
- ❑ Lookup(b) (found in scope 1)

Step 10: handle closing scope }

- ❑Call pop_scope() to remove scope level 2 entries
- ❑Return to scope level 1

Step 11: handle closing scope }

- ❑Call pop_scope() to remove scope level 1 entries
- ❑Return to scope level 0 (global scope)

Symbol Table: at scope level 0

name	kind	type	decl_line	scope	offset	size
x	var	int	1	0	Gaddr0	4
y	var	float	2	0	Gaddr4	8
foo	function	void	4	0	—	—

Symbol Table: at scope level 1

name	kind	type	decl_line	scope	offset	size
a	parameter	int	4	1	param+8	4
b	parameter	float	4	1	param+12	8
x	var	int	5	1	local-4	4

Symbol Table: at scope level 2

name	kind	type	decl_line	scope
a	var	float	8	2