

# SYMBOL TABLE DESIGN

Two wavy lines, one dark blue and one light gray, curve across the middle of the slide.

# WHAT EXACTLY IS SYMBOL TABLE??

*Symbol tables are **data structures** that are used by compilers to hold information about source-program constructs.*

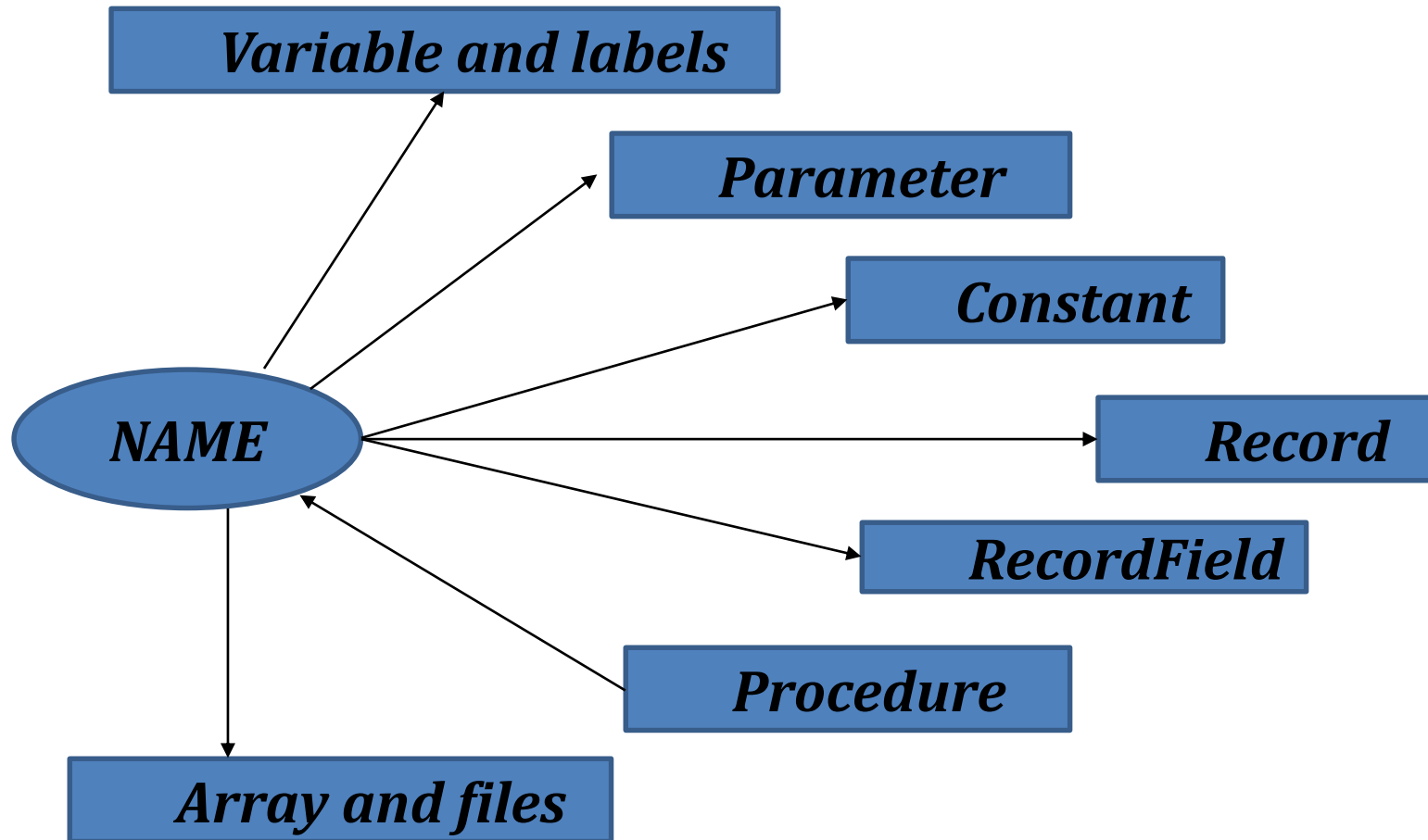
A symbol table is a necessary component because

- Declaration of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text

# INFORMATION PROVIDED BY SYMBOL TABLE

- *Given an Identifier, what is its name?*
- *What information is to be associated with a name?*
- *How do we access this information?*

# SYMBOL TABLE - NAMES



# SYMBOL TABLE-ATTRIBUTES

- Each piece of information associated with a name is called an *attribute*.
- Attributes are language dependent.
- Different classes of Symbols have different Attributes
- Examples:
  - Variables and Constants
    - Type, line number where declared, lines where referenced, scope
  - Procedure or Function
    - Number of parameters, parameters themselves, result type
  - Array
    - Dimensions, array bounds

# WHO CREATES SYMBOL TABLE??

- Identifiers and attributes are entered by the analysis phases while processing a definition (declaration) of an identifier
- In simple languages with only global variables and implicit declarations:
  - ✓ The **lexical analyser** can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
  - ✓ The **parser and/or semantic analyzer** enter identifiers and corresponding attributes

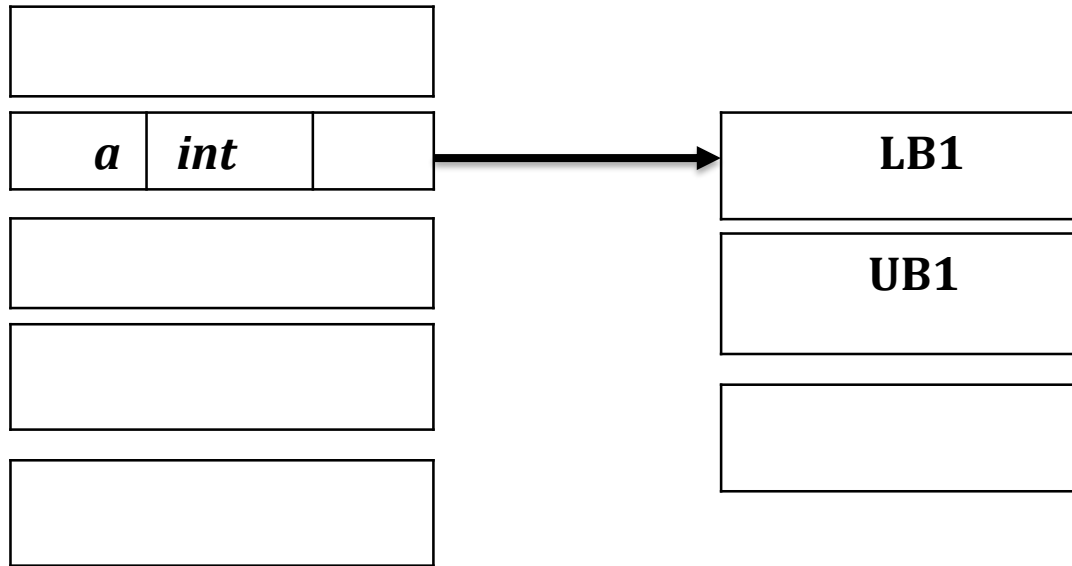
# USE OF SYMBOL TABLE

- Symbol table information is used by the analysis phases
  - To verify that used identifiers have been defined (declared)
  - To verify that expressions and assignments are semantically correct – type checking
  - To generate intermediate or target code

# IMPLEMENTATION OF SYMBOL TABLE

- Each entry in the symbol table can be implemented as a ***record*** consisting of several fields.
- These fields are ***dependent*** on the information to be saved about the name
- ***Entries in the symbol table records will not be uniform.***
  - since the information about a name depends on the usage of the name
- Therefore, some information are kept outside the symbol table
  - a pointer to this information is stored in the symbol table record.



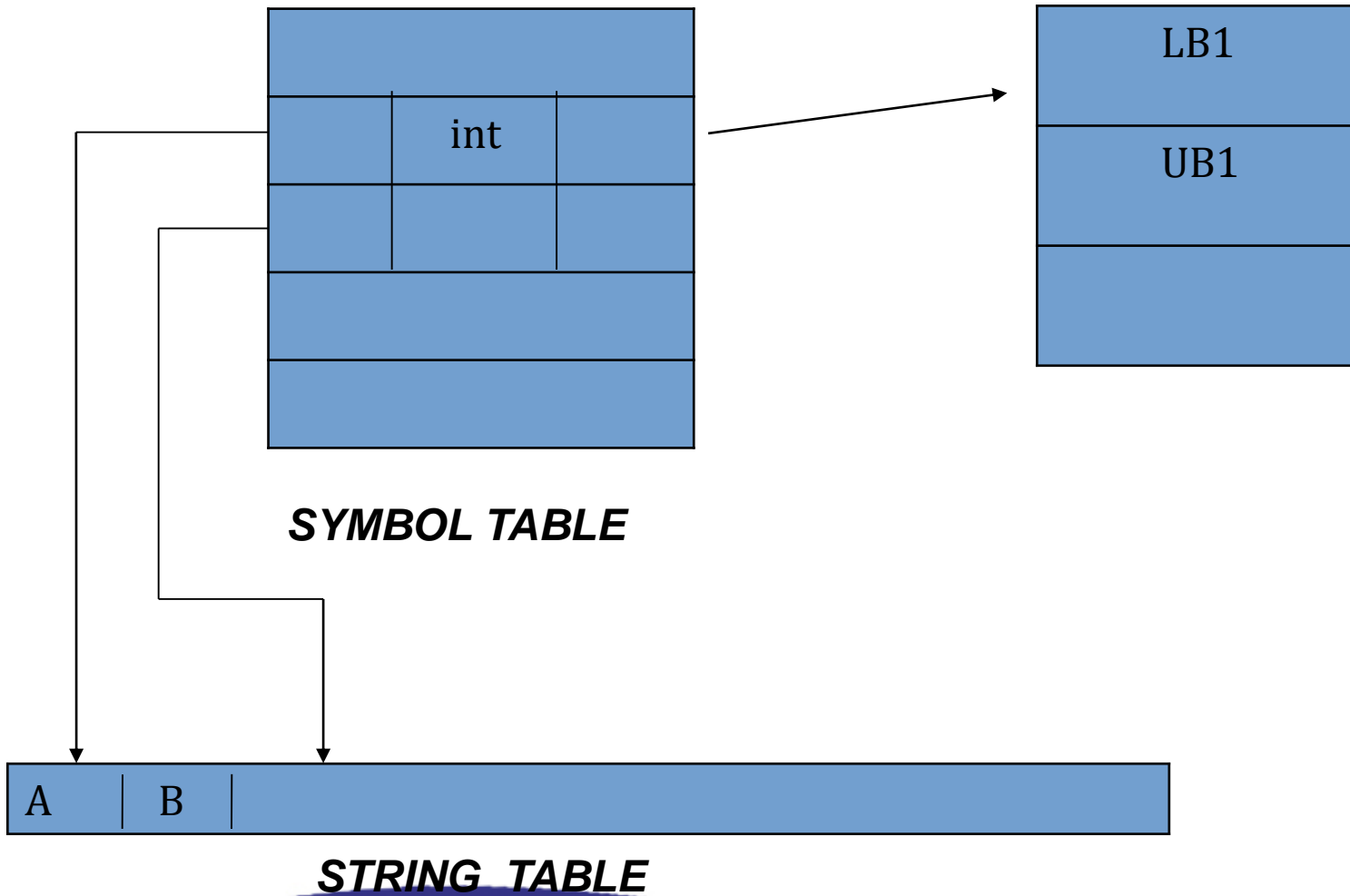


***SYMBOL TABLE***

***A pointer stored in the symbol table to remotely stored information for array a.***

# WHERE SHOULD NAMES BE HELD??

- If there is no limit for names (of the symbols)
  - A separate array of characters called a '*string table*' is used to store the name
  - a pointer to the name is kept in the symbol table record
- If there is an upper bound on the length of the name, then the name can be stored in the symbol table record itself.
- But If there is no such limit or the limit is already reached then an indirect scheme of storing name is used.



# SYMBOL TABLE AND SCOPE

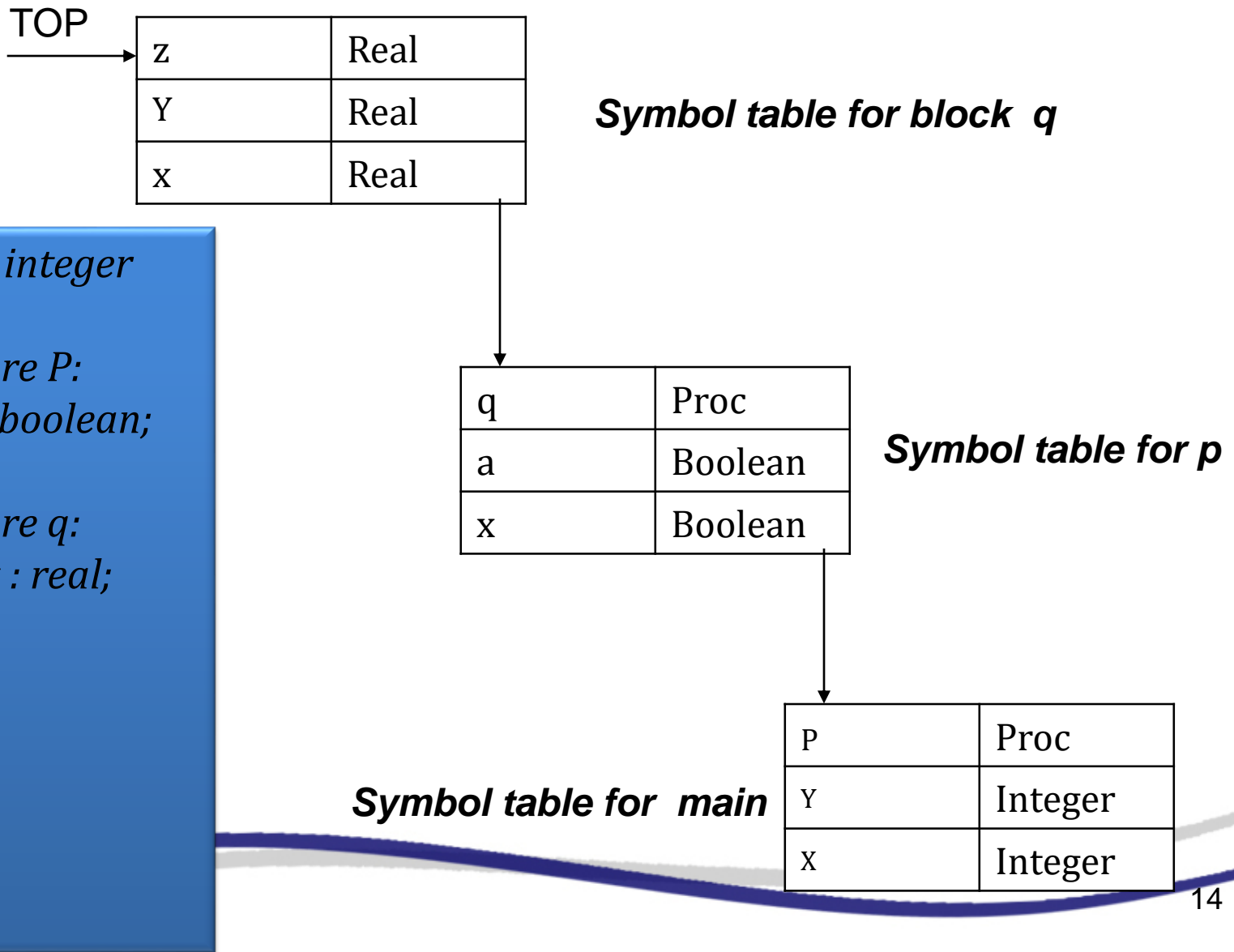
- Symbol tables need to support **multiple declarations** of the same identifier within a program.

*The scope of a declaration is the portion of a program to which the declaration applies.*

- Scopes can be implemented
  - either storing in a single table
  - or by setting up ***a separate symbol table for each scope.***

- Rules governing the scope of names in a block-structured language:
  1. *A name declared within a block B is valid only within B.*
  2. *If block B1 is nested within B2, then **any name that is valid for B2 is also valid for B1**, unless the identifier for that name is re-declared in B1.*
- A more complicated symbol table organization is required for scope rules.
- Scopes can be implemented as multiple tables.
  - Each table is list of names and their associated attributes and the tables are organized into a stack.

# SYMBOL TABLE ORGANIZATION



# NESTING DEPTH

- Another technique to represent scope information in the symbol table.
- Nesting depth of each procedure block is stored in the symbol table
- [***procedure name*** , ***nesting depth***] pairs are used as the key to accessing the information from the table.

*A nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure.*

- This number is basically a count of how many procedures are there in the referencing environment of the procedure .

*Var x,y : integer*

*Procedure P:*

*Var x,a :boolean;*

*Procedure q:*

*Var x,y,z : real;*

*begin*

*.....*

*end*

*begin*

*.....*

*End*

x	3	Real
y	3	Real
z	3	Real
q	2	Proc
a	2	Boolean
x	2	Boolean
P	1	Proc
y	1	Integer
z	1	integer



# SYMBOL TABLE DATA STRUCTURES

- Issues to consider :
- Two operations are required for symbol table
  - Insert
    - Add symbol to symbol table
  - Look UP
    - Find symbol in the symbol table (and get its attributes)
- Insertion is done only once
- Look Up is done many times
- Need Fast Look Up
  - *The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.*

# SYMBOL TABLE DATA STRUCTURES

- Unordered list: for a very small set of variables.
- Ordered linear list: insertion is expensive, but implementation is relatively easy.
- Binary search tree:  $O(\log n)$  time per operation for  $n$  variables.
- Hash table: most commonly used, and very efficient provided the memory space is adequately larger than the number of variables.

# LINKED LIST

- A linear list of records is the easiest way to implement symbol table.
  - The new names are added to the symbol table in the order they arrive.
  - Whenever a new name is to be added, the list is first searched sequentially to check if the name is already present in the table or not and if not, it is added accordingly.
- ✂ Time complexity –  $O(n)$
- ✂ Advantage – less space, additions are simple
- ✂ Disadvantages - higher access time.

# UNSORTED LIST

```
01 PROGRAM Main
02     GLOBAL a,b
03     PROCEDURE P (PARAMETER x)
04         LOCAL a
05     BEGIN {P}
06         ...a...
07         ...b...
08         ...x...
09     END {P}
10 BEGIN{Main}
11     Call P(a)
12 END {Main}
```

Look up Complexity  $O(n)$

Name	Characteristic		Other Attributes		
	Class	Scope	Declared	Referenced	Other
Main	Program	0	Line 1		
a	Variable	0	Line 2	Line 11	
b	Variable	0	Line 2	Line 7	
P	Procedure	0	Line 3	Line 11	parameter x
x	Parameter	1	Line 3	Line 8	
a	Variable	1	Line 4	Line 6	

# SORTED LIST

```

01 PROGRAM Main
02     GLOBAL a,b
03     PROCEDURE P (PARAMETER x)
04         LOCAL a
05     BEGIN {P}
06         ...a...
07         ...b...
08         ...x...
09     END {P}
10 BEGIN{Main}
11     Call P(a)
12 END {Main}

```

Look up Complexity  $O(\lg n)$

If stored as array (complex insertion)

Look up Complexity  $O(n)$

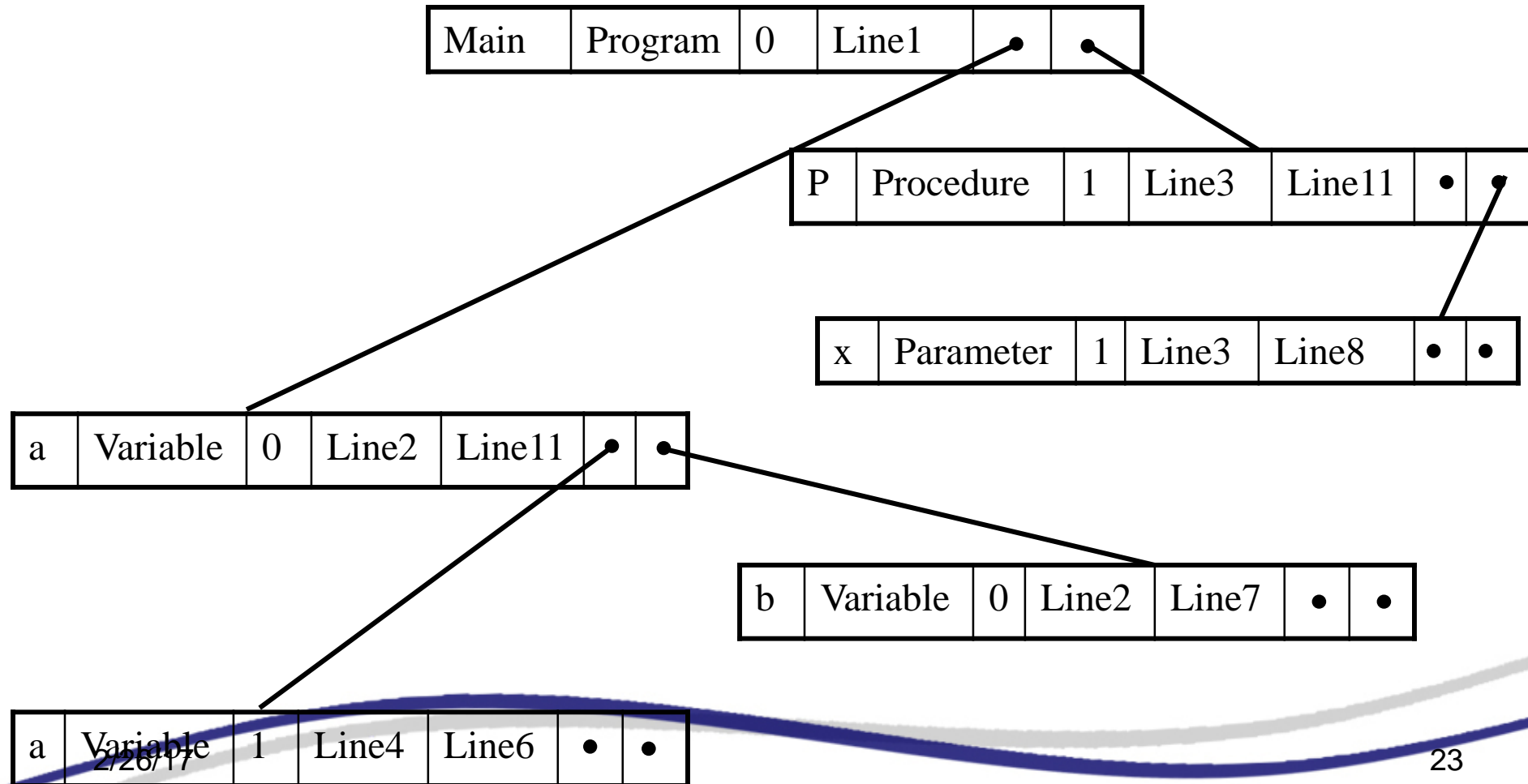
If stored as linked list (easy insertion)

Name	Characteristic Class	Other Attributes			
		Scope	Declared	Reference	Other
a	Variable	0	Line 2	Line 11	
a	Variable	1	Line 4	Line 6	
b	Variable	0	Line 2	Line 7	
Main	Program	0	Line 1		
P	Procedure	0	Line 3	Line 11	1, parameter, x
x2/26/17	Parameter	1	Line 3	Line 8	

# SEARCH TREES

- Efficient approach for symbol table organisation
- We add two links left and right in each record in the search tree.
- Whenever a name is to be added, first the name is searched in the tree.
- If it does not exist then a record for the new name is created and added at the proper position.
- This has alphabetical accessibility.

# BINARY TREE



# BINARY TREE

*Lookup complexity if tree balanced*  
 $O(\lg n)$

*Lookup complexity if tree  
unbalanced*  
 $O(n)$

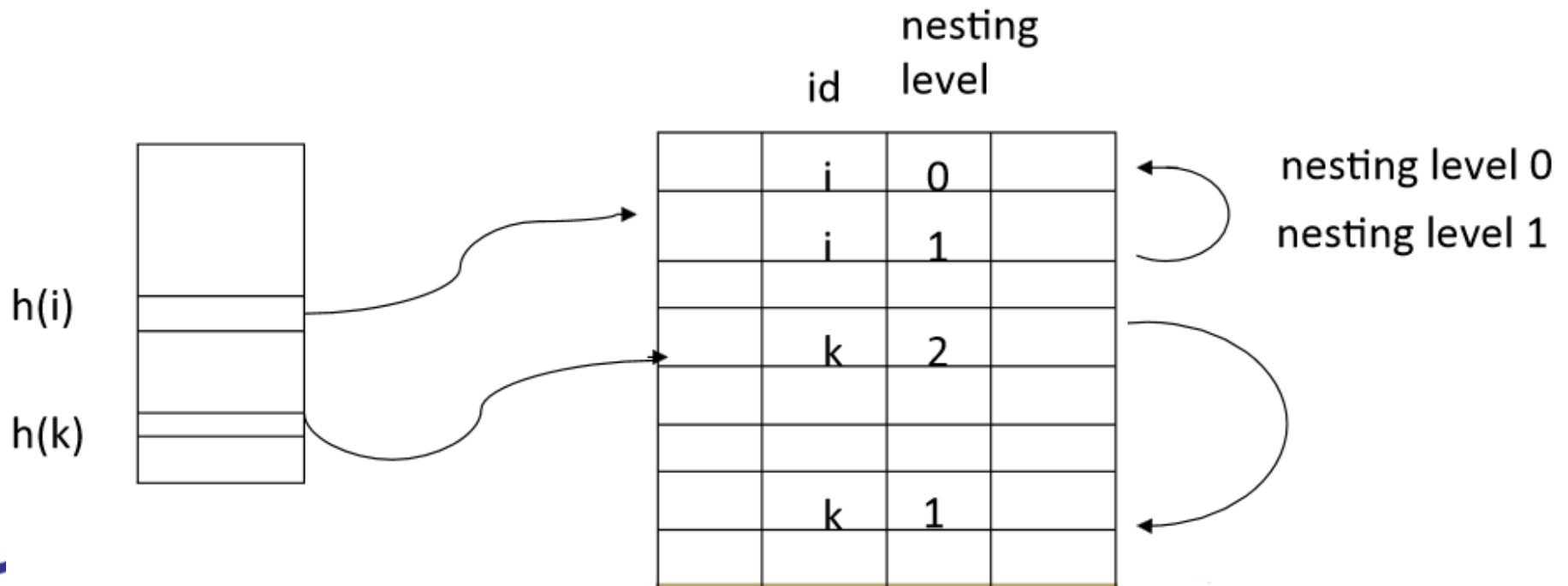


# HASH TABLE

- Table of  $k$  pointers numbered from zero to  $k-1$  that point to the symbol table and a record within the symbol table.
- To enter a name in the symbol table, the hash value of the name is found by applying a suitable hash function.
- The hash function maps the name into an integer between zero and  $k-1$  and this value is used as an index in the hash table.

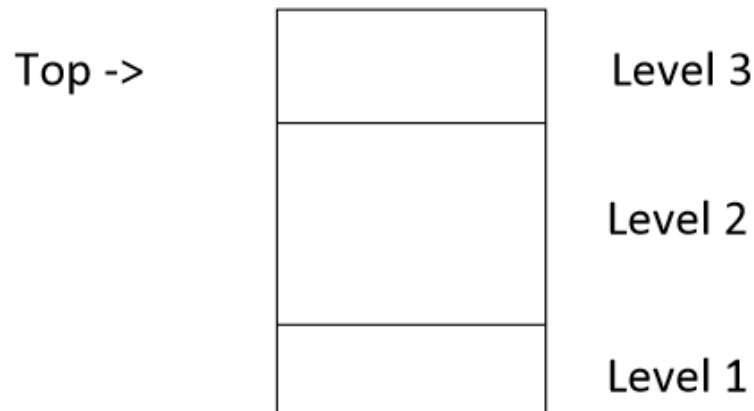
# Single Hash Table to Implement Symbol Table

- Link together different entries for the same identifier and associate nesting level with each occurrence of same name.
- The first one is the latest occurrence of the name, i.e., highest nesting level



# Single Hash Table to Implement Symbol Table

- During exit from a procedure - dissociate all entries of the nesting level we are exiting
  1. Search for the correct items to remove – rehash -- expensive
  2. Use extra pointer in each element to link all items of the same scope (scope chain)
  3. Use an active ST stack that has entries for each scope

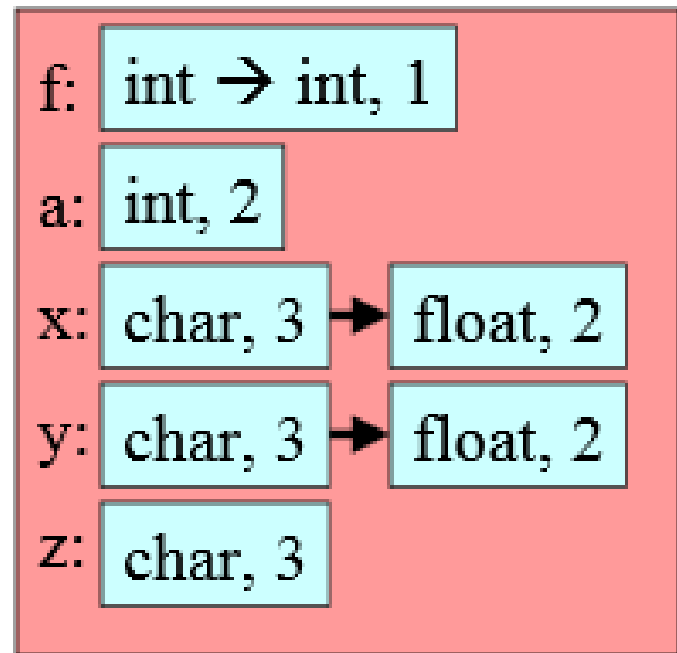


Pop entry and delete from hash table - keep stack entries around

# Example

```
int f (int a) {  
    float x, y;  
    while (...) {  
        char x, y, z;  
    }  
}  
  
void g () {  
    int x;  
    f(1);  
}
```


- After processing the declarations inside the while loop:



# List of Operations

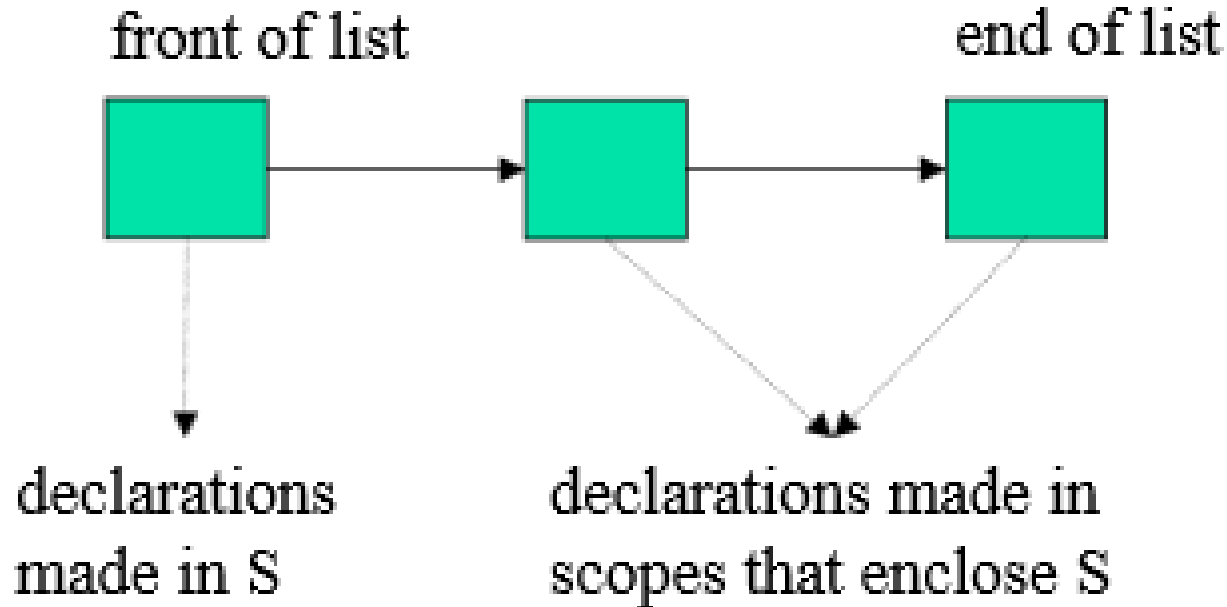
- On scope entry:
  - increment the current level number
- To process a declaration of x:
  - look up x in the symbol table
    - If x is there, fetch the level number from the first list item.
      - If that level number = the current level then issue a "multiply declared variable" error;
      - otherwise, add a new item to the front of the list with the appropriate type and the current level number

# List of Operations

- To process a use of  $x$ :
    - look up  $x$  in the symbol table
    - If it is not there, then issue an "undeclared variable" error
  - On scope exit:
    - scan all entries in the symbol table, looking at the first item on each list
    - If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry)
    - Finally, decrement the current level number
- 

# A List of Hash Table

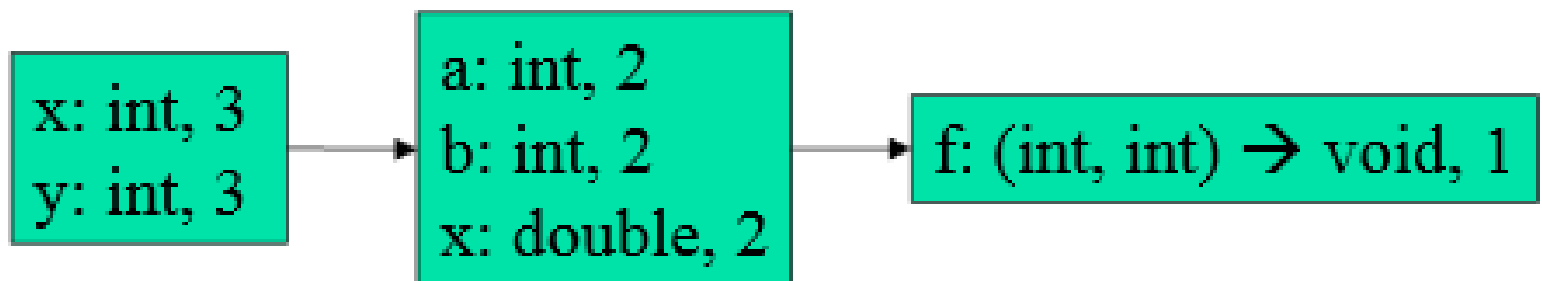
- One hash table for each currently visible scope
- At a given point, have symbol table accessible for each nesting level  $\leq$  current nesting level



# Example


```
void f(int a, int b) {  
    double x;  
    while (...) { int x, y; ... }  
}  
void g() { f(); }
```

After processing declarations inside the while loop:





# A List of Hash Table: Operations

- On scope entry:
    - increment the current level number and add a new empty hashtable to the front of the list.
  - To process a declaration of x:
    - look up x in the first table in the list
      - If it is there, then issue a "multiply declared variable" error;
      - otherwise, add x to the first table in the list
- 

# A List of Hash Table: Operations

- To process a use of  $x$ :
  - look up  $x$  starting in the first table in the list;
    - if it is not there, then look up  $x$  in each successive table in the list
    - if it is not in *any* table then issue an "undeclared variable" error
- On scope exit,
  - remove the first table from the list and decrement the current level number

# Insert Method Name

- Method names belong in the hashtable for the outermost scope
  - Not in the same table as the method's variables
- For example, in the previous example:
  - Method name *f* is in the symbol table for the outermost scope
  - Name *f* is *not* in the same scope as parameters *a* and *b*, and variable *x*
  - This is so that when the use of name *f* in method *g* is processed, the name is found in an enclosing scope's table

# Running time for each operation

## 1. **Scope entry:**

- time to initialize a new, empty hashtable;
- probably proportional to the size of the hashtable

## 2. **Process a declaration:**

- using hashing, constant expected time ( $O(1)$ )

## 3. **Process a use:**

- using hashing to do the lookup in each table in the list, the worst-case time is  $O(\text{depth of nesting})$ , when every table in the list must be examined

## 4. **Scope exit:**

- time to remove a table from the list, which should be  $O(1)$  if garbage collection is ignored
- 

# Static vs. Dynamic Scoping

- Scoping is generally divided into two classes:
  - Static Scoping
  - Dynamic Scoping



# Static Scope

- Static Scoping:
  - also called lexical scoping
  - the bindings between name and objects can be determined by examining the program text
- a property of the program text and unrelated to the run time call stack.
- programmer can figure out the scope just by looking at the code.



# Static Scope

- Typically, the current binding for a given name is the one encountered most recently in a top-to-bottom scan of the program.
- The compiler first searches the current block, then in the surrounding blocks successively and finally in the global variables.
- C, C++ and Java, variables are always statically (or lexically) scoped



# Static scope rules

- The simplest static scope rule has only a single, global scope (e.g., early Basic).
- A more complex scope rule distinguishes between global and local variables (e.g., Fortran).
- Languages that support nested functions (e.g., Pascal, Algol) require an even more complicated scope rule.
- Closest Nested Scope Rule
  - A name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is hidden by another declaration of the same name in one or more nested scopes.





# Hole and Qualifier

- A name-to-object binding that is hidden by a nested declaration of the same name is said to have a hole in its scope.
- In most languages, the object whose name is hidden is inaccessible in the nested scope.
- Some languages allow accesses to the outer meaning of a name by applying a qualifier or scope resolution operator.



```
// A C program to demonstrate  
//static scoping.
```

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

```
int x = 10;
```

```
// Called by g()
```

```
int f()
```

```
{
```

```
    return x;
```

```
}
```

```
//g() has its own variable  
// named as x and calls f()
```

```
int g()
```

```
{
```

```
    int x = 20;
```

```
    return f();
```

```
}
```

```
int main()
```

```
{
```

```
    printf("%d", g());
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

```
// A C program to demonstrate  
//static scoping.
```

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

```
int x = 10;
```

```
// Called by g()
```

```
int f()
```

```
{
```

```
    return x;
```

```
}
```

**Output of the program is 10**

```
//g() has its own variable
```

```
// named as x and calls f()
```

```
int g()
```

```
{
```

```
    int x = 20;
```

```
    return f();
```

```
}
```

```
int main()
```

```
{
```

```
    printf("%d", g());
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

# Dynamic Scoping

- The bindings between names and objects depend on the flow of control at run time.
- uncommon in modern languages
- In general, the flow of control cannot be predicted in advance by the compiler
- Languages with dynamic scoping tend to be interpreted rather than compiled.



```
// Dynamic scoping.  
#include<stdio.h>  
int x = 10;  
// Called by g()  
int f()  
{  
    return x;  
}
```

```
// g() has its own variable  
// named as x and calls f()  
int g()  
{  
    int x = 20;  
    return f();  
}  
  
main()  
{  
    printf(g());  
}
```

```
// Dynamic scoping.  
#include<stdio.h>  
int x = 10;  
// Called by g()  
int f()  
{  
    return x;  
}
```

**Output of the program is 20**

```
// g() has its own variable  
// named as x and calls f()  
int g()  
{  
    int x = 20;  
    return f();  
}  
  
main()  
{  
    printf(g());  
}
```

- To accommodate static scoping most compilers never delete anything from the symbol table. Instead, they manage visibility using enter scope and leave scope operations.
- Though implementation varies from compiler to compiler.



# Static vs. Dynamic Scope

- Static scope rules match the reference (use of variable) to the closest lexically enclosing declaration.
- Dynamic scope rules choose the most recent active declaration at runtime.





## Scoping example

```
main P() {  
    int X;  
    void A() {  
        X = X + 1;  
        print(X);  
    }  
    void B() {  
        int X;  
        X = 17;  
        A();  
    }  
    X = 23;  
    B();  
}
```

The main program calls B, then B calls A. When A prints X, which X is it?

With static scoping, it is X in P, the enclosing block of procedure A. The number printed will be 24.


With dynamic scoping (search is done up the chain of calls), it is X in B, the most recently entered block with a declaration of X. The number printed will be 18.

# Static and Dynamic Links

- A static link points to the activation record of its lexically-scoped parent.
- Static chain is a chain of static links connecting certain activation record instances in the stack.
- A dynamic link points to its caller activation record.



# Implementing a Symbol Table for Object-Oriented Programs

- Single global table to map class names to class symbol tables
    - Created in pass over class definitions
    - Used in remaining parts of compiler to check field/method names and extract information
  - All global tables persist throughout the compilation (and beyond in a real Java compiler...)
  - One or multiple symbol tables for each class: 1 entry for each method/field; Contents: type information, public/private, storage locations
  - Local symbol table for each method: 1 entry for each local variable or parameter; Contents: type information, storage locations
- 

# Scope in Object oriented programming

- In OOP, scope extends beyond functions and the main program.
- Each class defines a scope that cover every function's scope in that class.
- Inheritance and access modifiers also make some variables and functions visible outside their scope.
- The scope for a derived class is often placed inside of the scope of a base class

