

# Bindings: Names and Attributes

- Names are a fundamental abstraction in languages to denote entities
- Meanings associated with these entities is captured via attributes associated with the names
- Attributes differ depending on the entity:
  - location (for variables)
  - value (for constants)
  - formal parameter types (functions)
- Binding: Establishing an association between name and an attribute.

# Names


- **Names** or **Identifiers** denote various language *entities*:
  - Constants
  - Variables
  - Procedures and Functions
  - Types, ...

- Entities have *attributes*

<i>Entity</i>	<i>Example Attributes</i>
Constants	<u>type</u> , <u>value</u> , ...
Variables	<u>type</u> , <u>location</u> , ...
Functions	<u>signature</u> , <u>implementation</u> , ...

# Attributes

- Attributes are associated with names (to be more precise, with the entities they denote).
- Attributes describe the *meaning* or *semantics* of these entities.

<code>int x;</code> 	There is a <u>variable</u> , named <u>x</u> , of type <u>integer</u> .
<code>int y = 2;</code>	Variable named <u>x</u> , of type integer, with initial value 2.
<code>Set s=new Set();</code>	Variable named <u>s</u> , of type <u>Set</u> that refers to an object of class <u>Set</u>

- An *attribute* may be
  - static: can be determined at translation (compilation) time, or
  - dynamic: can be determined only at execution time.

# Static and Dynamic Attributes

- int x;
  - The type of x can be statically determined;
  - The value of x is dynamically determined;
  - The location of x (the element in memory will be associated with x) can be statically determined if x is a global variable.
- Set s = new Set();
  - The type of s can be statically determined.
  - The value of s, i.e. the object that s refers to, is dynamically determined.



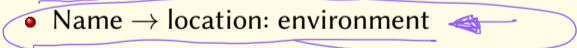

Static vs. Dynamic specifies the *earliest* time the attribute can be computed  
... not when it is computed in any particular implementation.

# Binding

“Binding” is the process of associating attributes with names.

- Binding time of an attribute: whether an attribute can be computed at translation time or only at execution time.
- A more refined classification of binding times:
  - **Static:**
    - Language definition time (e.g. boolean, char, etc.)
    - Language implementation time (e.g. maxint, float, etc.)
    - Translation time (“compile time”) (e.g. value of n in const int n = 5;)
    - Link time (e.g. the definition of function f in extern int f();)
    - Load time (e.g. the location of a global variable, i.e., where it will be stored in memory)
  - Dynamic:
    - Execution time

# Binding Time (Continued)

- Examples
  - type is statically bound in most langs
  - value of a variable is dynamically bound
  - location may be dynamically or statically bound
- Binding time also affects where bindings are stored
  - Name → type: symbol table 
  - Name → location: environment  
  - Location → value: memory 

# Declarations and Definitions

- **Declaration** is a syntactic structure to establish bindings.

- `int x;`
- `const int n = 5;`
- `extern int f();` ←
- `struct foo;` ←







*extern int x;*

- **Definition** is a declaration that usually binds *all* static attributes.

- `int f() { return x; }`
- `struct foo { char *name; int age; };`

- Some bindings may be implicit, i.e., take effect without a declaration.
  - FORTRAN: All variables beginning with [i-nl-N] are integers; others are real-valued.
  - PROLOG: All identifiers beginning with [A-Z\_] are variables.

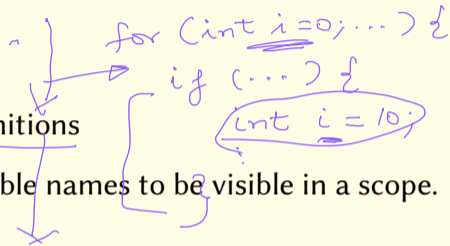
# Scopes

- Region of program over which a declaration is in effect
  - i.e. bindings are maintained
- Possible values
  - Global 
  - Package or module 
  - File 
  - Class 
  - Procedure 
  - Block 



# Visibility

- Redefinitions in inner scopes supercede outer definitions
- Qualifiers may be needed to make otherwise invisible names to be visible in a scope.
- Examples
  - local variable superceding global variable
  - names in other packages.
  - private members in classes.



# Symbol Table

Maintains bindings of attributes with names:

$$\text{SymbolTable} : \text{Names} \longrightarrow \text{Attributes}$$

- In a compiler, only *static attributes* can be computed; thus:

$$\text{SymbolTable} : \text{Names} \longrightarrow \text{StaticAttributes}$$

- While execution, the names of entities no longer are necessary: only locations in memory representing the variables are important.

$$\text{Store} : \text{Locations} \longrightarrow \text{Values}$$

(*Store* is also called as *Memory*)

- A compiler then needs to map variable names to locations.

$$\text{Environment} : \text{Names} \longrightarrow \text{Locations}$$

# Blocks and Scope

- Usually, a name refers to an entity within a given context.

```
class A {  
    int x;  
    double y;  
    int f(int x) { // Parameter "x" is different from field "x"  
        B b = new B();  
        y = b.f(); // method "f" of object "b"  
        this.x = x;  
        ...  
    }  
}
```

- The context is specified by “Blocks”
  - Delimited by “{” and “}” in C, C++ and Java
  - Delimited by “begin” and “end” in Pascal, Algol and Ada.

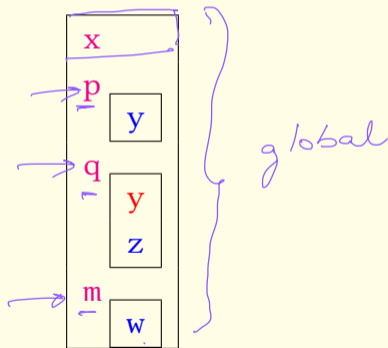
# Scope

**Scope:** Region of the program over which a binding is maintained.

```

int x;
void p(void) {
  char y;
  ...
}
void q(int y) {
  double z;
  ...
}
m() {
  int w;
  ...
}

```



# Lexical Scope

**Lexical scope:** the scope of a binding is limited to the block in which its declaration appears.

- The bindings of local variables in C, C++, Java follow lexical scope.
- Some names in a program may have a “meaning” outside its lexical scope.  
e.g. field/method names in Java
  - Names must be *qualified* if they cannot be resolved by lexical scope.  
e.g. `a.x` denotes the field `x` of object referred by `a`.  
`a.x` can be used even outside the lexical scope of `x`.
- Visibility of names outside the lexical scope is declared by visibility modifiers (e.g. public, private, etc.)

# Namespaces

- Namespaces are a way to specify “contexts” for names.
  - `www.google.com`:
    - The trailing `com` refers to a set of machines
    - `google` is subset of machines in the set `com`  
`google` is interpreted here in the context of `com`
    - `www` is a subset of machines in the set `google`  
`www` is interpreted here in the context of `google.com`
  - Other common use of name spaces: directory/folder structure.
- Names should be fully qualified if they are used outside their context.  
e.g. `Stack::top()` in C++, `List.hd` in OCAML.
- Usually there are ways to declare the context *a priori* so that names can be specified without qualifying them.

`std::list<...>`

packages

C++

}  
}

# Lifetimes

The lifetime of a binding is the interval during which it is effective.

<pre> int fact(int n) {   int x;   if (n == 0)     return 1;   else {     x = fact(n-1);     return x * n;   } } </pre>	<p><u>fact: n = 2</u> ←</p> <hr/> <p>fact: n = 2 → <u>fact: n = 1</u></p> <hr/> <p>fact: n = 2 → <u>fact: n = 1</u> → <u>fact: n = 0</u> ←</p> <hr/> <p>fact: n = 2 → <u>fact: n = 1, x = 1</u></p> <hr/> <p>fact: n = 2, x = 1 ←</p> <hr/> <p><u>2</u></p>
---	---

*fact(2)*

- Each invocation of `fact` defines new variables `n` and `x`.
- The lifetime of a binding may exceed the scope of the binding.
  - e.g., consider the binding `n=2` in the first invocation of `fact`.
  - Call to `fact(1)` creates a new local variable `n`.
  - But the first binding is still effective.

# Symbol Table

- Uses data structures that allow efficient name lookup operations in the presence of scope changes.
- We can use
  - hash tables to lookup attributes for each name
  - a scope stack that keeps track of the current scope and its surrounding scopes
    - the top most element in the scope stack corresponds to the current scope
    - the bottommost element will correspond to the outermost scope.



# Support for Scopes

- Lexical scopes can be supported using a scope stack as follows:
- Symbols in a program reside in multiple hash tables
  - In particular, symbols within each scope are contained in a single hash table for that scope
- At anytime, the scope stack keeps track of all the scopes surrounding that program point.
- The elements of the stack contain pointers to the corresponding hash table.

## Support for Scopes (Continued)

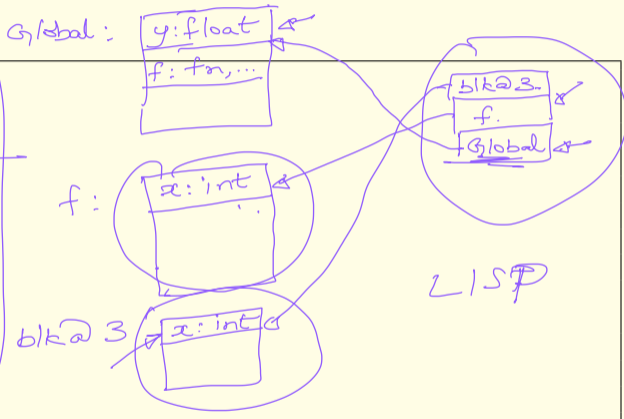
- To lookup a name
- Symbols in a program reside in multiple hash tables
  - Start from the hash table pointed to by the top element of the stack.
  - If the symbol is not found, try hash table pointed by the next lower entry in the stack.
  - This process is repeated until we find the name, or we reach the bottom of the stack.
- Scope entry and exit operations modify the scope stack appropriately.
  - When a new scope is entered, a corresponding hash table is created. A pointer to this hash table is pushed onto the scope stack.
  - When we exit a scope, the top of the stack is popped off.

# Example

```

1: float y = 1.0
2: void f(int x) {
3:   for(int x=0; ...) {
4:     float x1 = x + y;
5:   }
6: }
7:   float x = 1.0;
8: }
9: }
10: main() {
11:   float y = 10.0;
12:   f(1);
13: }

```



# illustration

- At (1)
  - We have a single hash table, which is the global hash table.
  - The scope stack contains exactly one entry, which points to this global hash table.
- When the compiler moves from (1) to (2)
  - The name `y` is added to the hash table for the current scope.
  - Since the top of scope stack points to the global table, “`y`” is being added to the global table.
- When the compiler moves from (2) to (3)
  - The name “`f`” is added to the global table, a new hash table for `f`'s scope is created.
  - A pointer to `f`'s table is pushed on the scope stack.
  - Then “`x`” is added to hash table for the current scope.

# Static vs Dynamic Scoping

- Static or lexical scoping:
  - associations are determined at compile time
  - using a sequential processing of program
- Dynamic scoping:
  - associations are determined at runtime
  - processing of program statements follows the execution order of different statements

# Example

- if we added a new function "g" to the above program as follows:

```
void g() {  
    int y;  
    f();  
}
```

- Consider references to the name "y" at (4).
  - With static scoping, it always refers to the global variable "y" defined at (1).
  - With dynamic scoping
    - if "f" is called from main, "y" will refer to the float variable declared in main.
    - If "f" is invoked from within "g", the same name will refer to the integer variable "y" defined in "g".

## Example (Continued)

- Since the type associated with “y” at (4) can differ depending upon the point of call, we cannot statically determine the type of “y” .
- Dynamic scoping does not fit well with static typing.
- Since static typing has now been accepted to be the right approach, almost all current languages (C/C++/Java/OCAML/LISP) use static scoping.

## Scopes in OCAML:

- Most names are at the “top-level,” which corresponds to global scope.
- Formal parameters of functions are within the scope of the function.
- “Let” statement introduces new bindings whose scope extends from the point of binding to the end of the let-block.
- Example

```
let v =  
    let x = 2  
    and y = 3  
in x*y;;
```