

# Name, Binding and Scope

Dr. Nguyen Hua Phung  
nhphung@hcmut.edu.vn

HCMC University of Technology, Viet Nam

November 7, 2020

- Name - character string used to represent something else.
  - identifiers,
  - operators (+, &, \*).
- Use **symbol** instead of **address** to refer an entity.
- Abstraction

## Definition

- Binding - the operation of associating two things.
- Binding time - the moment when the binding is performed.

## Some issues

- Early binding vs. Late binding
- Static binding vs. Dynamic binding
- Polymorphism - A name is bound to more than one entity.
- **Alias - Many names are bound to one entity.**

- Language design time
- Language implementation time
- Programming time
- Compilation time
- Linking time
- Load time
- Runtime

- *Object* - any entity in the program.
- *Object lifetime* - the period between the object creation and destruction.
- *Binding lifetime*
- *Dangling reference*

```
p = new int;
```

```
q = p;
```

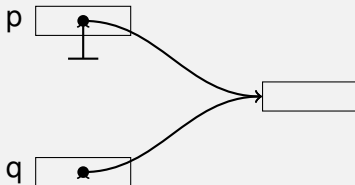
```
delete p;
```

```
*q;
```

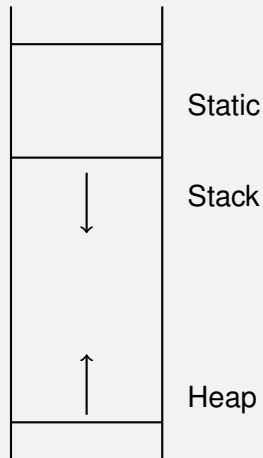
- *Leak memory - Garbage*

```
p = new int;
```

```
p = null;
```



- Static
- Stack Dynamic
- Explicit Heap Dynamic
- Implicit Heap Dynamic



## Definition

Scope of a binding is the textual region of the program in which the binding is effective.

## Static vs. Dynamic

- Static scope, or lexical scope, is determined during compilation
  - Current binding - in the block most closely surround
  - Global scope
  - Local static scope
- Dynamic scope is determined at runtime.
  - Current binding - the most recently execution but not destroyed

## Definition

A block is a textual region, which can contain declarations to that region

Example,

```
procedure foo()  
var x:integer;  
begin  
    x := 1;  
end;  
  
    {  
        int x;  
        x = 1;  
    }
```



- A reference to an identifier is always bound to **its most local declaration**
- A declaration is **invisible** outside the block in which it appears
- Declarations in enclosing blocks are **visible** in **inner blocks**, unless they have been **re-declared**
- Blocks may be named and its name declaration is considered as a **local declaration of outer** block.

## Example on Static scope

```
var A, B, C: real; //1
procedure Sub1 (A: real); //2
    var D: real;
    procedure Sub2 (C: real);
        var D: real;
        begin
            ... C:= C+B; ...
        end;
    begin
        ... Sub2(B); ...
    end;
begin
    ... Sub1(A); ...
end.
```

Variable	Scope
A:real //1	Main
B:real //1	Main, Sub1, Sub2
C:real //1	Main, Sub1
A:real //2	Sub1, Sub2
...	

```
procedure Big is
  X : Real;
  procedure Sub1 is
    X : Integer;
    begin -- of Sub1
      ...
    end; -- of Sub1
  procedure Sub2 is
    begin -- of Sub2
      ... X ...
    end; -- of Sub2
begin -- of Big
  ...
end; -- of Big
```

X in Sub2 ?

Calling chain:

Big  $\rightarrow$  Sub1  $\rightarrow$  Sub2

X  $\Rightarrow$  X:Integer in Sub1

Calling chain:

Big  $\rightarrow$  Sub2

X  $\Rightarrow$  X:Real in Big

- The **referencing environment** of a **statement** is the collection of all names that are visible to the statement
- In a **static-scoped** language, it is the local names plus all of the visible names in all of the enclosing scopes
- In a **dynamic-scoped** language, the referencing environment is the local bindings plus all visible bindings in all active subprograms

## Example on Static-scoped Language

```
var A, B, C: real; //1
procedure Sub1 (A: real); //2
    var D: real;
    procedure Sub2 (C: real);
        var D: real;
        begin
            ... C:= C+B; ...
        end;
    begin
        ... Sub2(B); ...
    end;
begin
    ... Sub1(A); ...
end.
```

Function	Referencing Environment
Main	A, B, C, Sub1
Sub1	A, B, C, D, Sub1, Sub2
Sub2	A, B, C, D, Sub1, Sub2

# Example on Dynamic-scoped Language

	main	→ sub2	→ sub2	→ sub1
void sub1() {	c	b	b	a
int a, b;	d	c	c	b
...				
} /* end of sub1 */				
void sub2() {				
int b, c;				
...				
sub1;				
} /* end of sub2 */				
void main() {				
int c, d;				
...				
sub2();				
} /* end of main */				

Frame	Referencing Environment
main	c → o1, d → o2
sub2	b → o3, c → o4, d → o2
sub2	b → o5, c → o6, d → o2
sub1	a → o7, b → o8, c → o6, d → o2

```
def <func-name>(<parameter-list >):  
    <stmt-list >
```

- nested function
- pass-by-value (pointer)
- matched by position and by name
- default value
- arbitrary parameters
- arbitrary keyword parameters
- return statement

```
def outer(x):  
    y = x + 1  
    def inner(z):  
        return z + 1  
    return inner(y)  
print(outer(3)) => 5  
print(inner(2)) => wrong
```

- **inner** function is visible inside **outer** but invisible outside **outer**





```
def foo(param1, param2 = 0):  
    print(param1, param2)  
print(foo(1,2)) => 1 2  
print(foo(param2 = 2,param1 = 1)) => 1 2  
print(foo(1)) => 1 0
```

```
def my_func(*kids):  
    print("My third child is" + kids[2])  
my_func('Tuong', 'Ca', 'Mam', 'Muoi')
```

- Allow arbitrary number of arguments
- Access the parameter as a *tuple*
- Define after normal parameters



```
def my_func(**rec):  
    for x,y in rec.items():  
        print(x,y)  
my_func(ho='nguyen', ten='thi ha',  
        namsinh=1996, mssv='0123456')
```

- Allow arbitrary number of keyword arguments
- Access the parameter as a *dictionary*
- Define after normal and arbitrary parameters

- Syntax:

**return** (<exp> ( , <exp>)\*)?

- Example:

```
def my_func(x):  
    x = 2  
    return x, x+2  
a, b = my_func(0)  
print(a, b) => 2 4
```

- Stop executing of a function call and return the result
- If no expression after **return**, *None* is returned
- If many expressions after **return**, a *tuple* is returned

- Read: Block rule, where a function is a block
  - ⇓ Local
  - ⇓ Nonlocal
  - ⇓ Global
    - Built-in or imported environments
- Write: **global**, **nonlocal**

```
from functools import *  * => imported env.  
x = 3                    => global  
def f():  
    y = 4                => nonlocal of g  
    def g():  
        t = 2            => local of g  
        print(z)
```

- ↓ declaration of z is looked firstly in local environment
- ↓ and then in nonlocal environments that enclose the local
- ↓ and then in global environment
- and lastly in imported environments

```
1  x, y = 3, 4           => x, y: global
2  def f():
3      x = 2             => x: local of f
4      return x + y      => x: local of f; y global
5  def g():
6      global x
7      x = 2             => x: global
8      return 2 + x
9  f()
10 print(x)              => 3
11 g()
12 print(x)              => 2
```

- firstly assigning to a variable makes the declaration of the variable in the current environment
- to assign to a global variable in a function, the declaration of **global** is required

```
1 x,y = 3,4           => x,y: global
2 def f():
3     x,z = 2,5       => x,z: local of f
4     def g():
5         nonlocal x
6         x = 2 * y    => x: nonlocal=>local of f;y:global
7         return z + x => x,z:nonlocal=>local of f;
8     print(g())       => 13
9     print(x)         => 8
10 f()
11 print(x)            => 3
```

- like **global** but for **nonlocal** variables



- Name
- Binding
- Scope
- Referencing Environment



, Maurizio Gabbrielli and Simone Martini, Programming Languages: Principles and Paradigms, Chapter 4, Springer, 2010.