

Programming Language Basics

This section explains important terminology and distinctions that appear in the study of programming languages. Various programming language concepts are

- The Static / Dynamic Distinction
- Environments and States
- Static Scope and Block Structure
- Explicit Access Control
- Dynamic Scope
- Parameter Passing Mechanisms
- Aliasing

1. The Static / Dynamic Distinction

This decides what decisions the compiler can make about a programming issues while designing a compiler for a programming language. Compiler can use either static policy or dynamic policy to decide the issues.

Static policy: issue is decided at *compile time*

Dynamic policy: issue is decided at *run time*

One issue is scope of declaration. This helps to determine the declaration for the variable that is referred.

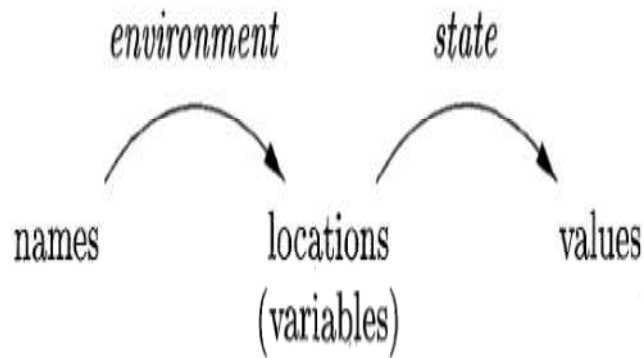
Static scope or lexical scope: Scope is determined by looking for declaration only at the program.

Dynamic scope: Scope is determined by looking for several different declarations.

Most languages, such as C and Java, use static scope.

2. Environments and States

Two-stage mapping from names to values is given below.



Environment: is a mapping from names to locations (variables) in the store.

State: is a mapping from locations in store to their values. Environments change according to the scope rules of a language.

Example:

Consider the following ‘C’ fragment. Variable *i* is declared both as global and local variable. Use of *i* in *f()* refers to the local variable. However, use of *i* out of *f()* refers to global variable.

```

...
int i;           /* global i      */
...
void f(...) {
    int i;       /* local i      */
    ...
    i = 3;       /* use of local i */
    ...
}
...
x = i + 1;      /* use of global i */

```

The environment and state mappings are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations. Most binding of names to locations is dynamic. Some declarations, such as the global *i* can be given a location in the store once and for all, as the compiler generates object code.

2. *Static versus dynamic binding* of locations to values. The binding of locations to values is generally dynamic as well, since we cannot tell the value in a location until we run the program. Declared constants are an exception. For instance, the C definition

```
#define ARRAYSIZE 1000
```

binds the name **ARRAYSIZE** to the value 1000 statically.

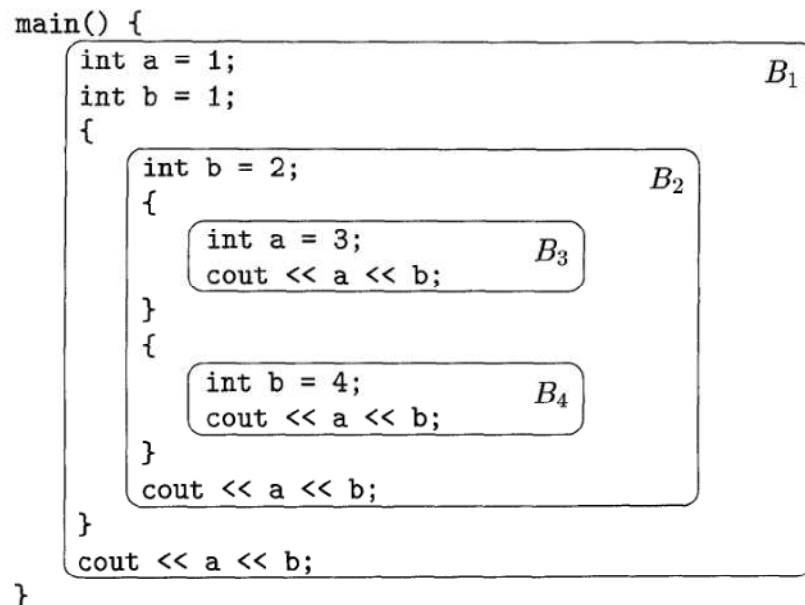
3. Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure (blocks). The scope of a declaration is determined implicitly by where the declaration appears in the program.

Example 1:

Consider, the following ‘C’ segment, which contains 4 blocks with several definitions of variables a and b.

```
main() {  
    int a = 1;  
    int b = 1;  
    {  
        int b = 2;  
        {  
            int a = 3;  
            cout << a << b;  
        }  
        {  
            int b = 4;  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```



Scope of declarations for the above code segment is given in the following table.

DECLARATION	SCOPE
<code>int a = 1;</code>	$B_1 - B_3$
<code>int b = 1;</code>	$B_1 - B_2$
<code>int b = 2;</code>	$B_2 - B_4$
<code>int a = 3;</code>	B_3
<code>int b = 4;</code>	B_4

Scope of `int a = 1;` is from block B1 upto B3. In B3, variable 'a' is redefined. In static scoping, a variable used in a block is determined from the declaration within the block. Otherwise, it is determined from the outer block in which the block is nested. If the declaration is not found even in the outer blocks, then error is generated. The output generated by the code segment is given as follows.

`cout << a << b;` in B3 : 3 2 (a from B3 and b from B2)

`cout << a << b;` in B4 : 1 4 (a from B1 and b from B4)

`cout << a << b;` in B2 : 1 2 (a from B1 and b from B2)

`cout << a << b;` in B1 : 1 1 (a from B1 and b from B1)

Example 2:

For the block-structured C code, indicate the values assigned to w, x, y, and x.

```

int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;

```

$w = 6 + 7 = 13$

$x = 4 + 5 = 9$

$y = 8 + 5 = 13$

$z = 4 + 5 = 9$

4. Explicit Access Control

Classes and structures introduce a new scope for their members. Similar to block structure, the scope of a member declaration x in a class C extends to any sub-class C' , except if C' has a local declaration of the same name x . Through the use of keywords like `public`, `private`, and `protected`, object oriented languages such as C++ or Java provide explicit control over access to member names in a super-class.

Private: Private members are accessible only by the methods of the same class or friend class

Public: Public names are accessible outside the class

Protected: Protected names are accessible to its sub-class.

5. Dynamic Scope

The term dynamic scope refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Two examples of dynamic policies: macro expansion in the C pre-processor and method resolution in object-oriented programming.

Consider, the following example in which identifier 'a' is a macro that stands for expression $(x + 1)$.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

In this example

- main() calls b() and c().
- b() calls a, a uses x which is defined in b(), a is evaluated with x value as 1.
- c() calls a, a uses x which is not defined in c(), main is the called procedure for c() which defines x as 2 and a is evaluated with x value as 2.
- The output for the code segment is

2

3

Difference between static scope and dynamic scope

Static scope	Dynamic scope
Related to space	Related to time
scope is determined from the nested (outer procedure)	scope is determined from the called procedure

6. Parameter Passing Mechanisms

Various parameter passing mechanisms are

- call-by-value
- call-by-reference
- call-by-name

The majority of languages use either "call-by-value," or "call-by-reference," or both.

a. Call-by-Value

In call-by-value, the actual parameter value is copied to formal parameters. This method is used in C, C++, Java, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Example: To swap two integers in C

```
void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

main( )
{
    int x = 5, y = 10;
    printf("Before swap: %d %d\n", x, y);
    swap(x,y);
    printf("After swap: %d %d\n", x, y);
}
```

Output is

Before swap: 5 10

After swap: 5 10

b. Call-by-Reference

In call-by-reference, the address of the actual parameter is passed to the corresponding formal parameter. Changes to the formal parameter thus appear as changes to the actual parameter. Call-by-reference is used for "ref" parameters in C++ and is an option in many other languages.

Example:

```
void swap(int *i, int *j)
```

```
{ int t;
```

```
    t = *i;
```

```
    *i = *j;
```

```
    *j = t;
```

```
}
```

```
main( )
```

```
{ int a = 5, b = 10;
```

```
    printf("Before swap: %d %d\n", a, b);
```

```
    swap(&a,&b);
```

```
    printf("After swap: %d %d\n", a, b);
```



```
}
```

Output

```
Before swap: 5 10
After swap: 10 5
```

7. Aliasing

It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Example

```
procedure p()
```

```
begin
```

```
    array a[1..10] of integer;
```

```
    q(a,a);
```

```
end
```

```
procedure q(x, y)
```

```
begin
```

```
    x [10] = 2;
```

end

Now, x and y have become aliases of each other. The value of y[10] also becomes 2.