Name, Binding and Scope

Dr. Nguyen Hua Phung
phung@cse.hcmut.edu.vn

HCMC University of Technology, Viet Nam

09, 2013

Name

name đại diện cho một địa chỉ nào đó

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

Thay vì dùng địa chỉ bộ nhớ: Trong lập trình, chúng ta không trực tiếp làm việc với địa chỉ bộ nhớ phức tạp (ví dụ: 0x7fff5fbff8d8). Thay vào đó, chúng ta dùng "tên" (ký hiệu) dễ nhớ và dễ sử dụng hơn.
Trừu tượng hóa (Abstraction): Việc sử dụng "tên" thay vì địa chỉ bộ nhớ là một hình thức trừu tượng hóa. Nó giúp chúng ta ẩn đi những chi tiết phức tạp bên dưới (địa chỉ bộ nhớ) và tập trung vào logic chương trình ở mức cao hơn.

Binding

Liên kết (Binding): Là quá trình kết nối một "tên" với một đối tượng hoặc một vị trí bộ nhớ cụ thể. Trong ví dụ int x = 5;, "liên kết" xảy ra khi tên x được gán với một vùng nhớ để lưu trữ giá trị 5. "Thời điểm liên kết" cho biết khi nào quá trình "liên kết" xảy ra. Thời điểm này có thể khác nhau tùy thuộc vào ngôn ngữ lập trình và cách chương trình được thực thi.

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.

Liên kết Tĩnh: Được cố định lúc biên dịch, không thay đổi khi chạy. Thường thấy ở C hoặc C ++.

Liên kết Động: Được quyết định lúc chạy, có thể thay đổi. Thường gặp ở các ngôn ngữ như Java, Python (ví dụ: ghi đè phương thức trong lập trình hướng đối tượng).

Khi biên dịch (Compile time): Đối với các ngôn ngữ biên dịch (như C, C++), một số liên kết được thực hiện ngay khi bạn biên dịch code thành file thực thi. (Thường gọi là liên kết tĩnh).

Khi chạy chương trình (Runtime): Đối với các ngôn ngữ thông dịch hoặc các tính năng như đa hình, một số liên kết chỉ được thực hiện khi chương trình đang chạy. (Thường gọi là liên kết động).

Một số vấn đề - Liên kết sớm (Early binding) so với Liên kết muộn (Late binding).

Liên kết sớm (Early binding - còn gọi là Static binding):
Xảy ra sớm: Quá trình liên kết diễn ra trước khi chương trình chạy, thường là ở thời điểm biên dịch.
Ít linh hoạt hơn: Các liên kết đã được xác định cố định từ trước.
Hiệu suất cao hơn: Do liên kết đã được thực hiện sớm, chương trình có thể chạy nhanh hơn một chút.

Liên kết muộn (Late binding - còn gọi là Dynamic binding):
Xảy ra muộn: Quá trình liên kết diễn ra trong khi chương trình đang chạy (thời điểm runtime).
Linh hoạt hơn: Liên kết có thể thay đổi tùy thuộc vào tình huống và dữ liệu trong quá trình chạy.
Kém hiệu suất hơn một chút: Cần thời gian để xác định liên kết khi chạy, có thể chậm hơn một chút so với liên kết sớm.

Binding Time

- Language design time 'int', 'float'
- Language implementation time int = 4 bytes...
- Programming time do dev, vd int x -> x là kiểu số nguyên, hoặc tên = x...
- Compilation time vd gọi hàm: Lệnh gọi ràng buộc với hàm được gọi...
- Linking time code trên nhiều file -> Link các file đó lại với nhau
- Load time thời gian nạp chương trình vào đĩa cứng để thực thi (địa chỉ tuyệt đối của biến trong bộ nhớ trong)
- Runtime những cái nào chạy mới xảy ra ràng buộc, ví dụ cin << a; -> đợi lúc nhập vào mới có giá trị

Thời điểm liên kết (Binding Time)
Liên kết có thể xảy ra ở các giai đoạn sau:

Thời điểm thiết kế ngôn ngữ: Khi ngôn ngữ được tạo ra (ví dụ: quy tắc của + được định nghĩa).
Thời điểm triển khai ngôn ngữ: Khi xây dựng trình biên dịch hoặc thông dịch.
Thời điểm lập trình: Khi bạn viết mã (ví dụ: khai báo biến).
Thời điểm biên dịch: Khi mã được dịch sang mã máy.
Thời điểm liên kết: Khi ghép các thư viện vào chương trình.
Thời điểm tải: Khi chương trình được nạp vào bộ nhớ.
Thời điểm chạy: Khi chương trình đang thực thi.

Object Lifetime

Object: Là bất kỳ thứ gì trong chương trình: biến, hàm, dữ liệu... Nói chung là thứ có thật mà chương trình dùng.

Object lifetime < Binding time

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

Object lifetime: Là khoảng thời gian từ lúc đối tượng được tạo (creation) đến lúc nó bị hủy (destruction).
Ví dụ: Mày khai báo biến, nó sống từ lúc khai báo đến khi chương trình không cần nữa.

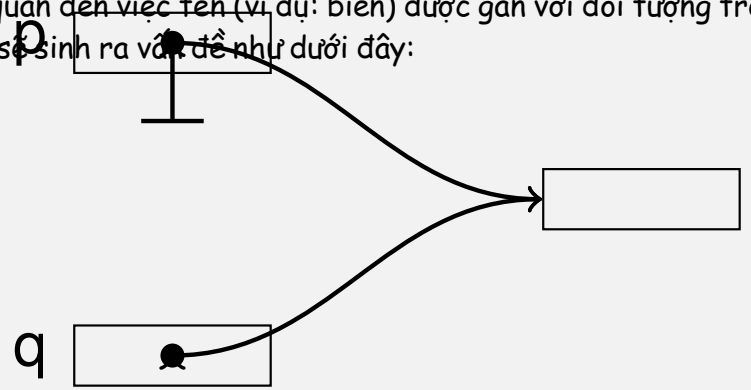
- *Binding lifetime*

Binding lifetime: Liên quan đến việc tên (ví dụ: biến) được gắn với đối tượng trong bao lâu. Nếu liên kết bị lỗi, sẽ sinh ra vấn đề như dưới đây:

- *Dangling reference*

```
p = new int;
q = p;
delete p;
*q;
```

Là khi con trỏ (pointer) trỏ tới một vùng nhớ đã bị xóa, nhưng chương trình vẫn cố dùng.



Object lifetime < Binding time
=> nguy hiểm hơn

- *Leak memory - Garbage*

Object lifetime > Binding time

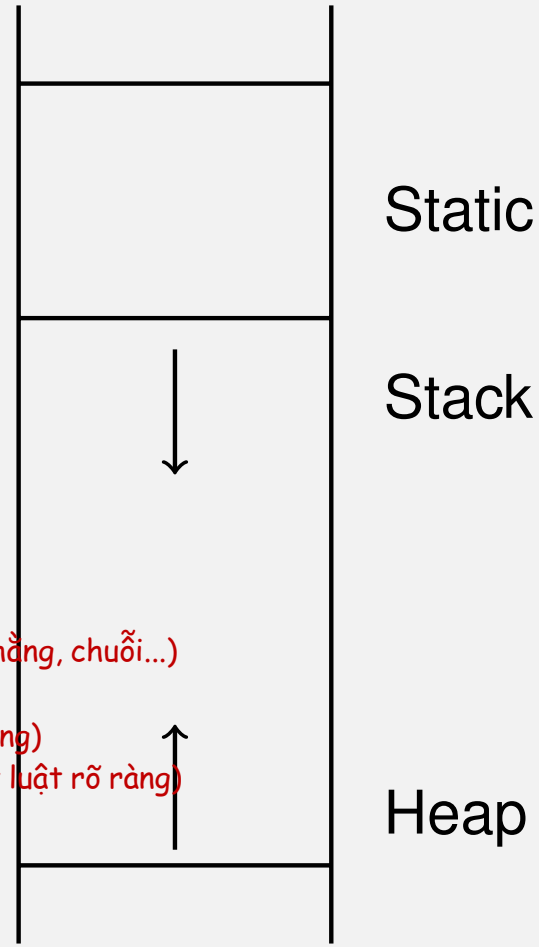
```
p = new int;
p = null;
```

khi cấp phát bộ nhớ mà quên xóa, dẫn đến mất kiểm soát vùng nhớ đó.

Object Allocation

Cấp phát đối tượng

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



Static: Object lifetime từ lúc chương trình chạy đến khi kết thúc (các biến toàn cục, hằng, chuỗi...)
Stack dynamic: Các biến cục bộ (gọi đệ quy các thứ, FILO - Tạo trước bị huỷ bỏ sau)
Explicit Heap Dynamic: Có lệnh cụ thể để tạo một đối tượng (không theo quy luật rõ ràng)
Implicit Heap Dynamic: Không có lệnh cụ thể, nhưng vẫn tạo đối tượng (không theo quy luật rõ ràng)

Static: Cố định từ đầu, sống suốt chương trình.
Stack Dynamic: Tạo/xóa tự động theo phạm vi, nằm trên stack.
Explicit Heap Dynamic: Mày tự cấp phát, tự xóa, nằm trên heap.
Implicit Heap Dynamic: Ngôn ngữ lo hết, mày chỉ cần xài, cũng trên heap.

1. Static (Tĩnh)
Nghĩa: Đối tượng được cấp phát bộ nhớ ngay từ lúc biên dịch (compile time), không thay đổi trong suốt chương trình.
Đặc điểm:
Kích thước và vị trí bộ nhớ cố định.
Tồn tại suốt vòng đời chương trình.
Ví dụ: Biến toàn cục trong C như `int x = 5;` – bộ nhớ được cấp sẵn khi chương trình chạy.
Ưu điểm: Nhanh, không cần quản lý thủ công.
Nhược điểm: Không linh hoạt, không đổi kích thước được.
2. Stack Dynamic (Động trên ngăn xếp)
Nghĩa: Đối tượng được cấp phát bộ nhớ trên stack khi chương trình chạy, cụ thể là khi vào một khối mã (block), và tự động bị xóa khi ra khỏi khối đó.
Đặc điểm:
Bộ nhớ được cấp lúc runtime, nhưng vẫn cố định kích thước.
Vòng đời gắn với phạm vi (scope), như một hàm.
Ví dụ: Biến cục bộ trong C như int x; trong hàm – tạo khi hàm chạy, mất khi hàm kết thúc.
Ưu điểm: Tự động quản lý (không cần xóa thủ công), nhanh.
Nhược điểm: Chỉ sống trong phạm vi nhất định, không dùng lâu dài được.
3. Explicit Heap Dynamic (Động rõ ràng trên heap)
Nghĩa: Đối tượng được cấp phát bộ nhớ trên heap (một vùng nhớ lớn) bằng cách lập trình viên tự yêu cầu, và cũng phải tự xóa.
Đặc điểm:
Cấp phát lúc runtime qua lệnh như `new` hoặc `malloc`.
Vòng đời do lập trình viên quyết định (xóa bằng delete hoặc free).
Ví dụ: Trong C++:
```
int* p = new int;  // Cấp phát trên heap
delete p;          // Tự xóa
```
Ưu điểm: Linh hoạt, dùng được lâu dài.
Nhược điểm: Dễ gây rò rỉ bộ nhớ hoặc tham chiếu lơ lửng nếu quên xóa hoặc xóa sai.
4. Implicit Heap Dynamic (Động ngầm trên heap)
Nghĩa: Đối tượng được cấp phát trên heap, nhưng ngôn ngữ tự động quản lý việc cấp phát và giải phóng (thường có garbage collector).
Đặc điểm:
Kích thước và vòng đời thay đổi tự do lúc runtime.
Lập trình viên không cần can thiệp thủ công.
Ví dụ: Trong Java hoặc Python:
```
ArrayList list = new ArrayList();  // Tự động cấp phát, JVM lo dọn rác
```
Ưu điểm: Dễ dùng, tránh lỗi quản lý bộ nhớ thủ công.
Nhược điểm: Chậm hơn vì có thêm cơ chế dọn rác (garbage collection).

```
int x;

void foo(int y) {
   static int z;
   int * t = malloc(sizeof(int));
   ...
}
```

Các lựa chọn
x is allocated in static memory: Đúng (biến toàn cục, nằm ở static memory).
y is allocated in stack memory: Đúng (tham số hàm, nằm trên stack).
z is allocated in static memory: Đúng (có static, nằm ở static memory, không phải stack).
t is allocated in heap memory: **Sai** (bản thân t nằm trên stack, **vùng nhớ nó trỏ tới mới ở heap**).

vì bản thân biến t nằm trên stack, không phải heap. Tuy nhiên, vùng nhớ mà t trỏ tới (do malloc) thì nằm trên heap, nên có thể gây nhầm lẫn.

t: Là con trỏ int*, được khai báo trong hàm, nên bản thân t là biến cục bộ, được cấp phát trên stack. Tuy nhiên, t = malloc(sizeof(int)) cấp phát một vùng nhớ trên heap cho giá trị mà t trỏ tới. Do đó, câu 4 "t is allocated in heap memory" cần được xem xét:

Given the following C fragment:

```
int x;

void foo(int y) {

   static int z;

   int * t = malloc(sizeof(int));

   ...

}
```

Choose the WRONG statement?
The lifetime of x is the same as the lifetime of the whole program
The lifetime of y is the same as the lifetime of function foo
The lifetime of z is the same as the lifetime of function foo
The lifetime of t is the same as the lifetime of function foo

Biến static dù có nằm trong hàm hay không, hàm có được gọi hay không thì nó vẫn có vòng đời bằng với program.
Biến static được tạo ra ngay từ khi chương trình bắt đầu chạy (start of program) và tồn tại cho đến khi chương trình kết thúc (end of program).
Điều này có nghĩa là nó sống suốt vòng đời của chương trình, bất kể hàm chứa nó có được gọi hay không.
Nó không bị xóa đi khi hàm kết thúc, không giống biến cục bộ bình thường (local variable).
Nằm ở static memory (bộ nhớ tĩnh), không phải stack, nên không bị ảnh hưởng bởi việc hàm vào/ra scope.

```
void foo() {
   static int z = 0;  // Khởi tạo lần đầu
   z++;
   printf("z = %d\n", z);
}
int main() {
   foo();  // In: z = 1
   foo();  // In: z = 2
   foo();  // In: z = 3
```

```
// position 1
int * foo() {
   // position 2
   x[0] = 1;
   return x;
}
```

Câu hỏi: Khai báo x ở đâu và kiểu nào gây lỗi runtime?

Phân tích từng lựa chọn

1. int x[10]; // position 1

Vị trí: Ngoài hàm foo, ở position 1 (global scope).

Loại: Biến mảng toàn cục, cấp phát ở static memory.

Xử lý:

x[0] = 1: Gán giá trị vào mảng, không vấn đề gì.

return x: Trả về con trỏ tới mảng x. Vì x là biến toàn cục, nó sống suốt chương trình, nên con trỏ trả về vẫn hợp lệ.

Kết quả: Không gây lỗi runtime (không dangling reference, không garbage).

2. int x[10]; // position 2

Vị trí: Trong hàm foo, ở position 2 (local scope).

Loại: Biến mảng cục bộ, cấp phát trên stack memory.

Xử lý:

x[0] = 1: Gán giá trị vào mảng, ok trong hàm.

return x: Trả về con trỏ tới x. Nhưng x là biến cục bộ, khi hàm foo kết thúc, vùng nhớ trên stack của x bị giải phóng. Con trỏ trả về giờ trỏ vào vùng nhớ không còn hợp lệ nữa -> dangling reference.

Kết quả: Gây lỗi runtime (dangling reference).

3. static int x[10]; // position 2

Vị trí: Trong hàm foo, ở position 2.

Loại: Biến mảng static, cấp phát ở static memory.

Xử lý:

x[0] = 1: Gán giá trị, không vấn đề.

return x: Trả về con trỏ tới x. Vì x là static, nó sống suốt chương trình, nên con trỏ trả về vẫn trỏ tới vùng nhớ hợp lệ.

Kết quả: Không gây lỗi runtime (vòng đời đủ dài, không dangling hay garbage).

4. int* x = malloc(10*sizeof(int)); // position 2

Vị trí: Trong hàm foo, ở position 2.

Loại: Con trỏ trỏ tới vùng nhớ cấp phát trên heap bằng malloc.

Xử lý:

x[0] = 1: Gán giá trị vào vùng nhớ heap, ok.

return x: Trả về con trỏ x. Vì vùng nhớ được cấp phát trên heap, nó vẫn tồn tại sau khi hàm kết thúc (cho đến khi gọi free). Con trỏ trả về hợp lệ.

Kết quả: Không gây lỗi runtime trong trường hợp này (nhưng nếu quên free sau này, có thể gây memory leak, nhưng câu hỏi không hỏi tới).

Đáp án

Câu gây lỗi runtime: int x[10]; // position 2.

Lý do:

x là biến cục bộ trên stack, khi hàm foo kết thúc, vùng nhớ của x bị giải phóng.

Con trỏ trả về (return x) giờ trỏ vào vùng nhớ đã bị thu hồi -> dangling reference.

Ngoài hàm, nếu cố truy cập (ví dụ int* ptr = foo(); printf("%d", ptr[0]);), chương trình có thể crash hoặc cho kết quả sai.

Kiểm tra các lựa chọn khác

1: x toàn cục -> sống lâu, không lỗi.

3: x static -> sống suốt chương trình, không lỗi.

4: x trên heap -> sống cho đến khi free, không lỗi trong ngữ cảnh này.

Chỉ 2 gây lỗi runtime vì vòng đời của x ngắn hơn con trỏ trả về.

Blocks

Một block là một vùng văn bản trong chương trình, nơi mày có thể khai báo các biến hoặc thực thể chỉ dùng được trong vùng đó.
Nghĩa là: Trong block, mày định nghĩa biến, và biến đó chỉ "sống" trong phạm vi của block đó thôi.

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;
}
```

Scope

Scope của một liên kết (binding) là vùng văn bản trong chương trình mà liên kết đó (ví dụ: giữa tên biến và giá trị) có hiệu lực.
Nói đơn giản: Scope quyết định chỗ nào trong code mày có thể dùng biến đó.

Definition

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

```
int x = 10;
void foo() {
  int x = 5;
  printf("%d", x);  // In 5, vì x trong block foo được dùng
}
```

Static vs. Dynamic

- Static scope, or lexical scope, is determined during compilation

  Định nghĩa: Được xác định lúc biên dịch (compile time), dựa trên cấu trúc code.

  - Current binding - in the block most closely surround
  - Global scope
  - Local static scope

  Khi mày dùng một biến, trình biên dịch nhìn vào block gần nhất bao quanh biến đó để tìm liên kết.
  Nếu không tìm thấy trong block hiện tại, nó sẽ tìm lên các block cha (nếu có), gọi là global scope nếu là toàn cục.

- Dynamic scope is determined at runtime.

  - Current binding - the most recently execution but not destroyed

    Được xác định lúc chạy (runtime), dựa trên thứ tự thực thi của chương trình.

Khi dùng biến, chương trình tìm liên kết gần nhất trong các hàm đã gọi trước đó, nhưng chưa bị hủy (dựa trên stack thực thi).
Nó không quan tâm cấu trúc code, mà nhìn vào lịch sử gọi hàm.
Ưu điểm: Linh hoạt trong một số trường hợp.
Nhược điểm: Khó đoán, dễ rối khi debug.

```
int x = 10;
void bar() {
  printf("%d", x);  // Dynamic scope sẽ in giá trị x từ hàm gọi nó
}
void foo() {
  int x = 5;
  bar();  // Nếu dùng dynamic scope, in 5 (x của foo)
}
```

Static Scope:

Quyết định lúc biên dịch.

Dựa trên block bao quanh (local -> global).

Ví dụ: C, C++, Java dùng static scope.

Dynamic Scope:

Quyết định lúc chạy.

Dựa trên hàm gọi gần nhất (most recent).

Ví dụ: Một số ngôn ngữ cũ như Lisp (phiên bản đầu) dùng dynamic scope.

Staic Scope Rules for Blocks

TẦM VỰC TĨNH (STATIC SCOPE)

- A reference to an identifier is always bound to **its most local declaration** Dùng local của nó (gần nhất)
- A declaration is **invisible** outside the block in which it appears Khai báo block trong -> block ngoài không thấy khai báo đó
- 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.

Trong một số ngôn ngữ, block có thể được đặt tên, và tên của block đó được xem như một khai báo local trong block bên ngoài chứa nó.
Nghĩa đơn giản: Nếu block có tên, tên đó chỉ dùng được trong block cha, giống như biến local của cha.

Thường block khai báo bên ngoài có thể được bên trong thấy (trừ khi bên trong overwrite thì nó sẽ dùng cái overwrite đó thay vì dùng của thằng cha)

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.
```

A bị khai báo lại

Đoạn này A có hiệu lực (main)

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

Example on Dynamic Scope

```
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

Referencing Environment

Referencing Environment (Môi trường tham chiếu)
Định nghĩa: Là tập hợp tất cả các tên (biến, hàm, v.v.) mà một câu lệnh trong chương trình có thể "thấy" và sử dụng được.
Nghĩa đơn giản: Khi mày viết một dòng code, môi trường tham chiếu là danh sách những thứ mày được phép gọi tên tại chỗ đó.

Nghĩa là những thằng mà thằng đó có thể sử dụng được.

- 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

So sánh Static vs Dynamic

Static-Scoped:
Dựa trên cấu trúc code (block lồng nhau).
Xác định lúc biên dịch.
Ví dụ: x trong block con che x ngoài nếu khai báo lại.
không quan tâm block.

Dynamic-Scoped:
Dựa trên thứ tự gọi hàm (stack thực thi).
Xác định lúc chạy.
Ví dụ: x lấy từ hàm gọi gần nhất,

Trong ngôn ngữ Static-Scoped (Phạm vi tĩnh)
Giải thích: Trong ngôn ngữ dùng phạm vi tĩnh (như C, Java), môi trường tham chiếu bao gồm:
Các tên khai báo trong block hiện tại (local names).
Tất cả các tên "thấy được" từ các block bên ngoài bao quanh (enclosing scopes).
Cách hoạt động:
Trình biên dịch nhìn cấu trúc code lúc biên dịch để xác định tên nào dùng được.
Nếu tên không có trong block hiện tại, nó tìm lên block cha, rồi cha của cha, cho đến ngoài cùng (global).
Ví dụ:

```
int x = 10; // Global scope
void foo() {
    int y = 5; // Local scope
    printf("%d %d", x, y); // Thấy cả x (ngoài) và y (trong)
}
```

=> Môi trường tham chiếu trong foo: x (từ global) + y (từ local).

Trong ngôn ngữ Dynamic-Scoped (Phạm vi động)
Giải thích: Trong ngôn ngữ dùng phạm vi động (như một số phiên bản cũ của Lisp), môi trường tham chiếu bao gồm:
Các liên kết (bindings) trong block hiện tại (local bindings).
Tất cả các liên kết "thấy được" từ các subprogram (hàm) đang hoạt động (active), tức là các hàm đã gọi trước đó nhưng chưa kết thúc.
Cách hoạt động:
Lúc chạy (runtime), chương trình nhìn vào stack thực thi để tìm tên.
Nó lấy liên kết gần nhất từ hàm nào gọi hàm hiện tại, không quan tâm cấu trúc code.
Ví dụ:

```
int x = 10;
void bar() {
    printf("%d", x); // Dynamic scope: x phụ thuộc vào hàm gọi bar
}
void foo() {
    int x = 5;
    bar(); // In 5, vì x từ foo gần nhất trong stack
}
```

=> Môi trường tham chiếu trong bar: x từ foo (vì foo gọi bar), không phải x global.

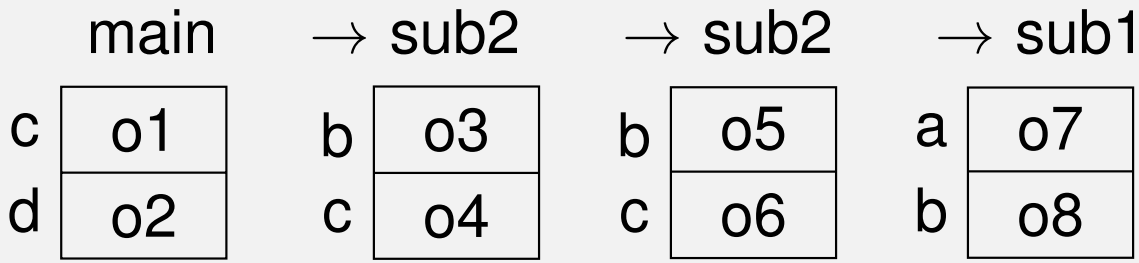
Example on Static-scoped Language

```
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

```
void sub1() {
        int a, 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

Summary [1]

- Name
- Binding
- Scope
- Referencing Environment

References I


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