

Names and Bindings

แทนที่ตัวเลข address

Name is Symbolic identifier used to refer to variable, constant, operation, type etc., instead of referring to low-level concepts like address or fragment of code

ความสัมพันธ์ ชื่อ กับ สิ่งที่ map ไป

Binding is An association between a name and the thing it names



Binding time is Time at which an association is created (or time at which any implementation decision is made)

- **Static binding** = Things are bound before run time (early binding) รู้ล่วงหน้า ถึงตอน compile
 - Associated with greater efficiency, e.g. **compiler decides on layout of variables in memory and generates efficient code to access them.** optimize ได้ตั้งแต่ตอน compile แต่ไม่ flexible
 - Compiled languages tend to have early binding times.
- **Dynamic binding** = Things are bound at run time (late binding) คิดตอน run ทำให้ละบรรทัด ไม่ optimize
 - Associated with greater flexibility, e.g. decision on which data value of **which type is bound to a variable**
name may be made at run time
 - Interpreted languages tend to have later binding times. ซ้ำ

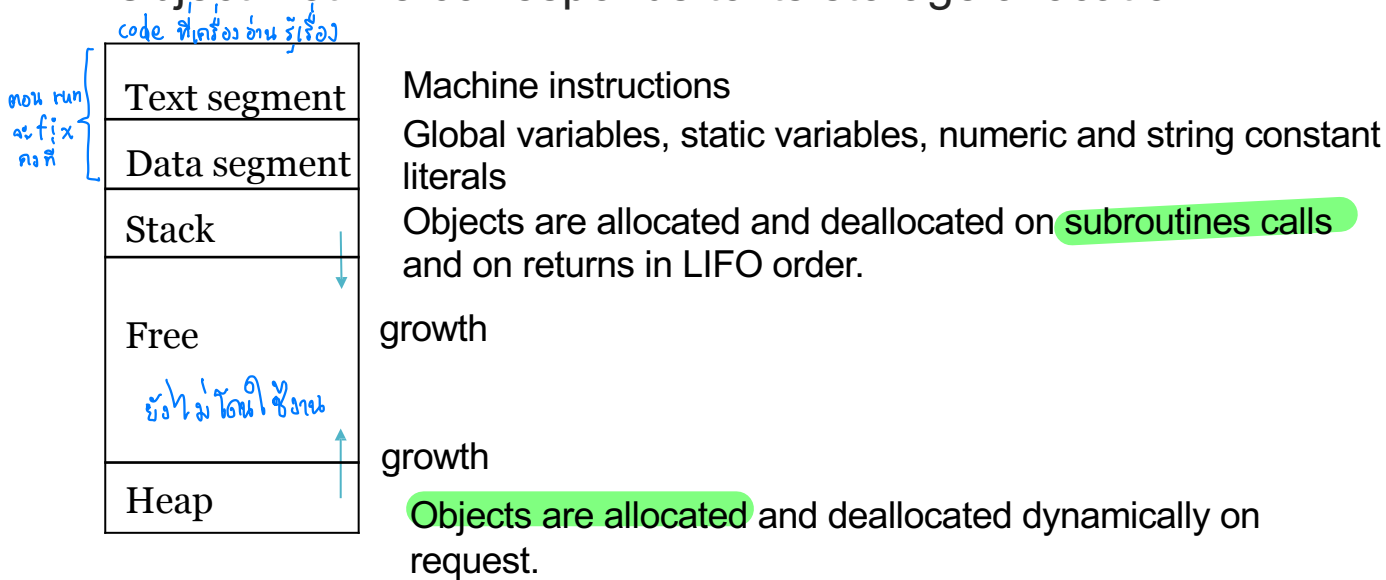
We will talk about binding of the identifiers to the variables they name.

Object Lifetime and Storage Management

Lifetime is time between creation and destruction.

An object to which a name is bound has its object lifetime.

Object lifetime corresponds to its storage allocation.



Fixed size. Static objects are given an absolute address and retained throughout program execution.

Stack-Based Allocation

Each instance of a called subroutine at run time has its own **frame** (or **activation record**) on stack which contains

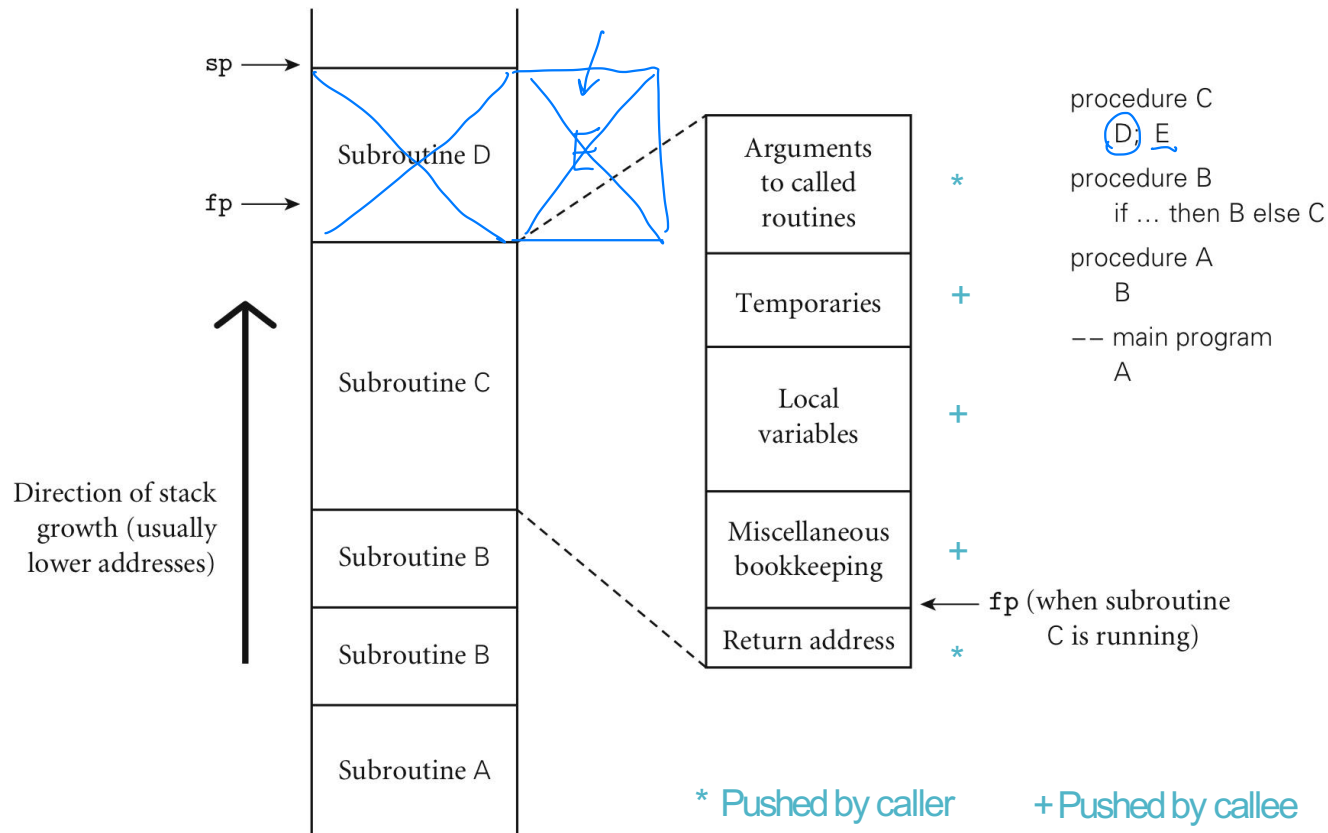
- Arguments (if any)
- Return address *ค่าที่อยู่หลังจากที่ function เสร็จแล้ว จะกลับไป run ตรงไหน*
- Bookkeeping : Saved values of registers, reference to other frame (see later) etc.
- Local variables (if any)
- Temporaries : Intermediate values produced in complex calculations

Stack pointer (sp) is a register for the address of the top of stack. *เก็บ address บนสุดของ stack*

Compiler cannot tell the location of a frame but the offsets of objects in a frame.

- Frame pointer (fp) is a register that points to a known (reference) location within a frame of the current subroutine. *จำจุดอ้างอิงใน frame ที่จุดนั้น*
- Code to access data within a frame adds a predetermined negative or positive offset to the value in fp.

Calling Chain and Stack Frames



Maintenance of Stack

Compiler generates the following:

ເຮັດຍົກ

Caller	
...	
Pre-call	Push arguments onto the stack.
jsr callee	Call subroutine. This also pushes the <u>return address</u> onto the stack.
Post-call	Deallocate the stack space it allocated in the pre-call (adding positive offset to sp), and continue at the address of the instruction immediately after the jsr.
...	

ຄູ່າເຮັດຍົກ

Callee	
Prologue	Push old fp value and update it to a new frame, push other register values that should not be changed, and push local variables (if any) onto the stack.
Main Body	Execute and store return value in a register (or in a location in the stack, if reserved by caller).
Epilogue	Restore the saved register values, and deallocate the stack space it allocated in the prologue (adding positive offset to sp).
rts ↓	Return to caller, jumping back to the return address.

Exercise: Write sequence of stack allocation for calling add3()

`sum = 0`

```
int add2 (int a, int b) {  
    return a + b;  
}  
  
int add3 (int a, int b, int c) {  
    int res;  
    res = add2(a, b);  
    return res + c;  
}  
  
global var int sum = 0;  
int main() {  
    sum += add3 (1, 2, 3);  
    return 0;  
}
```

main
return value=0

```
graph TD
    main["main  
return value=0"]
    add3["int add3 (int a, int b, int c) {  
    int res;  
    res = add2(a, b);  
    return res + c;  
}"]
    add2["int add2 (int a, int b) {  
    return a + b;  
}"]
    main --> add3
    add3 --> add2
    add2 --> add3
```


Heap-Based Allocation

เขียน code เพื่อสั่งจองพื้นที่ Data โดย manu

Heap is storage in which subblocks can be allocated and deallocated at arbitrary times.

Heaps are required for dynamically allocated pieces of linked data structures or objects whose size may change (e.g. string, list, set).

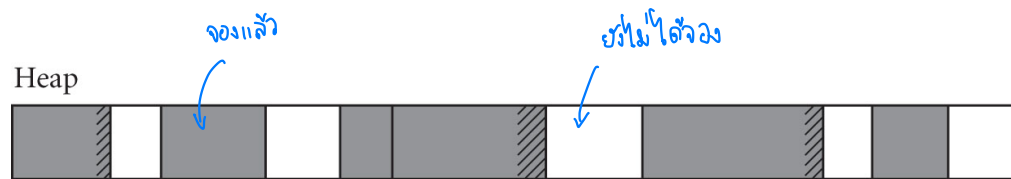
Allocation is by some operation in a program, e.g. `malloc()` in C, `new` in Java, `C++`.

เอาข้อมูล heap หมดไม่ได้ใช้แล้ว

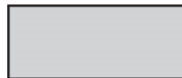
Deallocation is by garbage collection or some operation in a program, e.g. `free()` in C, `delete` in C++

Heap Storage Management

Storage management algorithms allocate blocks of the required size.



Allocation request



เหลือมากพอ แต่จองไม่ได้ เพราะพื้นที่กระจัดกระจายกัน

- **External fragmentation** happens when free spaces are scattered and not large enough to satisfy requests.
 - *Compaction is needed to move already allocated blocks.* ต้องเอามาชิดกัน
- **Internal fragmentation** happens when unneeded space is left in the allocated block as it is smaller than some minimum threshold. จองเกินแล้ว ไม่ได้ใช้

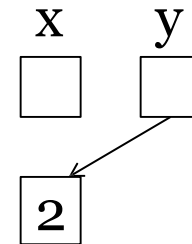
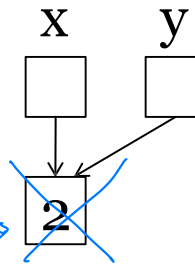
Some storage management algorithms maintain pools of different standard block sizes.

- Each request is rounded up to the next standard size, at the cost of internal fragmentation.

Problems with Manual Deallocation

Dangling reference is when the program accesses memory of the already released object (which may now be used by another object).

```
int *x = new int;  
int *y;  
*x = 2;  
y = x;  
delete x;
```

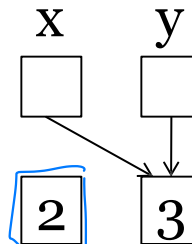
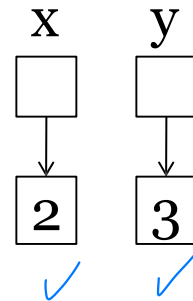


Try to change
pointer that does
not usable

segmentation
fault

Memory leak is when an object is not deallocated at the end of lifetime and heap space may run out.

```
int *x = new int;  
int *y = new int;  
*x = 2;  
*y = 3;  
x = y;
```



This is fixed
by garbage
collection

โดนของตลอดไป
ไม่พอประโยชน์

ต้องใช้ garbage collector

Exercise: Heap-Based Objects and Binding

Since lifetime means the time between creation and destruction,

Binding lifetime is

Object lifetime is

If object lifetime is longer than binding lifetime, we have.....

If binding lifetime is longer than object lifetime, we have

Garbage Collection

Garbage occurs when objects are no longer reachable from any program variables. Instead of explicit deallocation of heap-based objects, some languages provide run time library with garbage collection mechanism to implicitly identify and reclaim unreachable objects.

- More recent imperative languages, e.g. Java, C# *new ไม่ต้อง free*
- Most functional and scripting languages

Benefit

- Reducing programming errors

Drawback


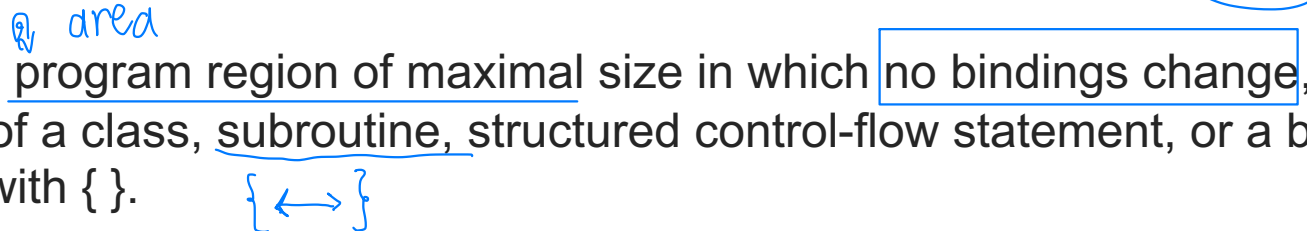
- Execution speed and complexity in language implementation *เข้าถึง memory 100%*

With language implementation becoming more complex, marginal complexity of garbage collection is reduced.

It is now an essential language feature with improved algorithms.

Scopes



- Scope of binding is a textual region of the program in which a name-to-object binding is active. 
- Scope is a program region of maximal size in which no bindings change, e.g. the body of a class, subroutine, structured control-flow statement, or a block delimited with { }. 
- **Referencing environment** is a set of active bindings at any given point in a program's execution.

Exercise: Scope and Referencing Environment

```
// c  
  
float op1(int x, float y) { int z; ... }  
  
float op2(int z) { ... }
```

op1 is considered one scope and op2 another scope.

Scope of x and y is ^{op 1}

Scope of z is ^{op 1 / op 2}

Referencing environment of op1 consists of ^{op 1, x, y, z₁, op2}

Referencing environment of op2 consists of ^{op 2, z₂, op1}

Static Scoping (or Lexical Scoping)

scope ကာကွယ် source code မှာရှိသော run code

Bindings between names and objects can be determined at compile time by examining the text of the program, without consideration of the flow of control at run time.

In most modern languages, scope of a binding is determined statically at compile time.

We can look at a C program, for example, and know which names refer to which objects at which points in the program based on purely textual rules. Then C is statically scoped.

Generally, scope of a local variable is limited to the subroutine in which it appears.

Classic Example: Nested Scope in Nested Subroutines

A feature in many languages (e.g. Pascal, Ada, Python) is nested subroutines. In nested subroutines, bindings of names to objects are resolved by closest nested scope rule.

- A name is known (or visible) in the scope in which it is declared, and in each internally nested scope.
- But a name can be hidden by another declaration of the same name in one or more nested scopes.
- To find the object that is bound to a given use of name
 - *We look for its declaration in the current innermost scope.*
 - *If there is one, it defines the active binding for the name.*
 - *Otherwise we continue outward, examining successively surrounding scopes.*

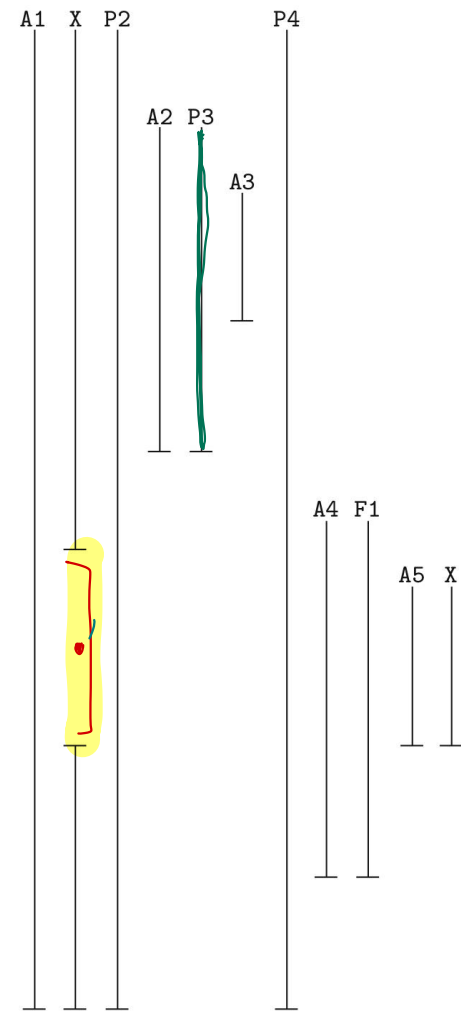
Nested Subroutines

Pascal

```

procedure P1(A1 : T1);
var X : real;
...
procedure P2(A2 : T2);
...
  procedure P3(A3 : T3);
  ...
  begin
    X = A3; (* body of P3 *)
  end;
  ...
  begin
    P3(3); (* body of P2 *)
  end;
  ...
procedure P4(A4 : T4);
...
  function F1(A5 : T5) : T6;
  var X : integer;
  ...
  begin
    ... (* body of F1 *)
  end;
  ...
  begin
    P2(2); (* body of P4 *)
  end;
  ...
begin
  P4(4); (* body of P1 *)
end

```



Access to Non-local Objects in Nested Subroutines

To find objects in lexically surrounding scopes, static link is maintained in each frame when the frame is active at run time.

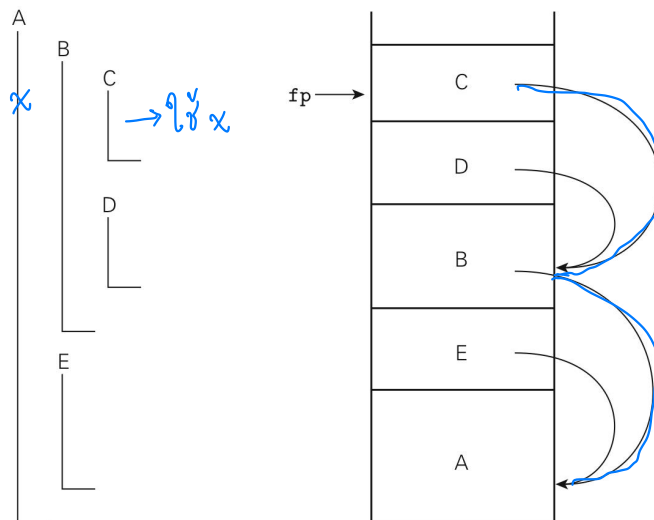
Static link points to parent frame, i.e. the frame of the most recent invocation of the lexically surrounding subroutines.

- Static link is the value of the fp of the parent frame, computed and passed in a register by caller, and stored as part of bookkeeping information.

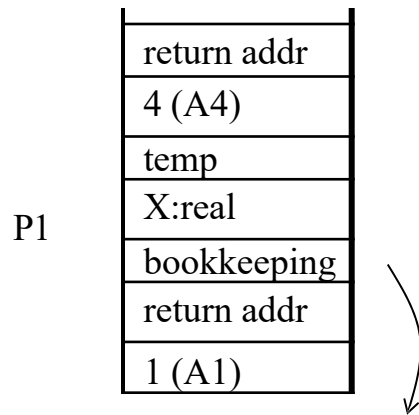
Sequence of nested calls at run time is A, E, B, D, C

C can find local objects in surrounding scope B by dereferencing **static chain** once and adding offset.

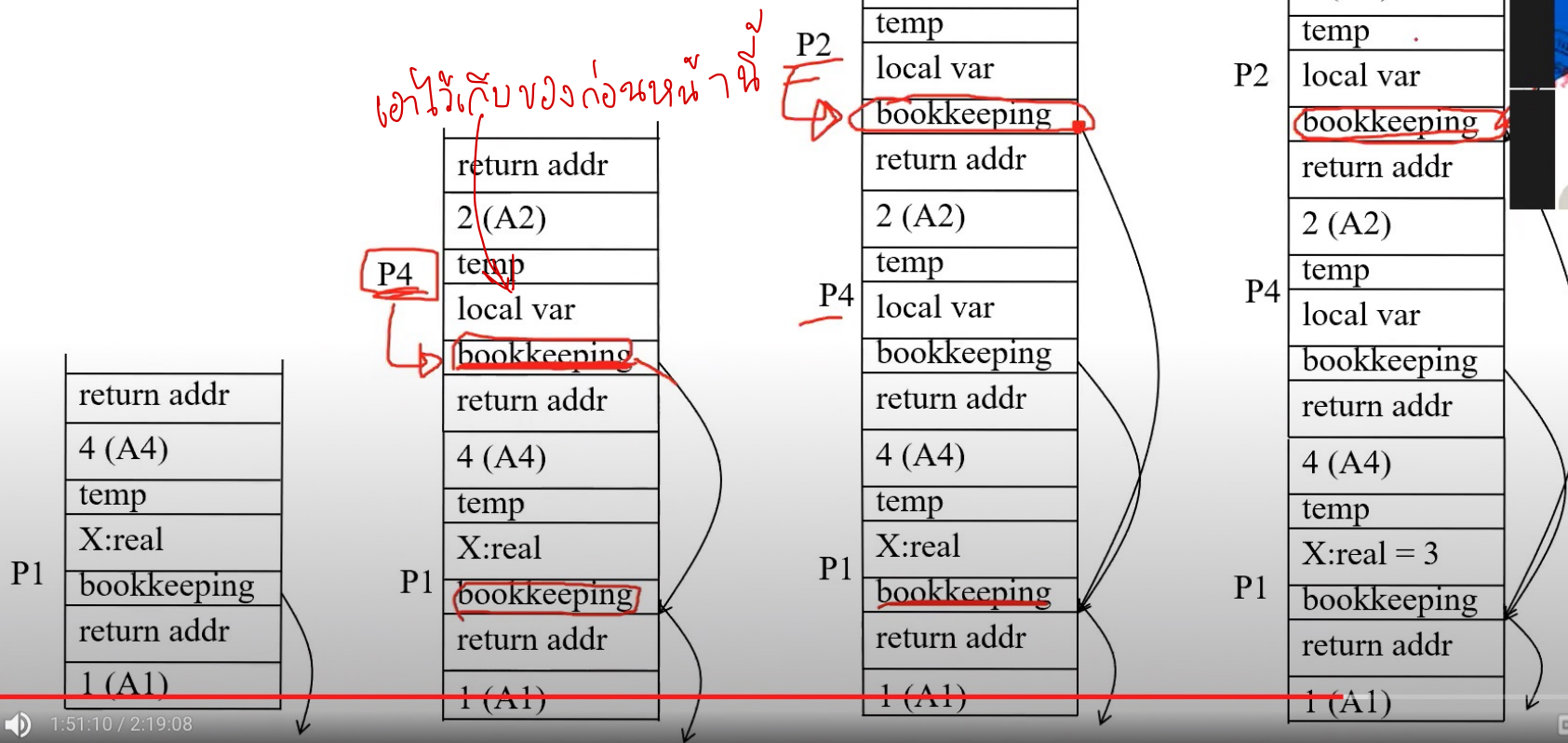
C can find local objects in B's surrounding scope, A, by dereferencing **static chain** twice and adding offset.



Exercise: Static Link in Nested Subroutines



Exercise: Static Link in Nested Subroutines



```

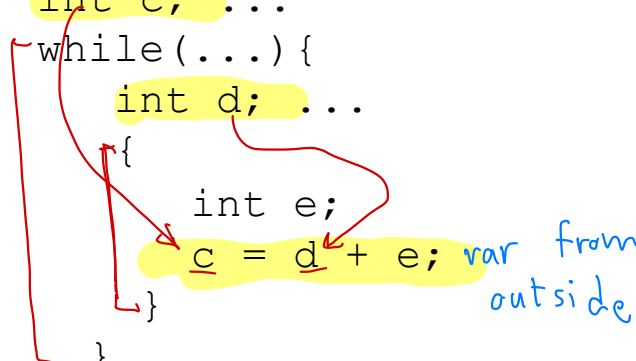
procedure P1(A1 : T1);
var X : real;
    procedure P2(A2 : T2);
        procedure P3(A3 : T3);
            begin
                X := A3;
            end;
        begin
            P3(3);
        end;
    procedure P4(A4 : T4);
        begin
            P2(2);
        end;
    begin
        P4(4);
    end;
end;
    
```

Nested Blocks

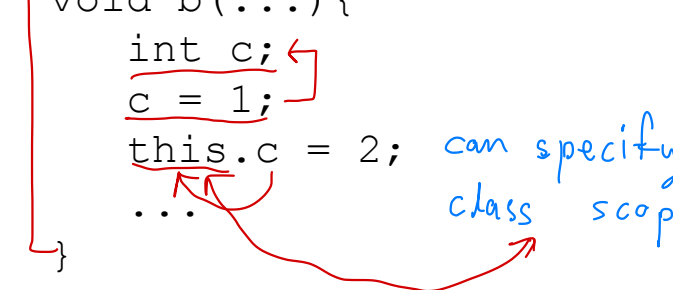
In some languages (e.g. C, C++, Java), declarations in nested {...} blocks hide outer declarations with the same name.

The space is allocated with local variables in subroutine prologue and deallocated in the epilogue.

```
// Java
class a {
    void b(...){
        int c; ...
        while(...){
            int d; ...
            {
                int e;
                c = d + e; var from outside
            }
        }
    }
}
```



```
// Java
class a {
    int c;
    void b(...){
        int c;
        c = 1;
        this.c = 2; can specify class scope
        ...
    }
}
```



Dynamic Scoping

หัดสลับใจตอน run

Bindings between names and objects depend on the flow of control, and on the order in which subroutines are called.

The current binding for a given name is the one encountered most recently during execution, and not yet destroyed by returning from its scope.

Type checking in expressions and argument checking in subroutine calls must be deferred until runtime.

Languages with dynamic scoping tend to be interpreted, rather than compiled, e.g. Lisp, Perl.

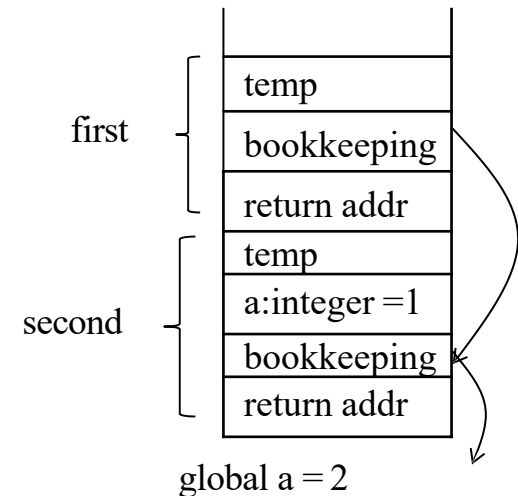
Bindings in Dynamic Scoping

With static scope rule, global *a* is reassigned to 1.

With dynamic scope rule, and if *first* is entered from *second*, local *a* is assigned to 1 .
The write refers to global *a* = 2.

```
1:  a : integer      -- global declaration
2:  procedure first
3:    a := 1
4:  procedure second
5:    a : integer      -- local declaration
6:    first ()
7:  a := 2
8:  if read_integer () > 0
9:    second ()
10: else
11:   first ()
12: write_integer (a)
```

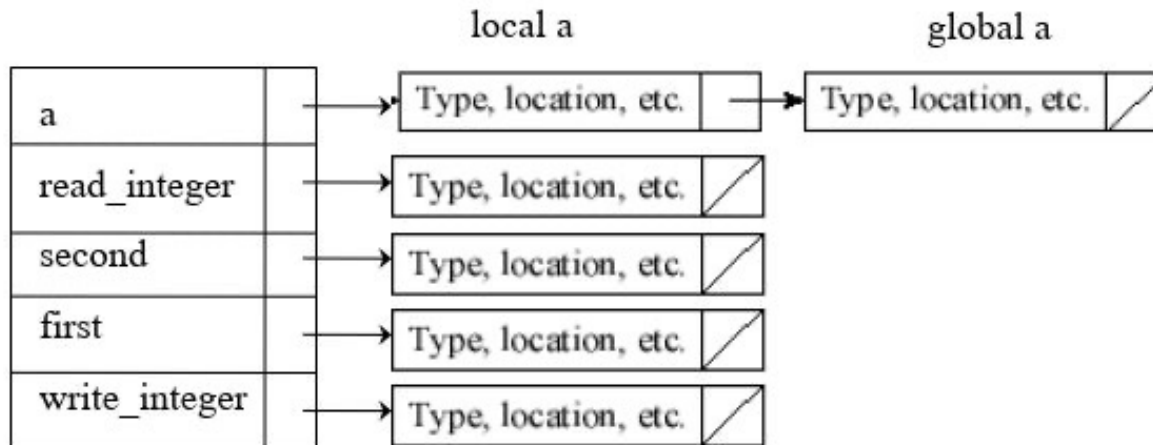
(Handwritten blue arrow from 'a' in line 12 to the word 'global')



Looking up for Bindings in Dynamic Scoping

Instead of traversing run time stack, a central reference table can be additionally implemented.

It contains a list of entries for each distinct name in the program, with the most recent occurrence of each at the beginning of the list (LIFO ordering of bindings).



Exercise: Bindings in Dynamic Scoping

With dynamic scope rule, if first is entered from main

- What does write_integer refer to, global a or local a?
- What does write_integer write?

```
1:  a : integer      -- global declaration
2:  procedure first
3:    a := 1
4:  procedure second
5:    a : integer      -- local declaration
6:    first ()
7:  a := 2
8:  if read_integer () > 0
9:    second ()
10: else
11:   first ()
12: write_integer (a)
```

