Memory Management

Programming Language Theory

Topics

- Static Management
- Dynamic Management w/ Stack
- Dynamic Management w/ Heap
- Scope Rule Implementation

Overview and Static Management

Memory Management

- Memory management is one of the key functions of an interpreter.
- While a program is running, various information is produced and loaded from or stored to memory.
- e.g.) values of local variables, temporary values of expressions, arguments and return values of functions, and many others.
- Hence it is necessary to decide how to deal with such memory access of a programming language.

Terminology

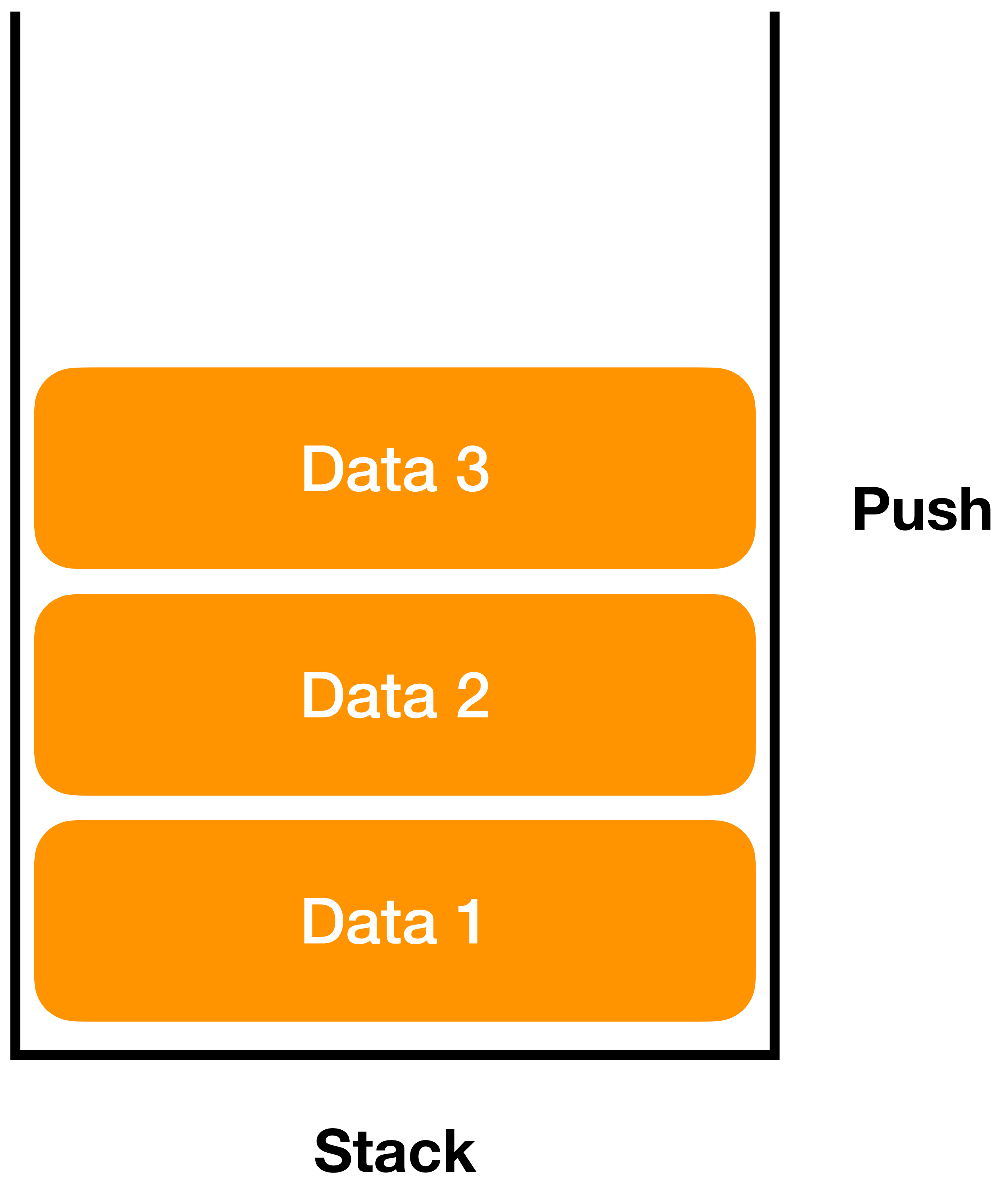
- We will use procedure, function, routine or subroutine as synonyms.
- They are all used to represent the concept of subprogram.
- In some PL, they might have different meaning (e.g. return value), but we will consider them as the same thing.
- Mostly, we will use ***procedure/function*** interchangeably in this lecture.

Stack

- Stack is a data structure which literally **stacks** data.
- Consider the boxes as data.
- When you stacked the boxes, the **blue box** was placed before the **purple box**.
- You can't simply pull out the **blue box**, until you pick the **purple box** first.

Stack

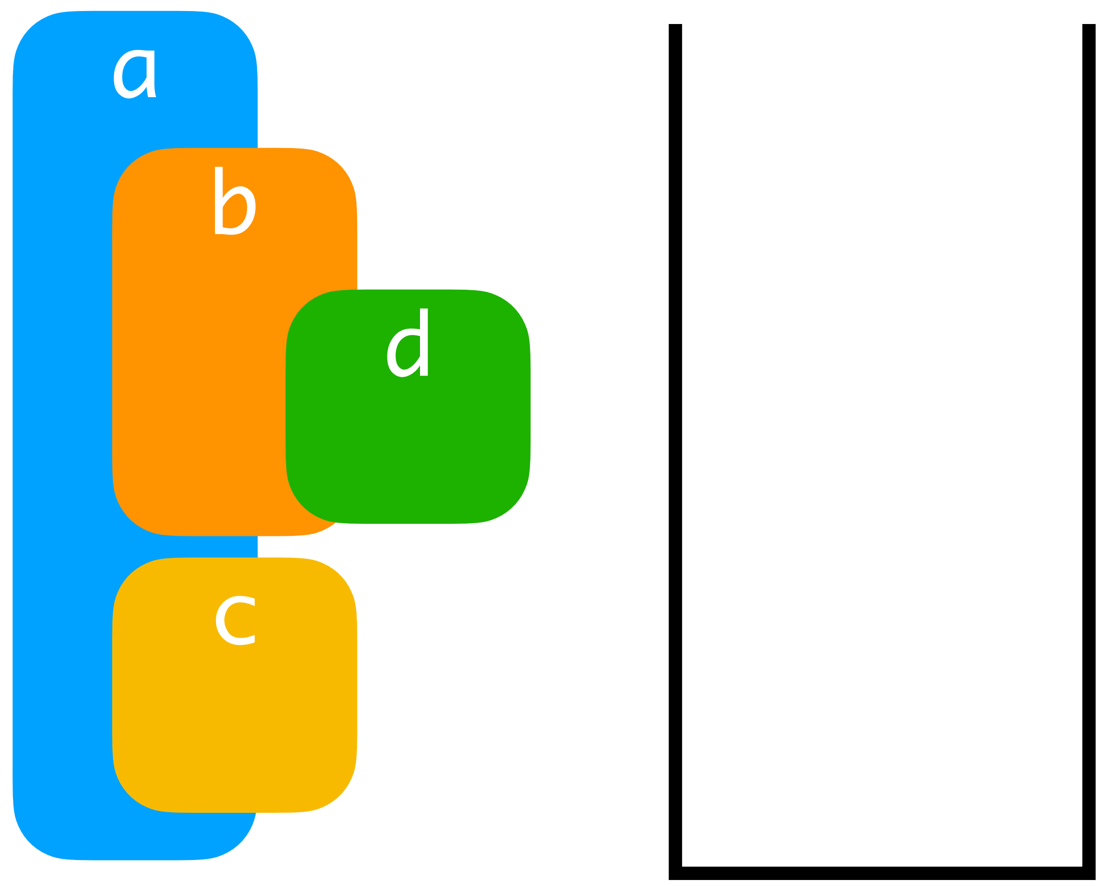
- Stack works similarly.
- The data inserted first, will be the one pulled out the last.
- It is called **LIFO** - **L**ast **I**n, **F**irst **O**ut.
- We can **push** the data to stack, and **pop** the last pushed data from the stack.



Stack and Procedure

- Stack naturally fits to procedures, since they are also activated in **LIFO**.
- The last procedure or block entered, is the first one which is exited.
- Environment changes can be also handled in this way.

```
a(){
  b(){
    d(){
    }
    d();
  }
  c(){
  }
  b();
  c();
}
```



Heap

- How about Heap?
- Heap is also a data structure related to priority queue or heap sort.
- However, in PL, ***heap is just a certain place in memory*** which can be allocated to a program.
- So there is no clear connection to the data structure heap in this case.

Static Management

- Static memory management is performed by the compiler, before program execution.

- Statically allocated objects are located in a fixed zone of memory.

- These objects stay in there for the entire program execution.

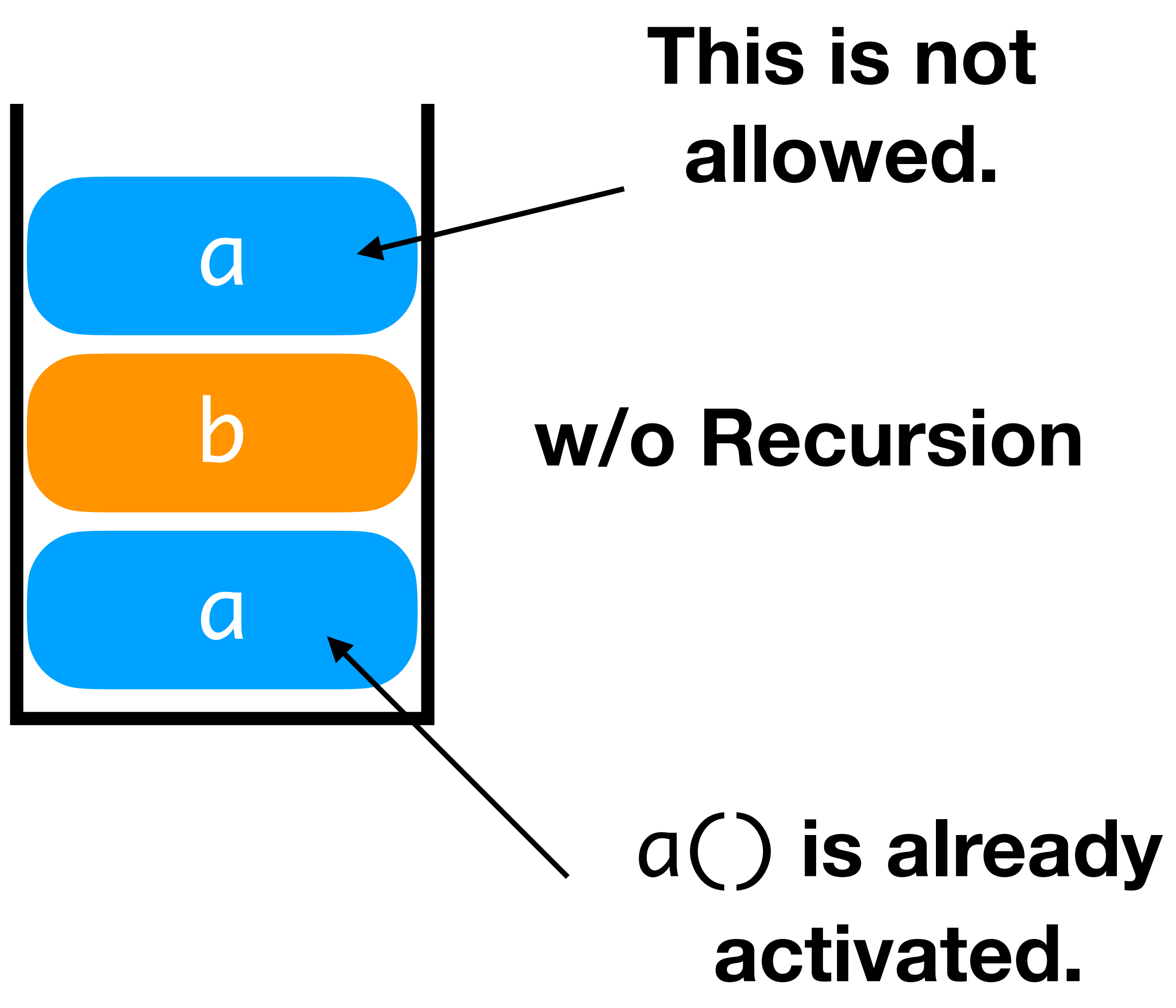
Which can be allocated statically?

- ***Global Variables***: they are available for entire program.
- ***Object Code***: machine instructions generated by the compiler.
- ***Constants***: only if their values can be decided during compile time.
- ***Compiler-generated Tables***: they are used for runtime support of a program.

Without Recursion

- Without recursion, more than one procedures cannot be activated at the same time.
- Hence it's possible to handle other components of PL statically.
    - e.g.) local variable, arguments, temporary values, return values and return address.
    - Because the same local variables can only be appeared in the stack once.

```
a(){
  b();
  ~~a();~~
}
```



Dynamic Management w/ Stack

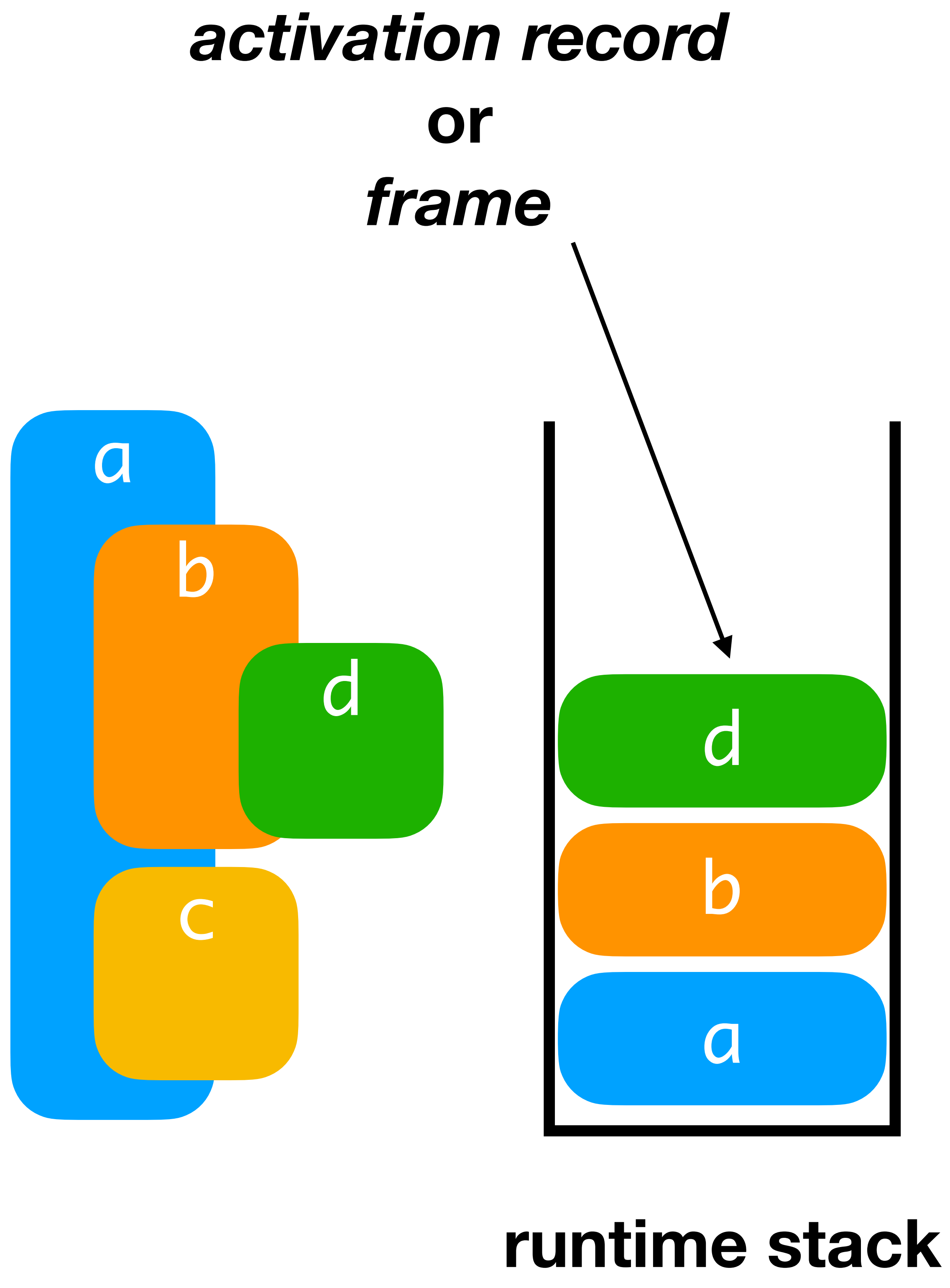
Dynamic Management

- Static management is not sufficient,
  - since not all program components can be determined before runtime.
- We can use stack and heap for dynamic memory management.

Dynamic Management w/ Stack

- We have already seen the basic concepts using stack.
- Each memory space allocated to a procedure activation (or an inline block) is called ***activation record*** or ***frame***.
- The stack containing activation records is called ***runtime stack***.

```
a(){
  b(){
    d(){
    }
    d();
  }
  c(){
  }
  b();
  c();
}
```



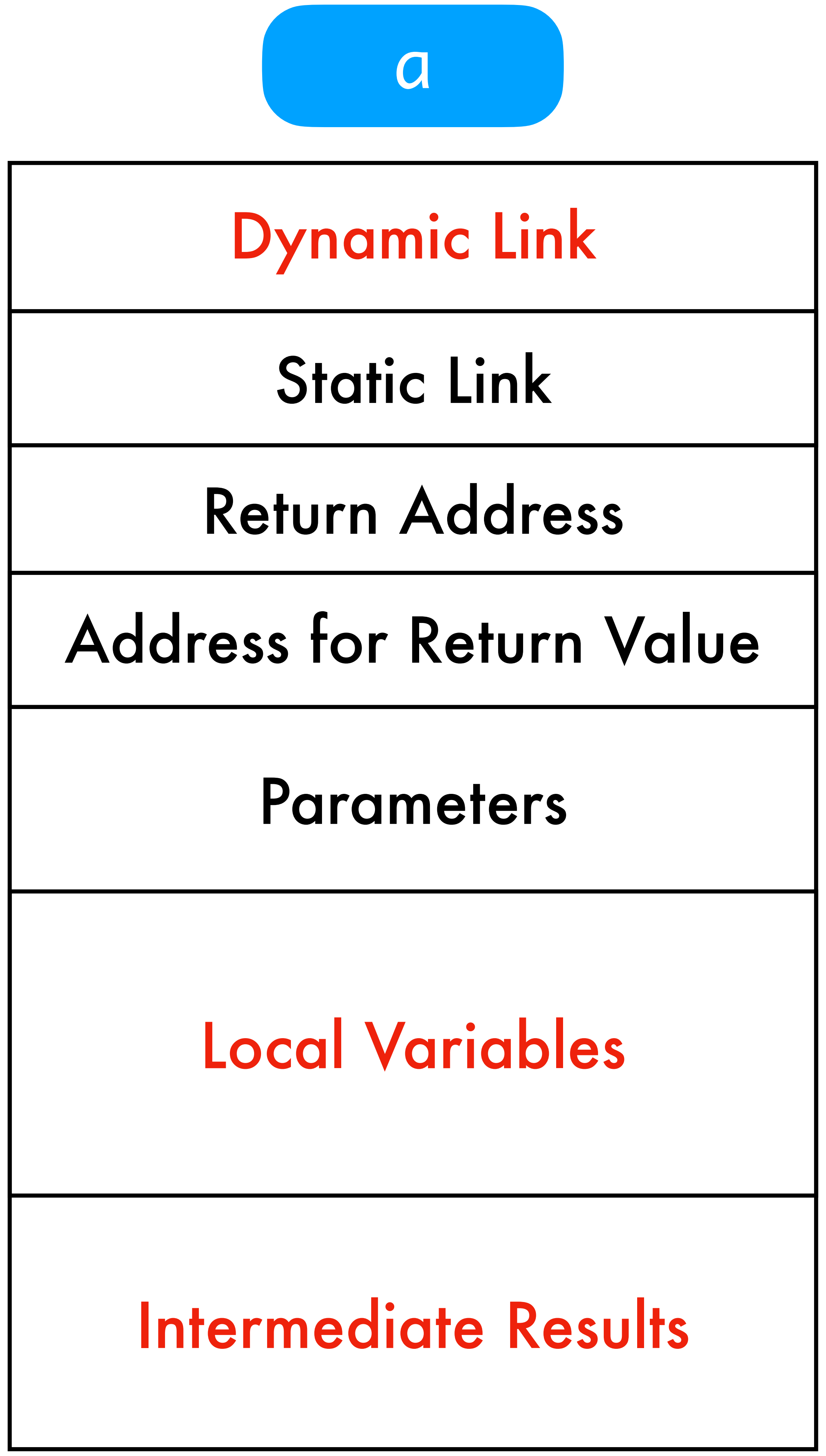
What's in Activation Record?

- Activation records for in-line blocks are much simpler than those of procedures.

- We will consider procedures only.

  - Information in for in-line blocks' activation records is a subset of those in procedures' activation records.

  - Red color means that it's common for in-line blocks and procedures.

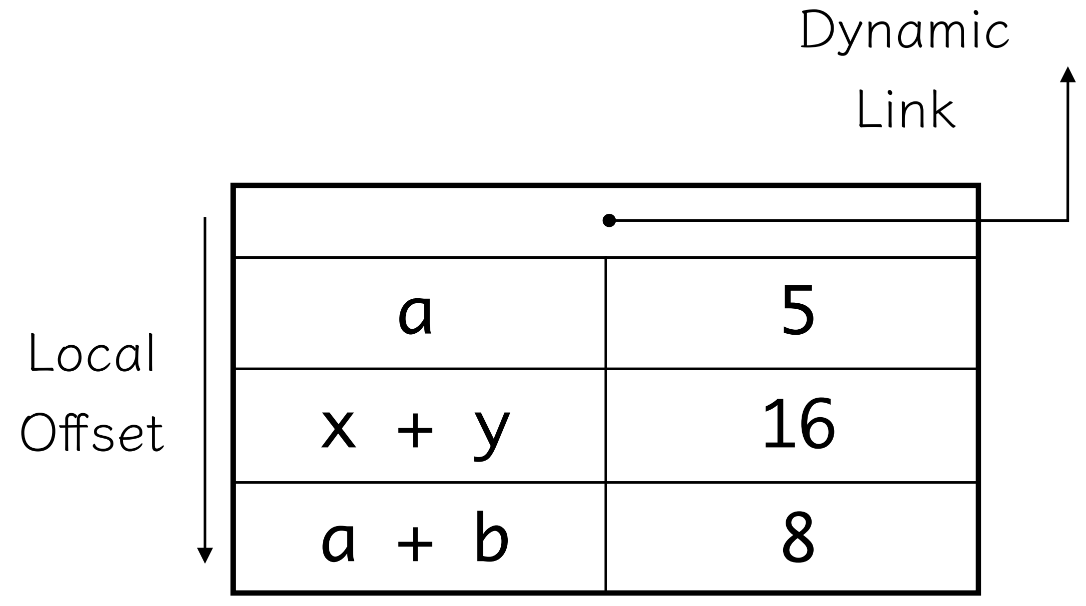


Dynamic Link

- An activation record with a local variable and intermediate results.
- **Dynamic Link** points to the start of *previous activation record* on the stack.
- This link is necessary since activation records have different sizes in general.
- From the start of activation record, we can use local offset to find a specific local variable.

In-line block

```
{
  int a = 5;
  b = (x+y) / (a+b);
}
```



Activation Record

Others

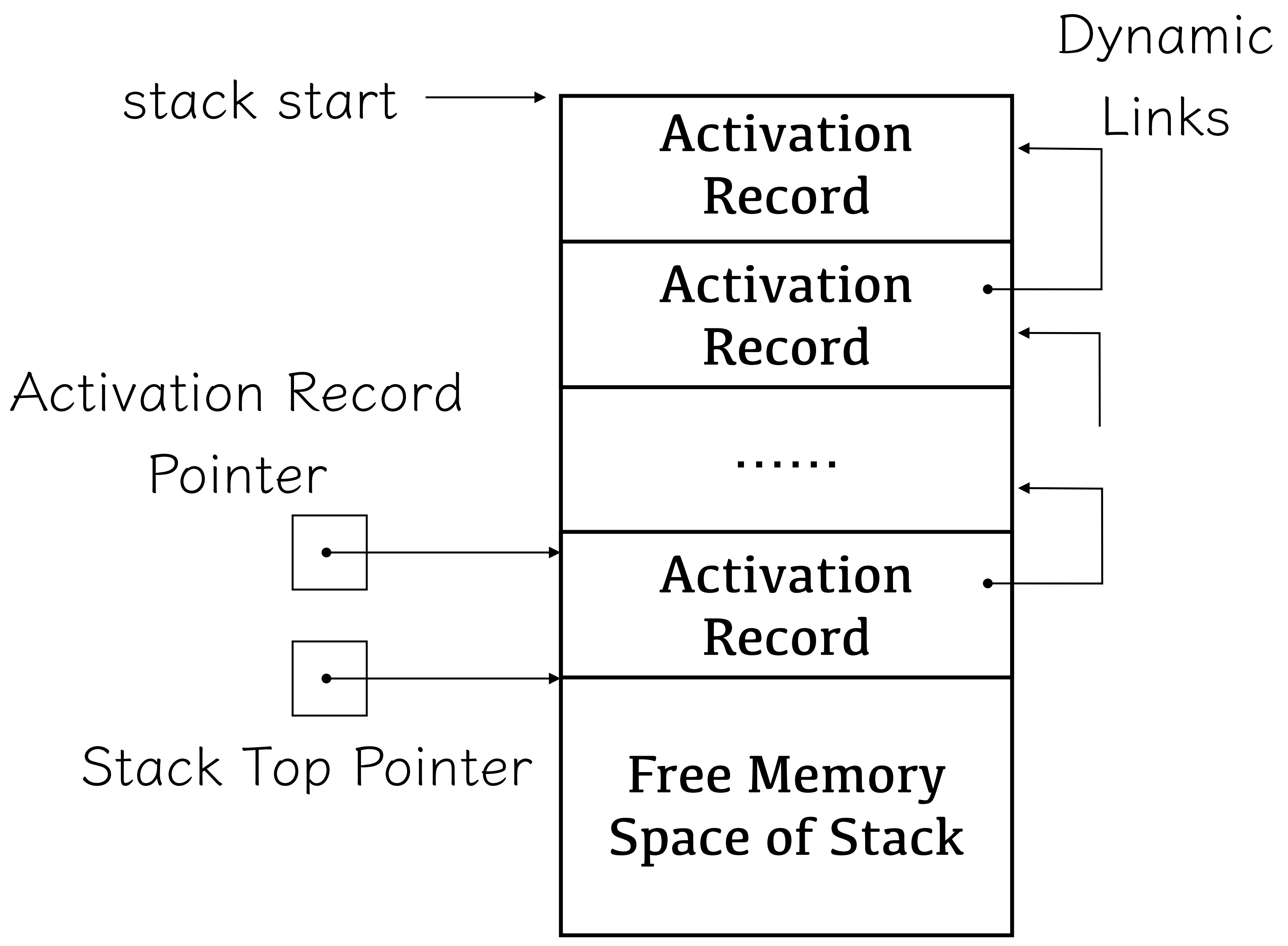
- **Return Address**: the next instruction after procedure call is finished.
- **Address for Return Value**: Return value will be stored in caller’s frame.
- **Parameters**: passed from caller.

Stack Management

- Activation records are stored or removed from the stack at runtime.
- When ***procedure B is called by procedure A***, both ***A (caller)*** and ***B (callee)*** manage such operations on the stack.

Stack Management

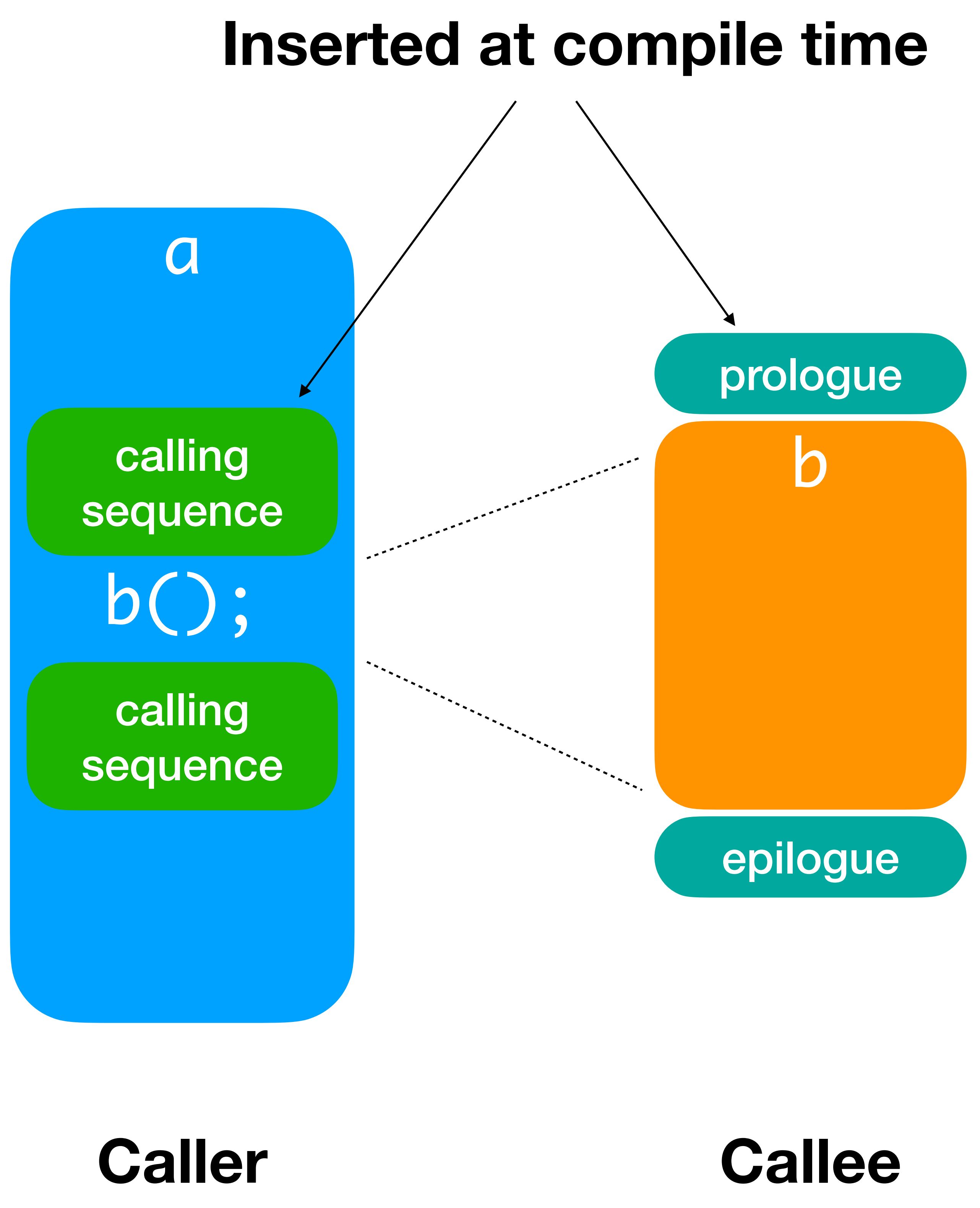
- Stack of Activation Records
- ***Activation Record Pointer***: Also called ***frame pointer*** or ***current environment pointer***.
- **Stack Top Pointer**: it shows where is the start of "free space".



Procedure Call

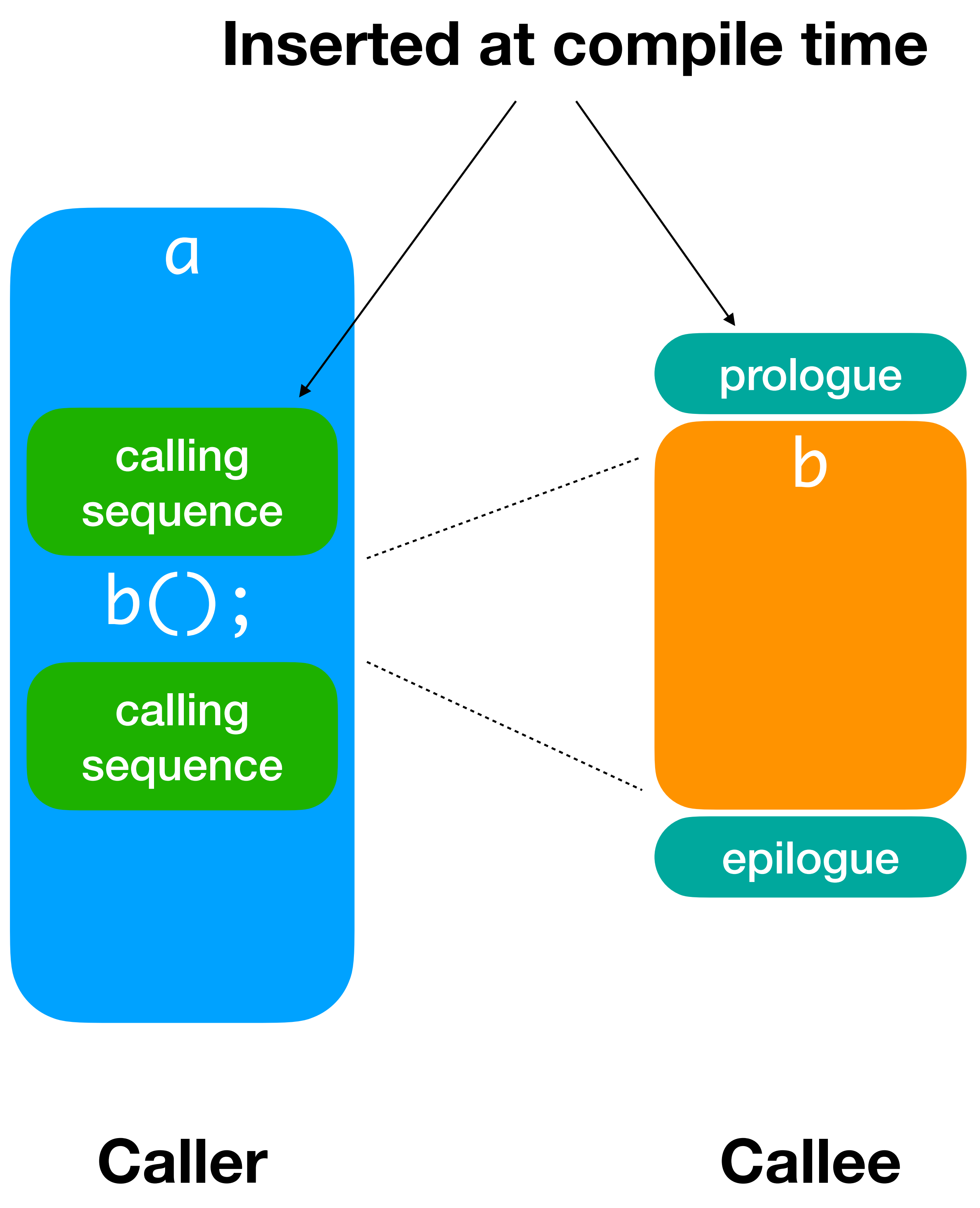
- ***Calling Sequence*** is inserted immediately before and after the procedure call by the compiler.

- ***Prologue*** and ***Epilogue*** are also inserted before and after procedure (callee) execution.



Procedure Call

- The exact operations of three code fragments depend on the compiler and implementation.
- To optimize the size of code, a large part of such code is given to callee.
    - Since it is inserted only once at declaration.
    - Otherwise, we need to add many times for each call.



Tasks before Procedure Call

- The following tasks will be done by ***calling sequence*** *and* ***prologue***.

- Program Counter Modification
- Stack Space Allocation
- Activation Record Pointer Modification
- Parameter Passing
- Register Save
- Initialize Code Execution

Tasks after Procedure Call

- The following tasks will be done by ***epilogue*** *and* ***calling sequence***.

- Update Program Counter
- Value Return
- Return of Registers
- Finalize Code Execution
- Stack Space Deallocation

Dynamic Management w/ Heap

Heap

- Why we need ***Heap*** for memory management?
    - We already have stack, and it seems natural to manage memory for procedures.
- Some languages have statements which allow ***explicit memory allocation***.

Explicit Memory Allocation

- With explicit memory allocation, there is no guarantee of LIFO.
- In the example, pointer variable p is the first one allocated, and also the first one deallocated.
- If we use the stack, we can't deallocate p before q, since q is at the top.

```
int *p, *q;
p = malloc(sizeof(int));
q = malloc(sizeof(int));
*p = 1;
*q = 2;
free(p);
free(q);
```

Allocation

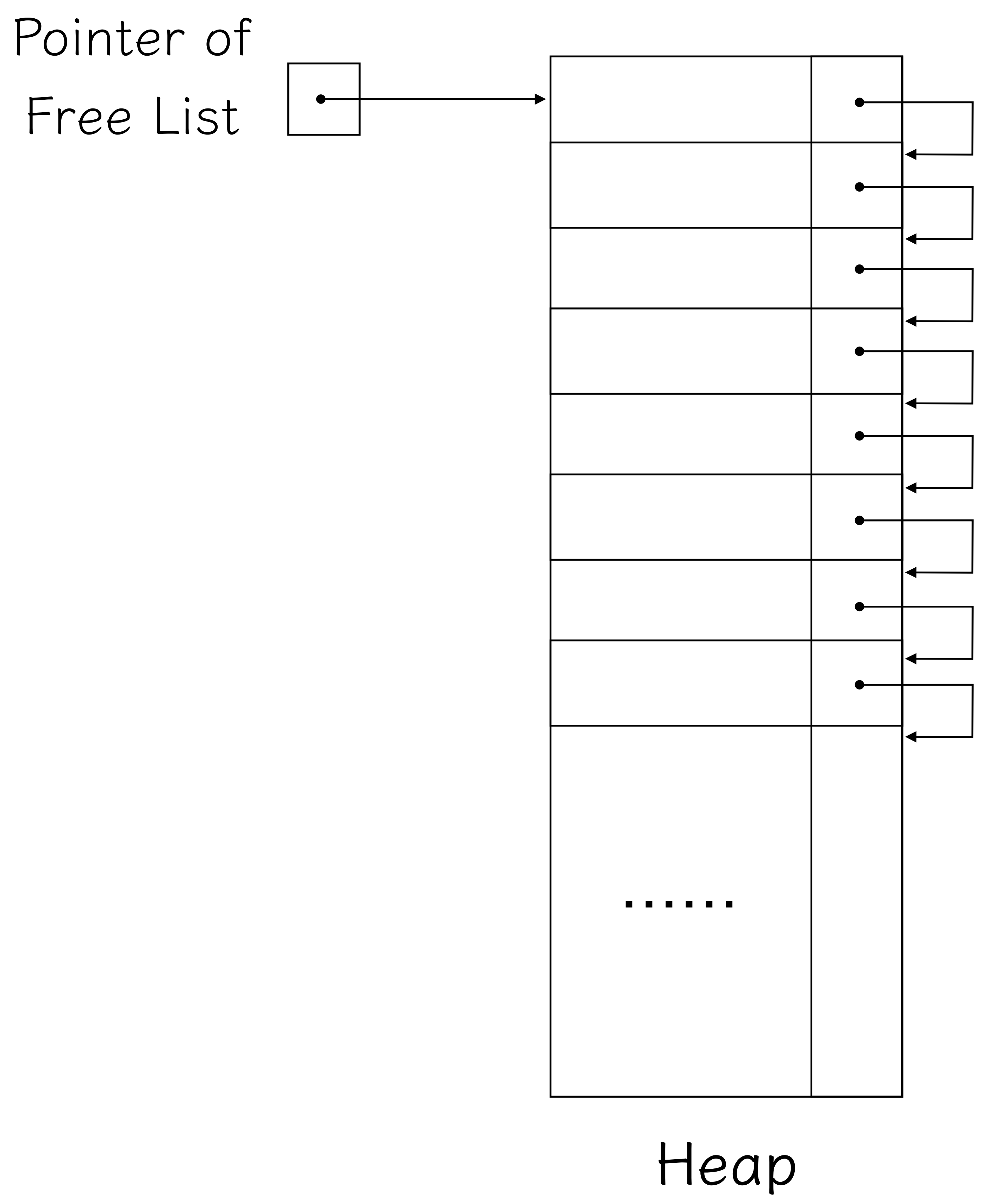
Deallocation

Heap Management

- Heap management methods fall into two main categories,
- based on how the memory blocks are considered,
  - *Fixed Length Blocks*
  - *Variable Length Blocks*

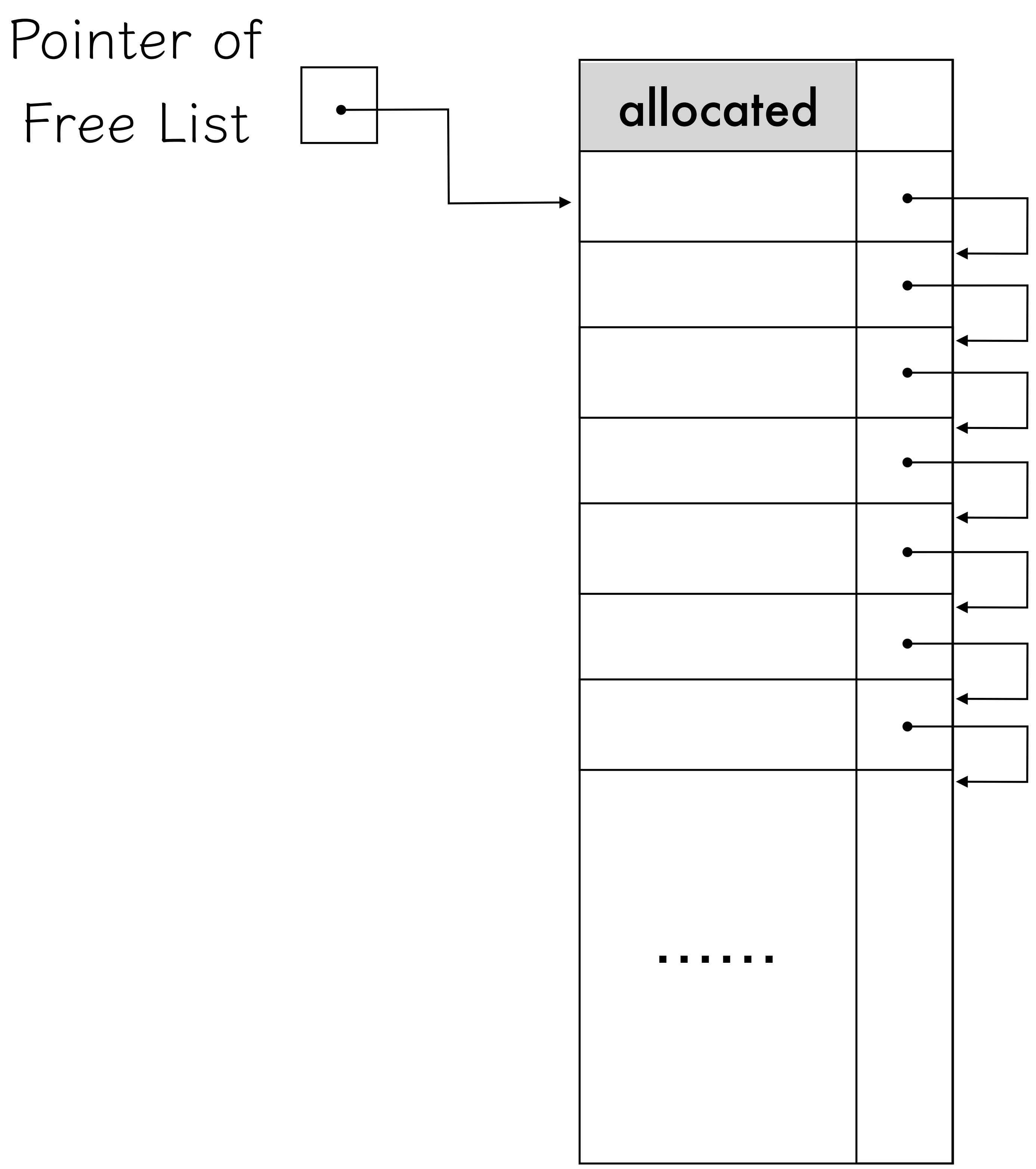
Fixed Length Blocks

- Divide the heap to multiple fixed length blocks.
- Using a free list to maintain the list of free blocks.
- For each request, the first free block will be allocated.
- The pointer of the free list points to the first block on the list.



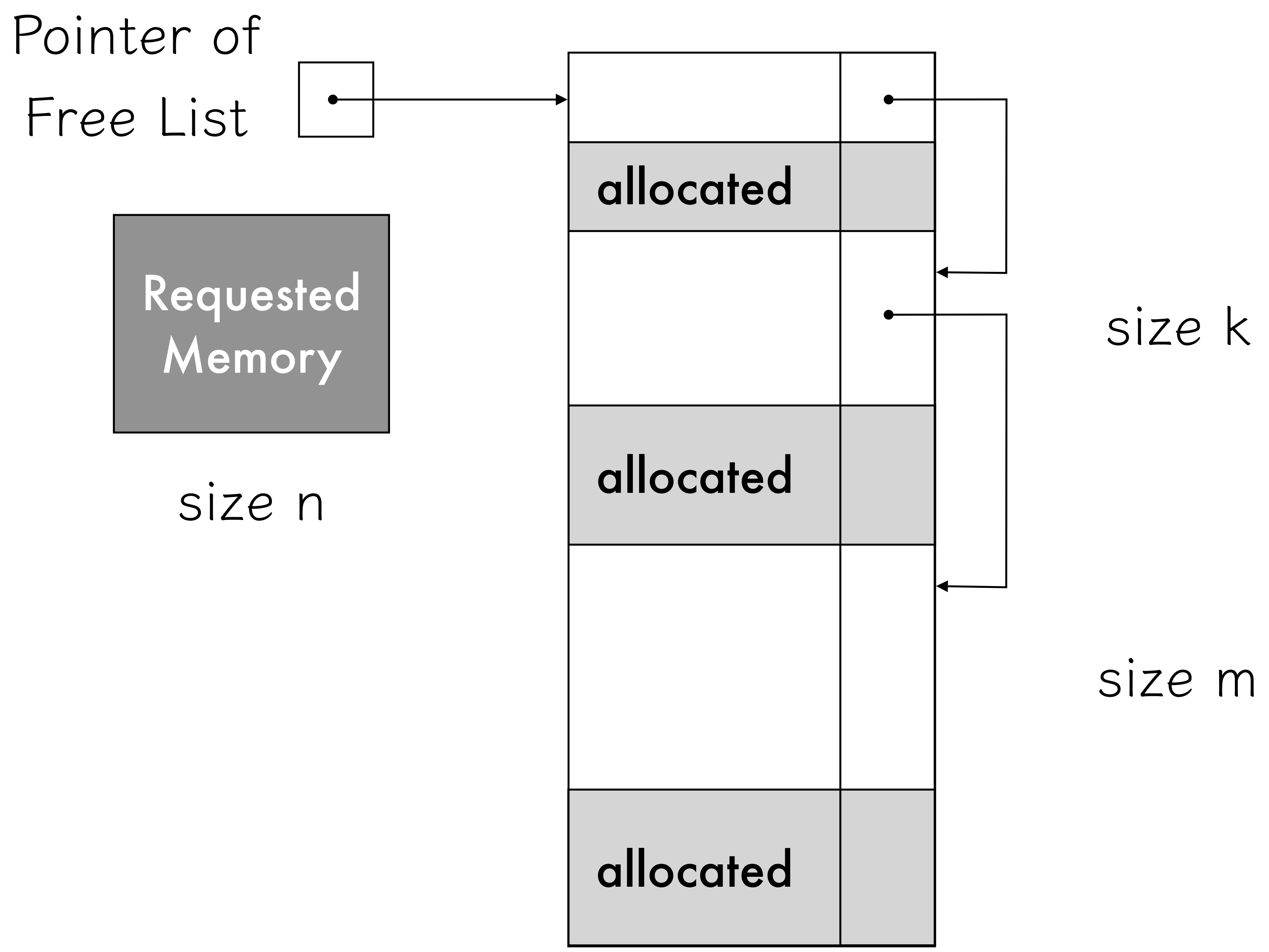
Fixed Length Blocks

- When there is a request,
  - the first block is assigned, and
  - the block is removed from the free list.
- When a block is freed (or deallocated),
  - the block is back to the free list.
- Multiple blocks can be assigned to satisfy the request.



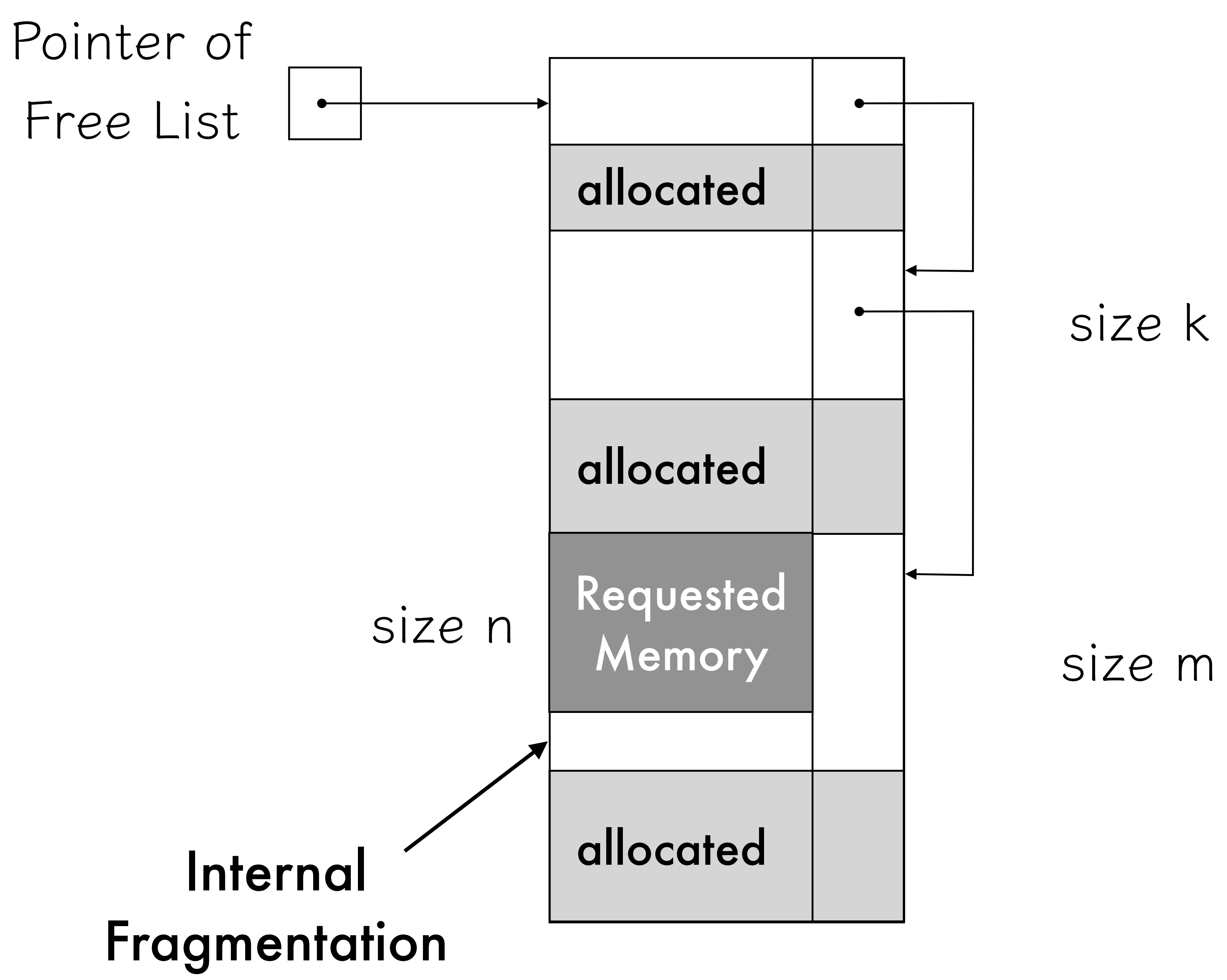
Variable Length Blocks

- Similar to fixed length blocks, it maintains a free list for available blocks.
- The size of blocks can be different.
- When a request for memory of size n, it allocates a free block fits to this size.
  - e.g.) n > k and n < m.
  - The third free block is allocated.



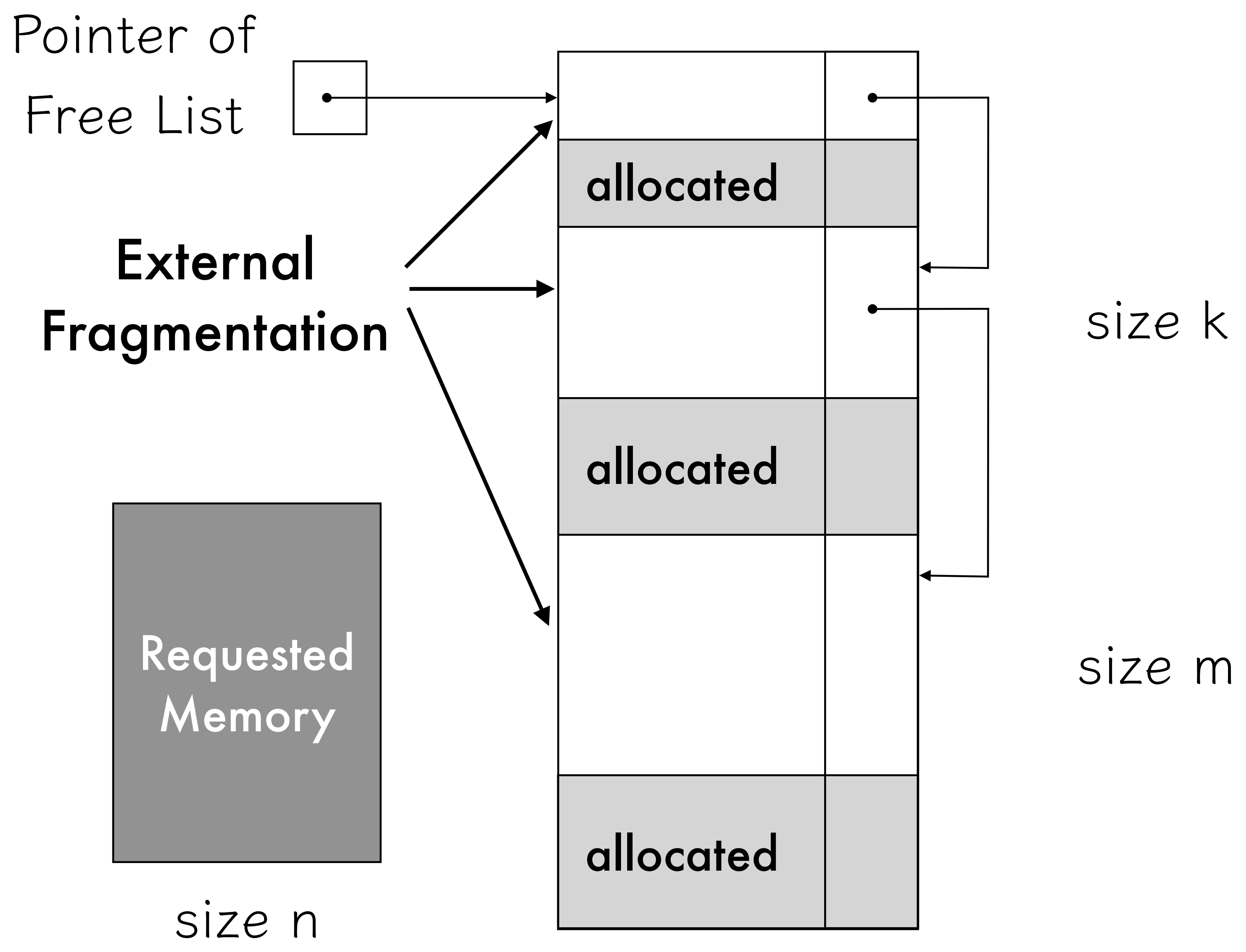
Fragmentation

- Variable length method causes fragmentation.
- Due to fragmentation, memory space is wasted,
  - or it reduces the performance of programs.
- ***Internal Fragmentation***: Allocated block size is greater than the requested size.
  - m > n, then d = m - n is wasted.



Fragmentation

- ***External Fragmentation***
  - Due to the scattered free blocks, requested memory cannot be allocated,
    - even if there exists enough space.
  - m + k > n, but they are not consecutive.



Using Single Free List

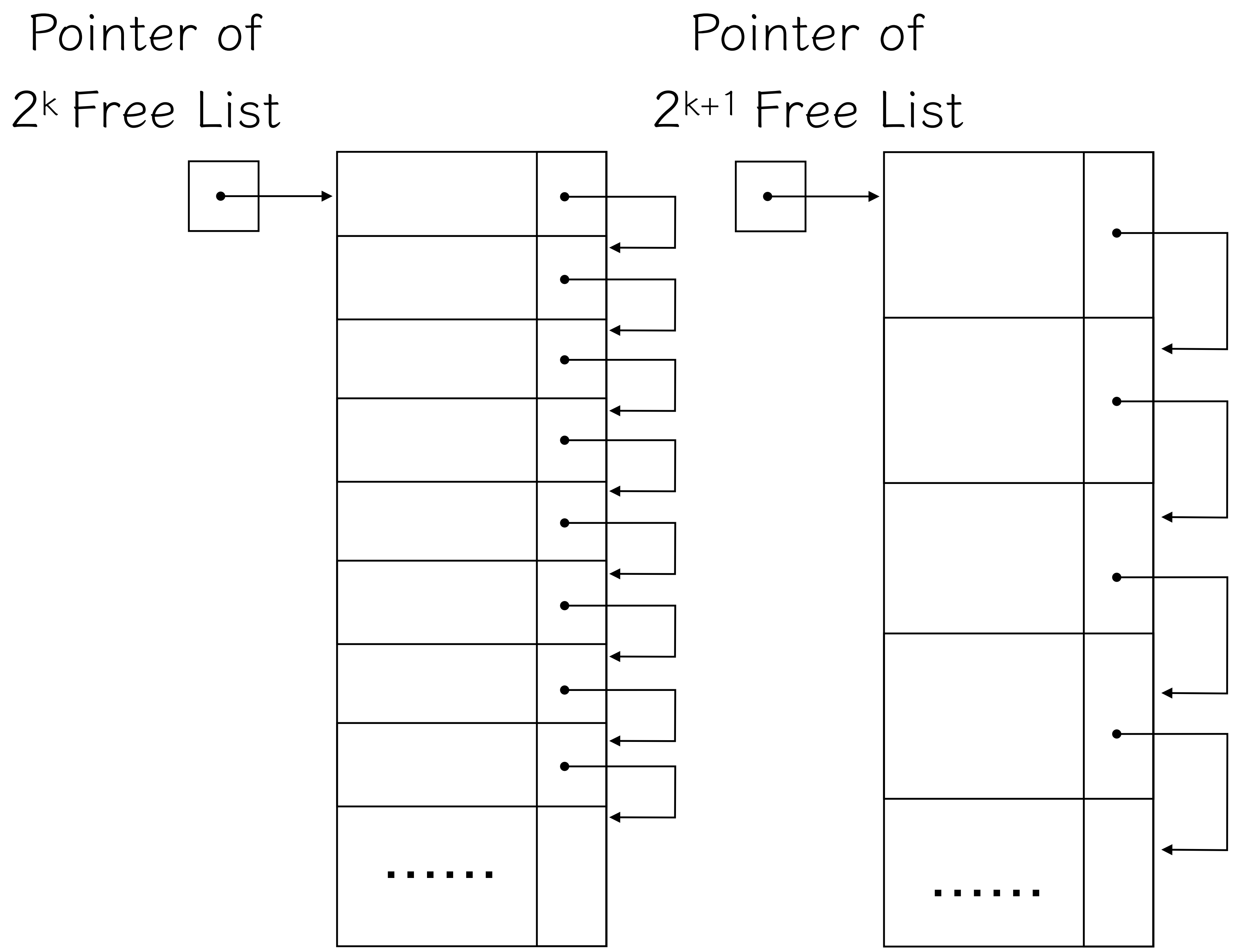
- When there is a request for memory allocation of size *n*,
- Directly use the free list.
    - *First Fit*: allocate the first block bigger than size *n*.
    - *Best Fit*: allocate the size *k* >= *n* block which has the minimum *d* = *k* - *n*.
- Free Memory Compaction.
    - When the end of the heap is reached, move all active blocks to the end.

Multiple Free Lists

- Buddy System
  - Have multiple free lists with size power of 2 (i.e. 2^n).
  - For size n request, find a block from the free list of $2^k >= n$ blocks.
  - If there is no available block, then search 2^{k+1} free list next.
- Fibonacci Heap
  - Instead of 2^n free lists, use Fibonacci numbers as block sizes in free lists.
  - Fib(n) = Fib(n-1) + Fib(n-2)

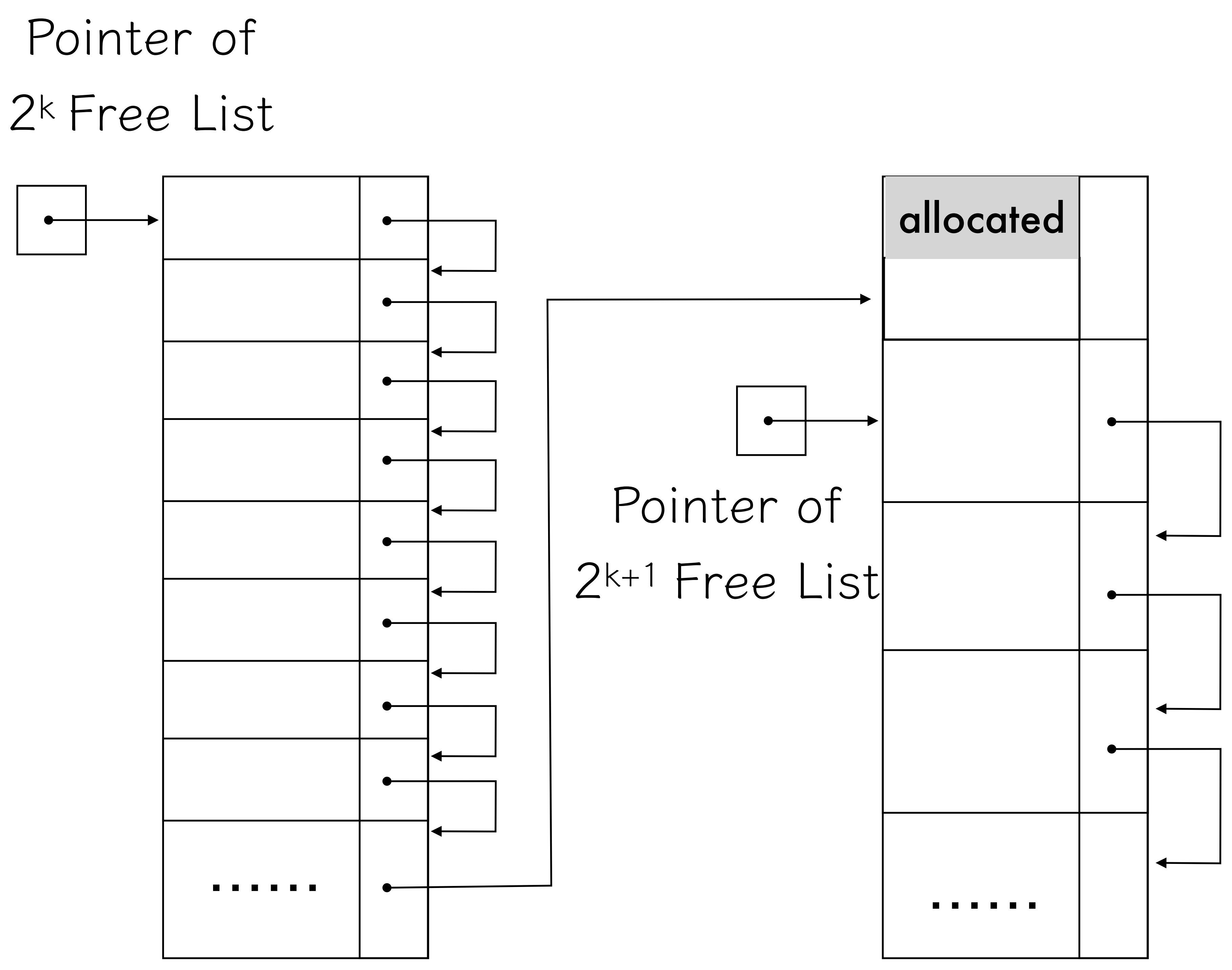
Multiple Free Lists

- **Buddy System**
  - Have multiple free lists with size power of 2 (i.e. 2^n).
  - For size n request, find a block from the free list of 2^k >= n blocks.
  - If there is no available block, then search 2^{k+1} free list next.



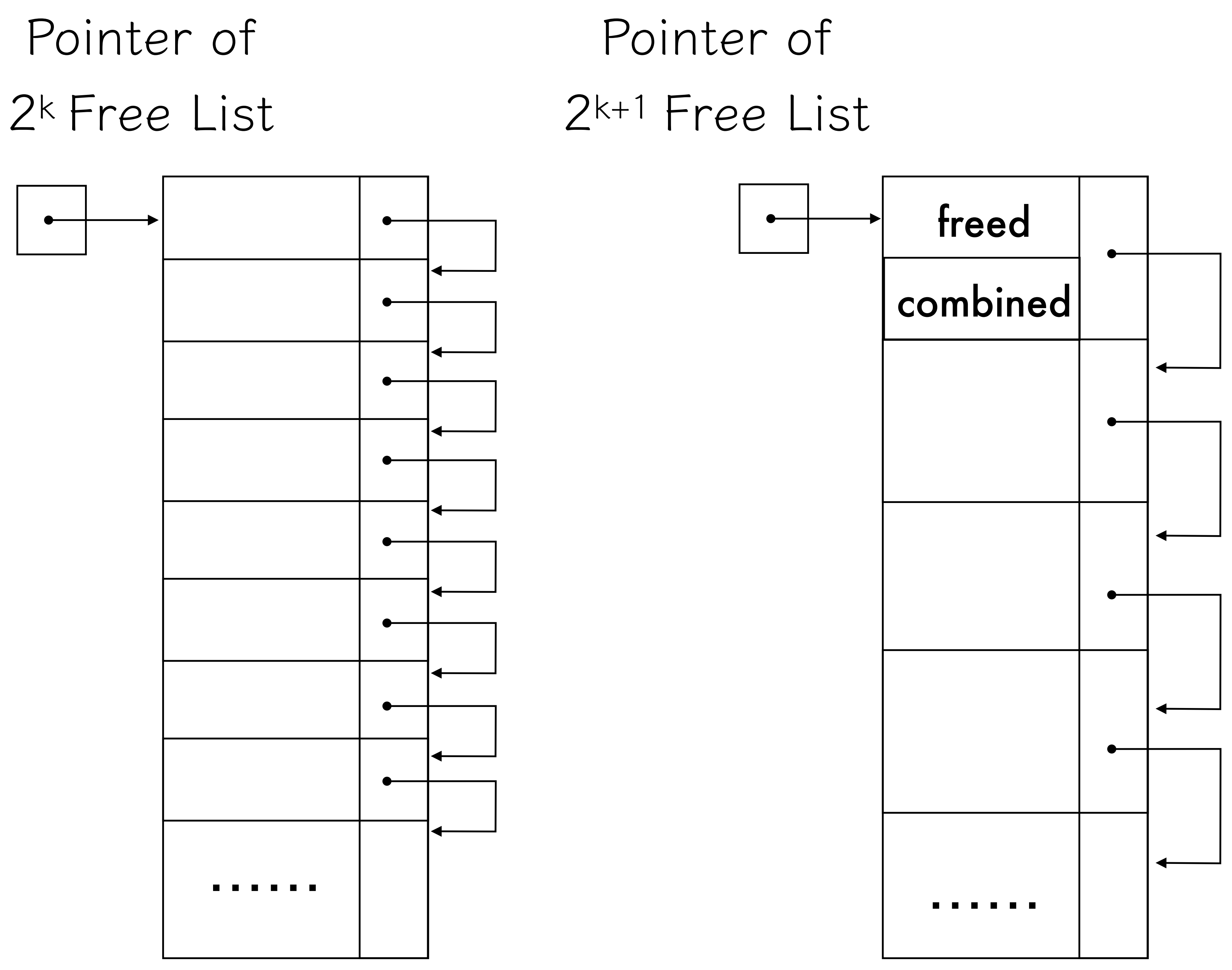
Multiple Free Lists

- When a free block is found in 2^{k+1} free list,
  - Split this block into two 2^k blocks.
  - Allocate one of them, and connect the other to 2^k free list.



Multiple Free Lists

- Next time the allocated block is freed,
  - Find its *buddy* which is resulted by the split, and check it is also free.
  - Combine them and attach it to 2^{k+1} free list again.



Scope Rule Implementation

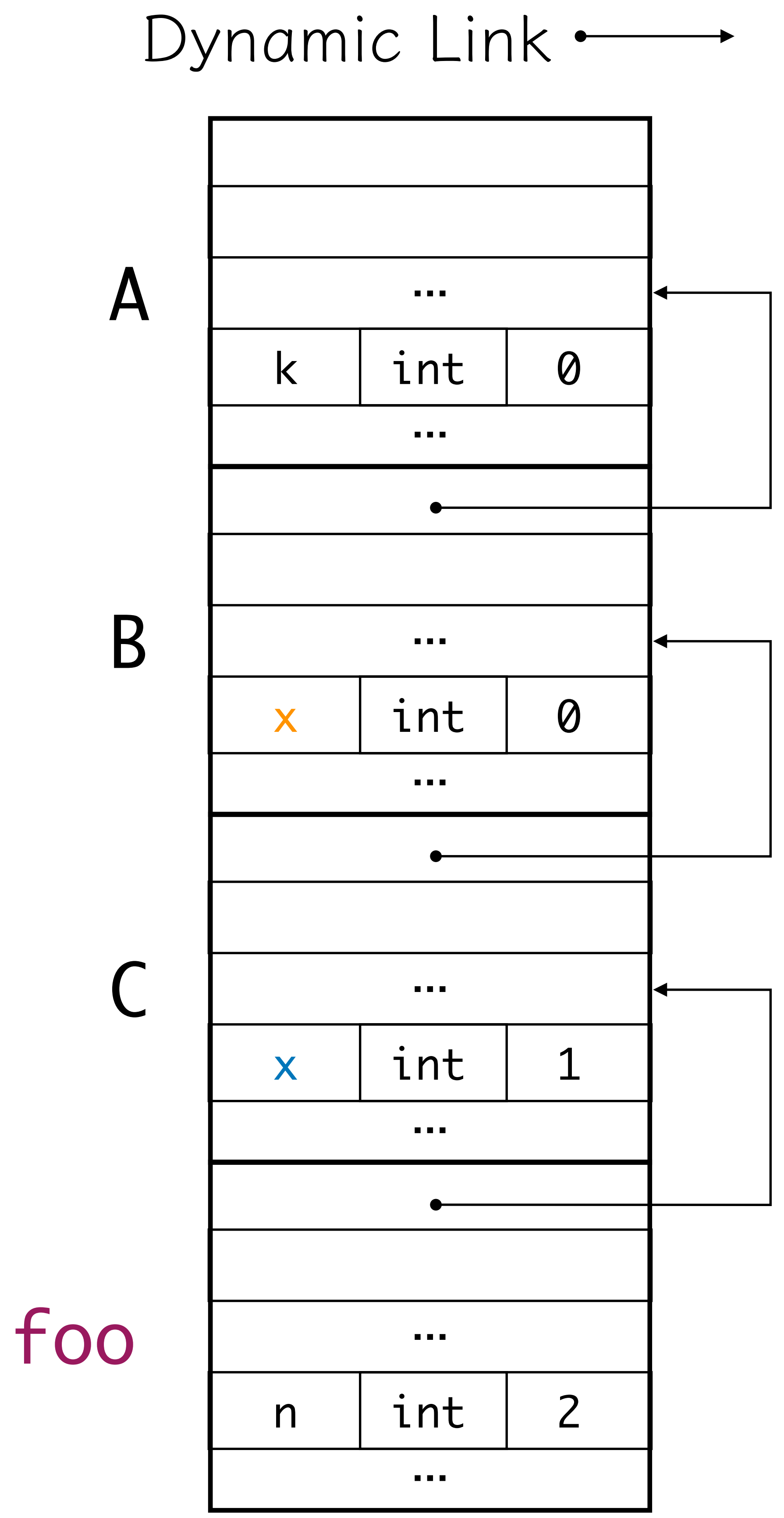
Scope Rule Implementation

- Static Scope Rule Implementation
  - Static Link
  - The Display
- Dynamic Scope Rule Implementation
  - Association Lists and CRT

Static Scope Rule Implementation

- Static scope rule implementation requires more management than simply using the stack.
- In procedure foo, variable x refers to x in Block B, not C.
- However, the activation record connected to foo with dynamic link is Block C.

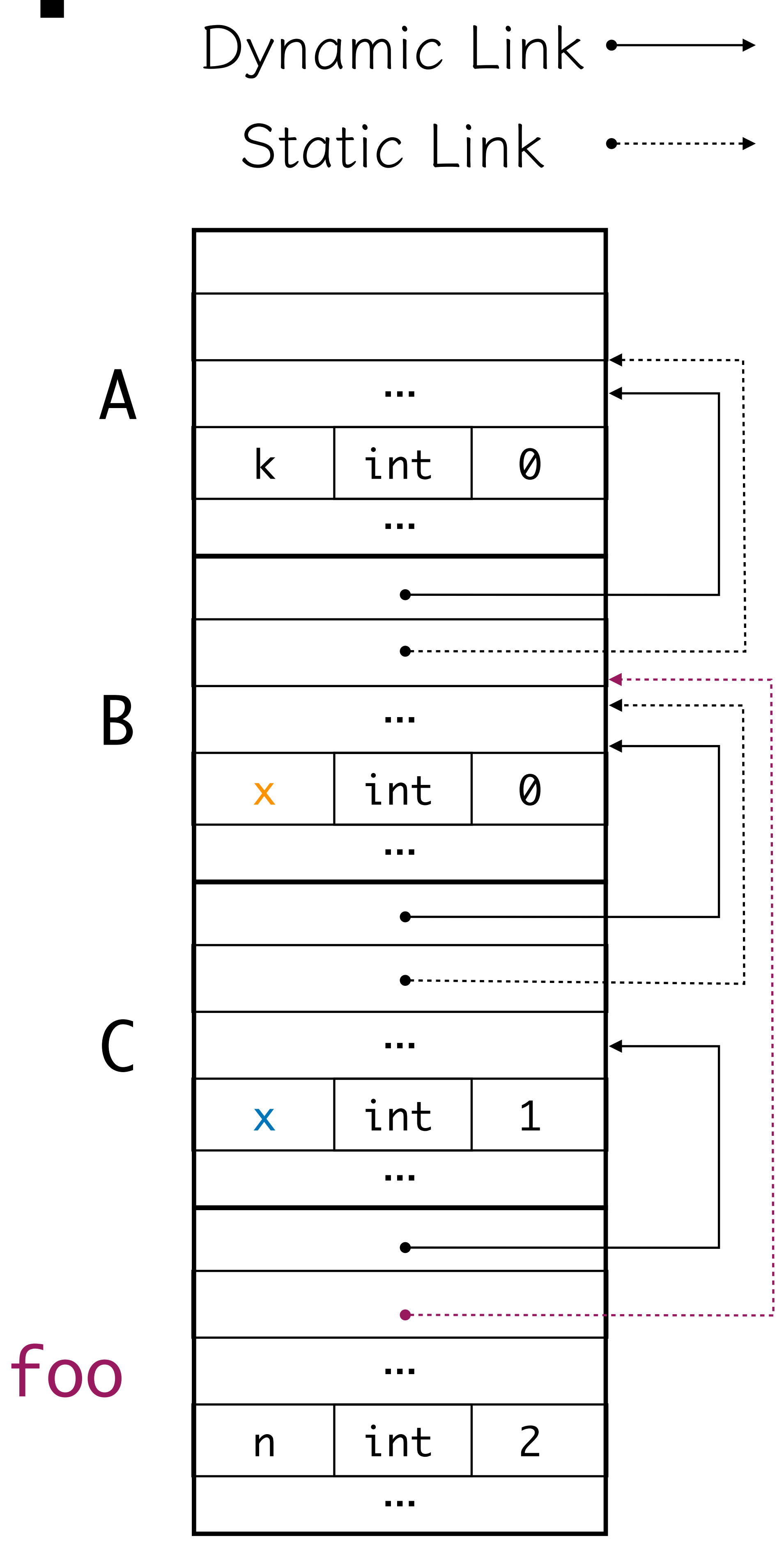
```
A:{ int k=0;
  B:{
    int x=0;
    void foo(int n) {
      x = n-1;
      k = n+1;
    }
    C:{
      int x=1;
      foo(2);
    }
  }
}
```



Static Scope Rule Implementation

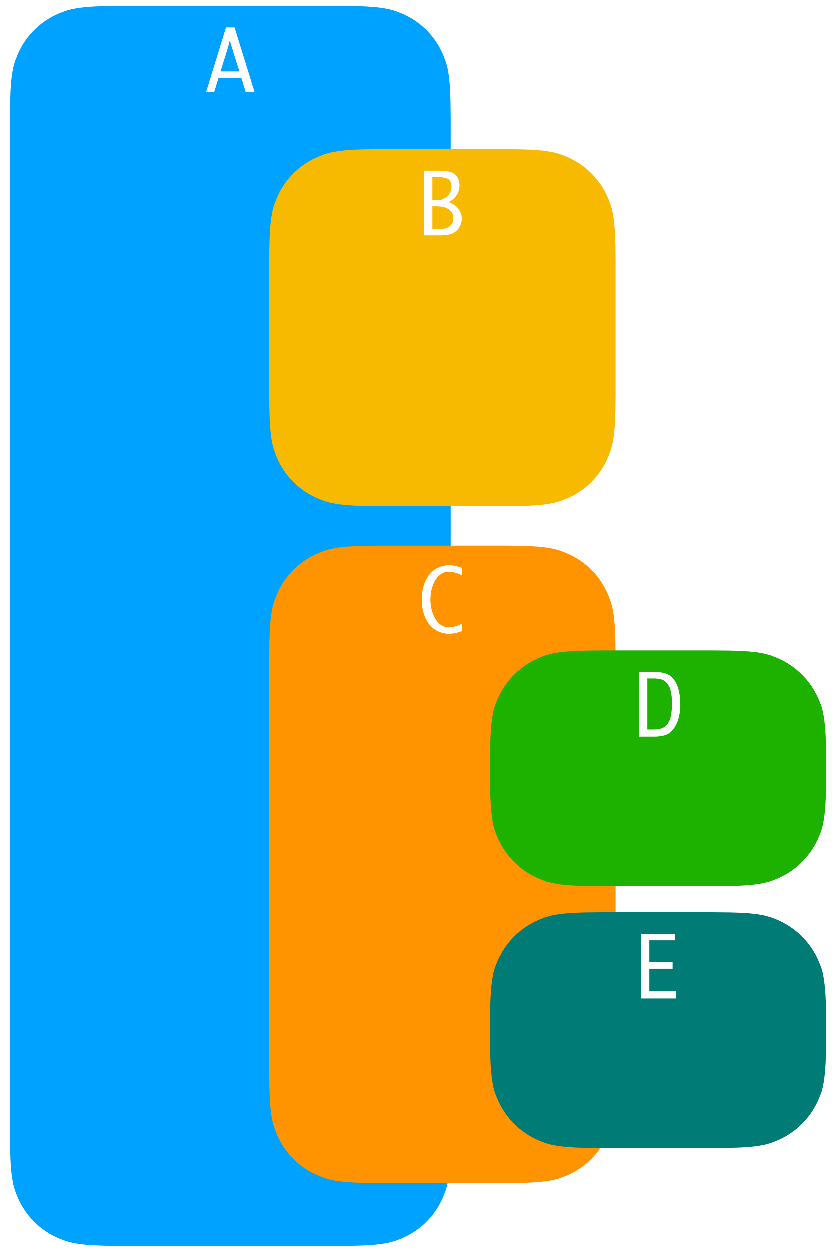
- So we need one more link to the block immediately enclosing the block.
- Using ***static link*** (dotted line) to point the enclosing block.
- When managing static scope, use static links instead of dynamic links.

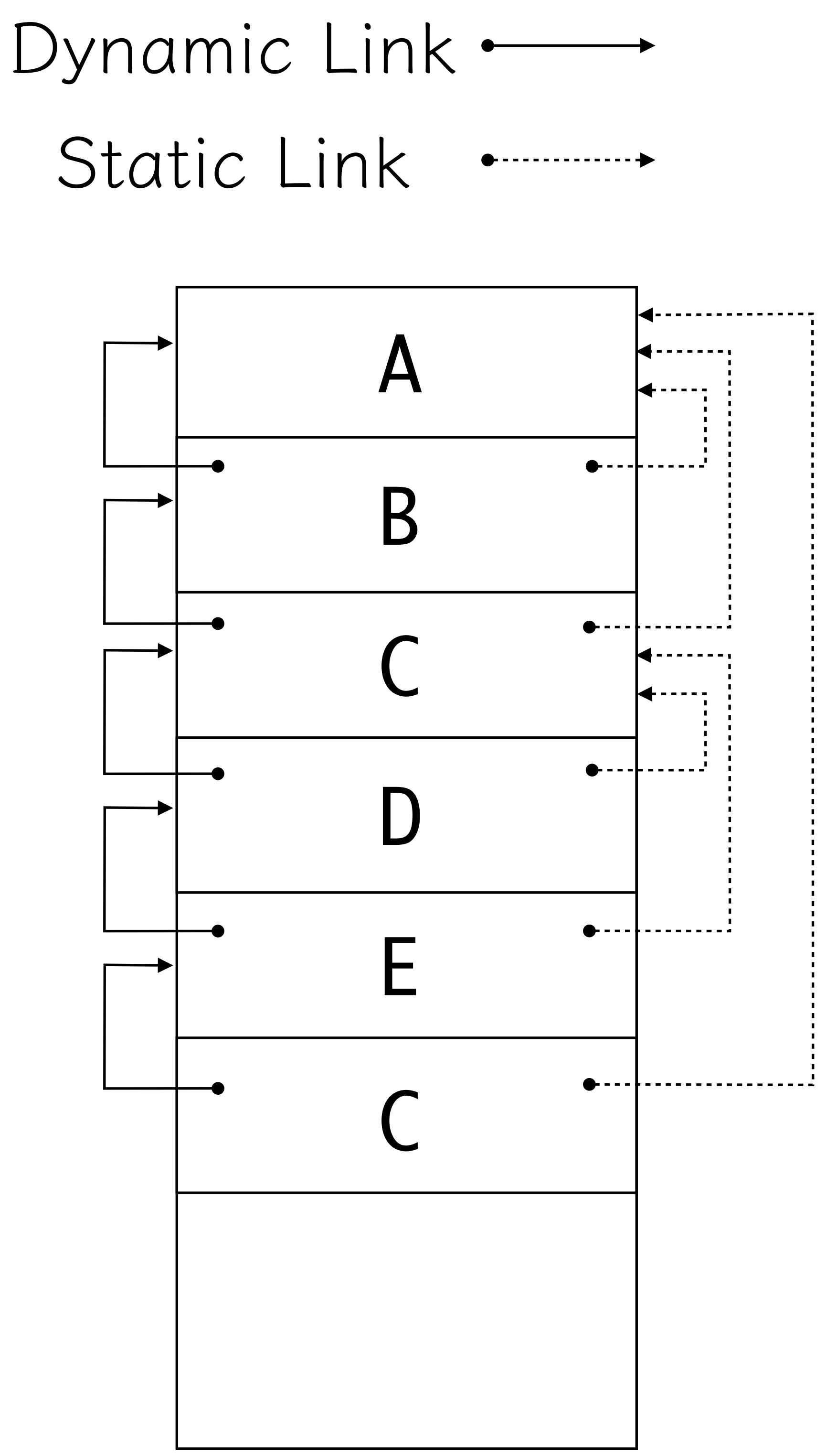
```
A:{ int k=0;
  B:{
    int x=0;
    void foo(int n) {
      x = n-1;
      k = n+1;
    }
    C:{
      int x=1;
      foo(2);
    }
  }
}
```



Static Link

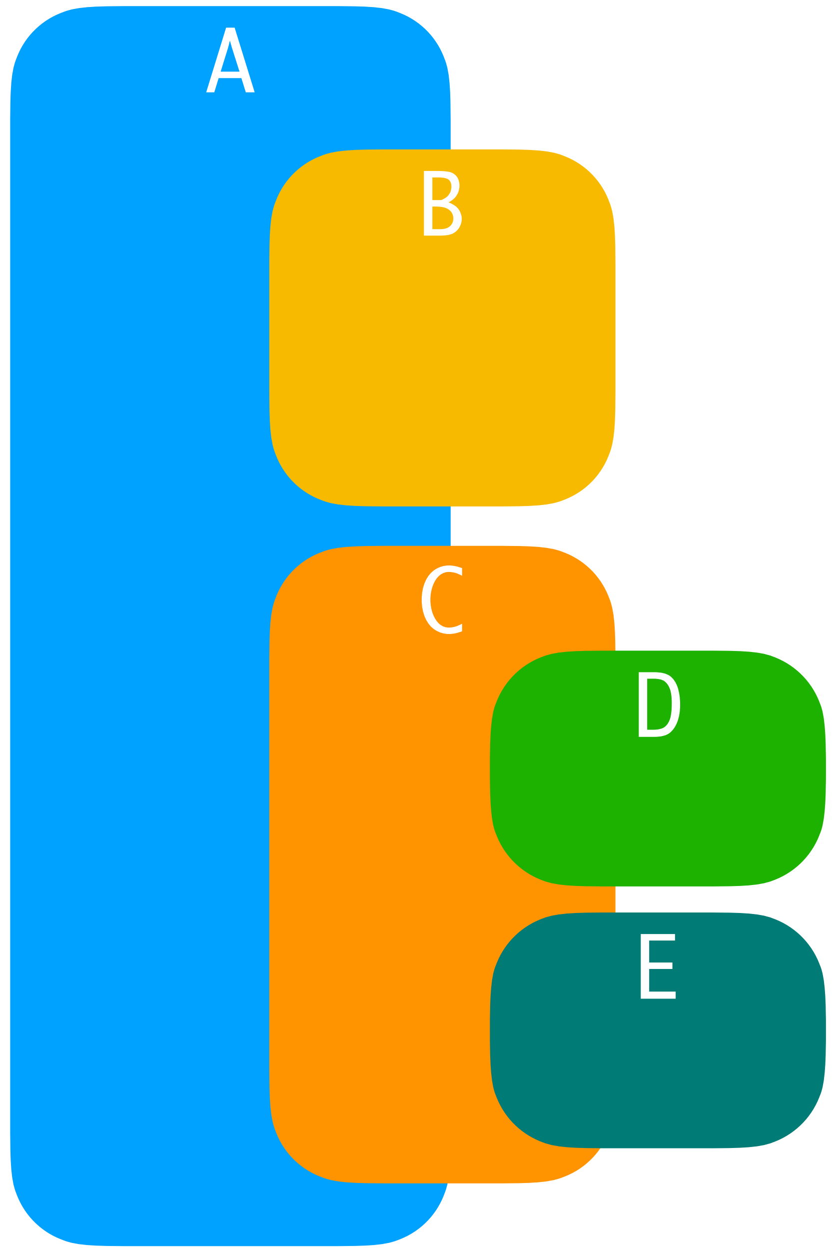
- Consider an example with the following structures.
- Blocks B, C are inside Block A.
- Blocks D, E are enclosed in Block C.
- There is a sequence of calls,
    - A, B, C, D, E, C

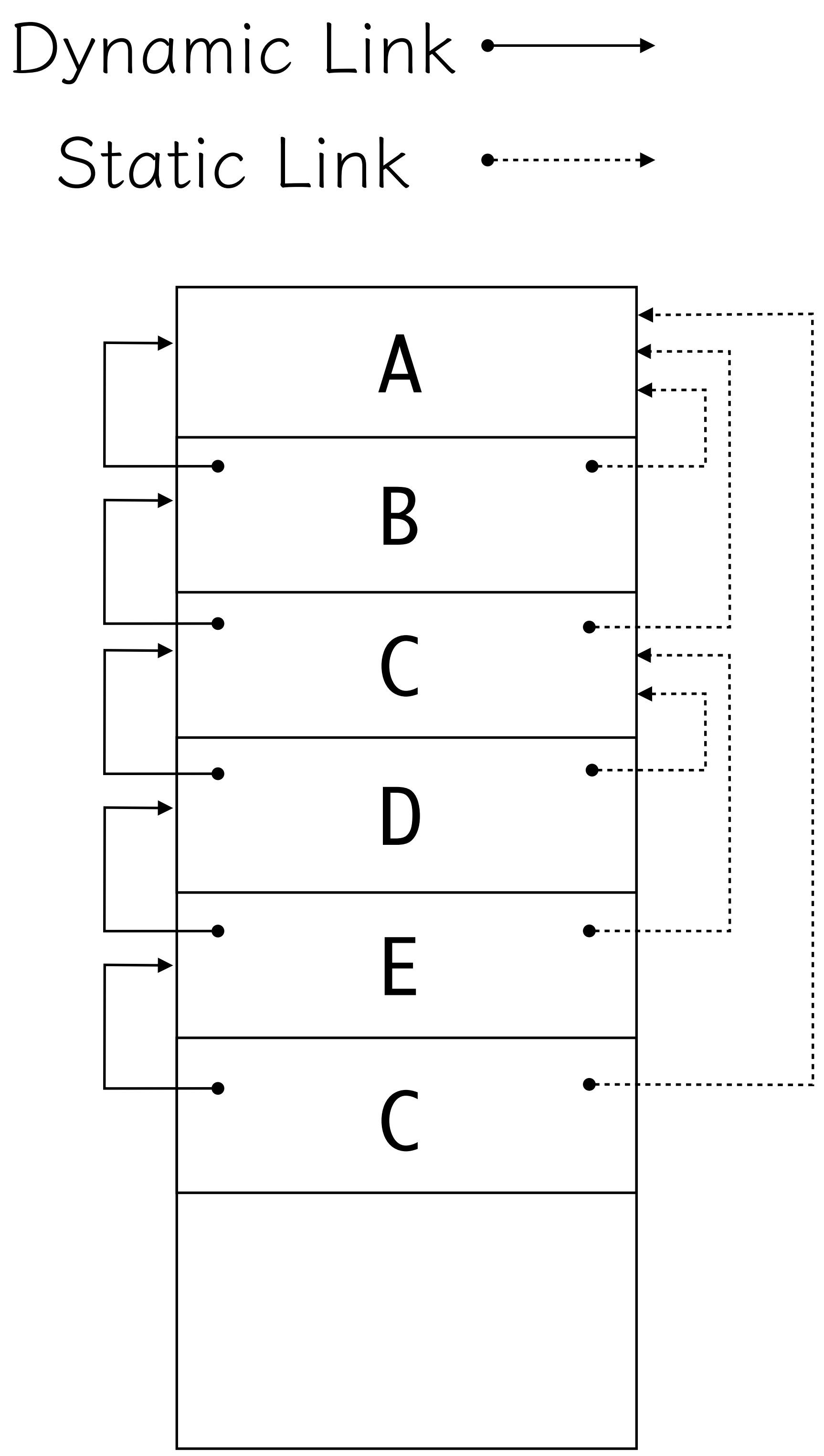




Static Link

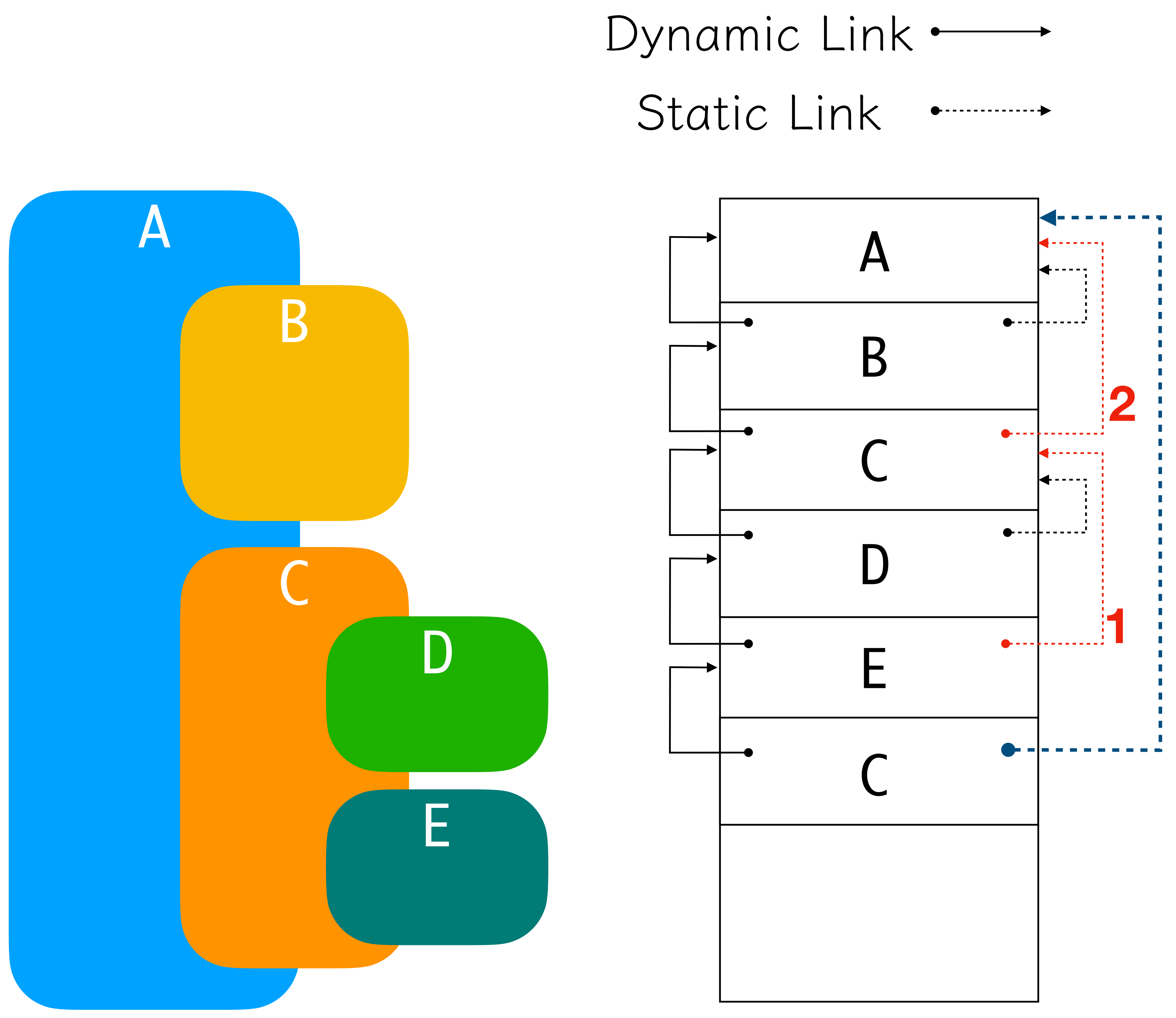
- When activation records are pushed to the stack, it is necessary to decide the address for static link.
- In most common approach, the caller calculates the link and passes it to the callee.
- There are two possible cases.





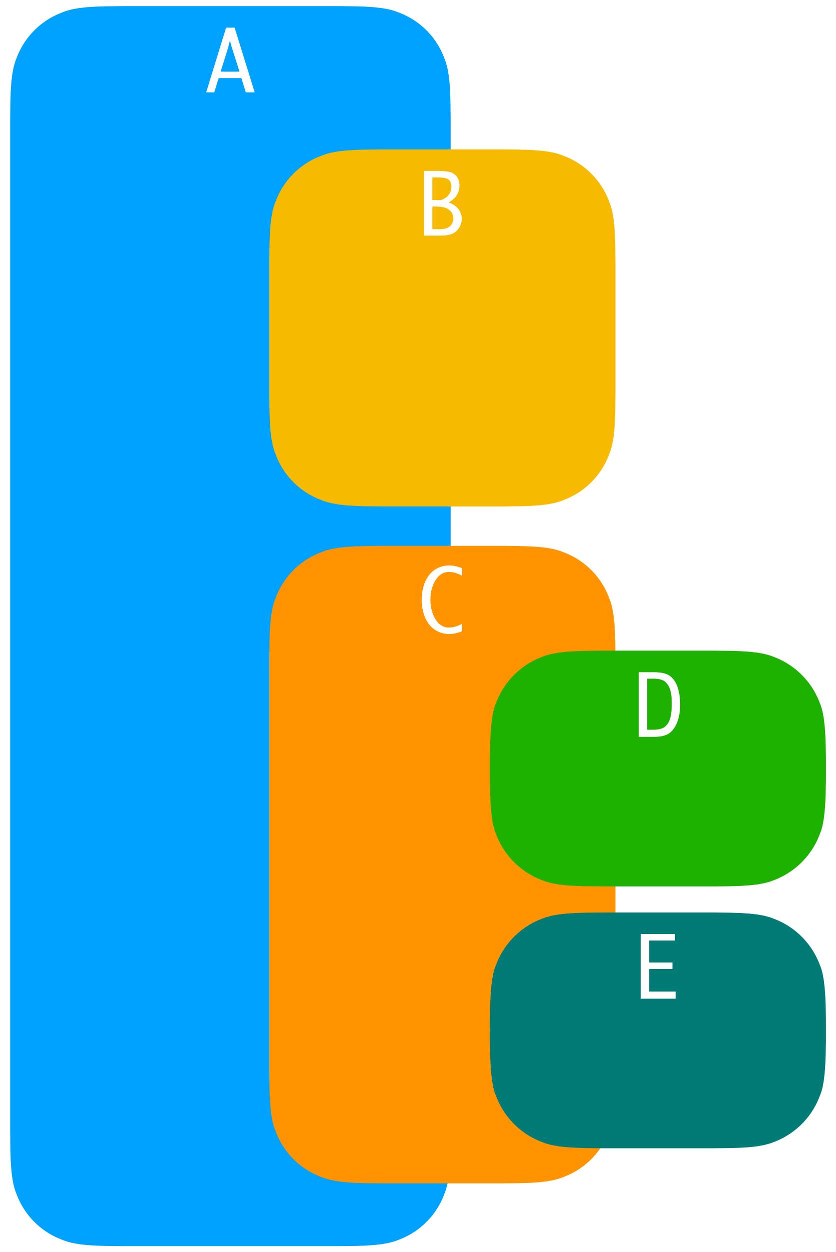
Static Link

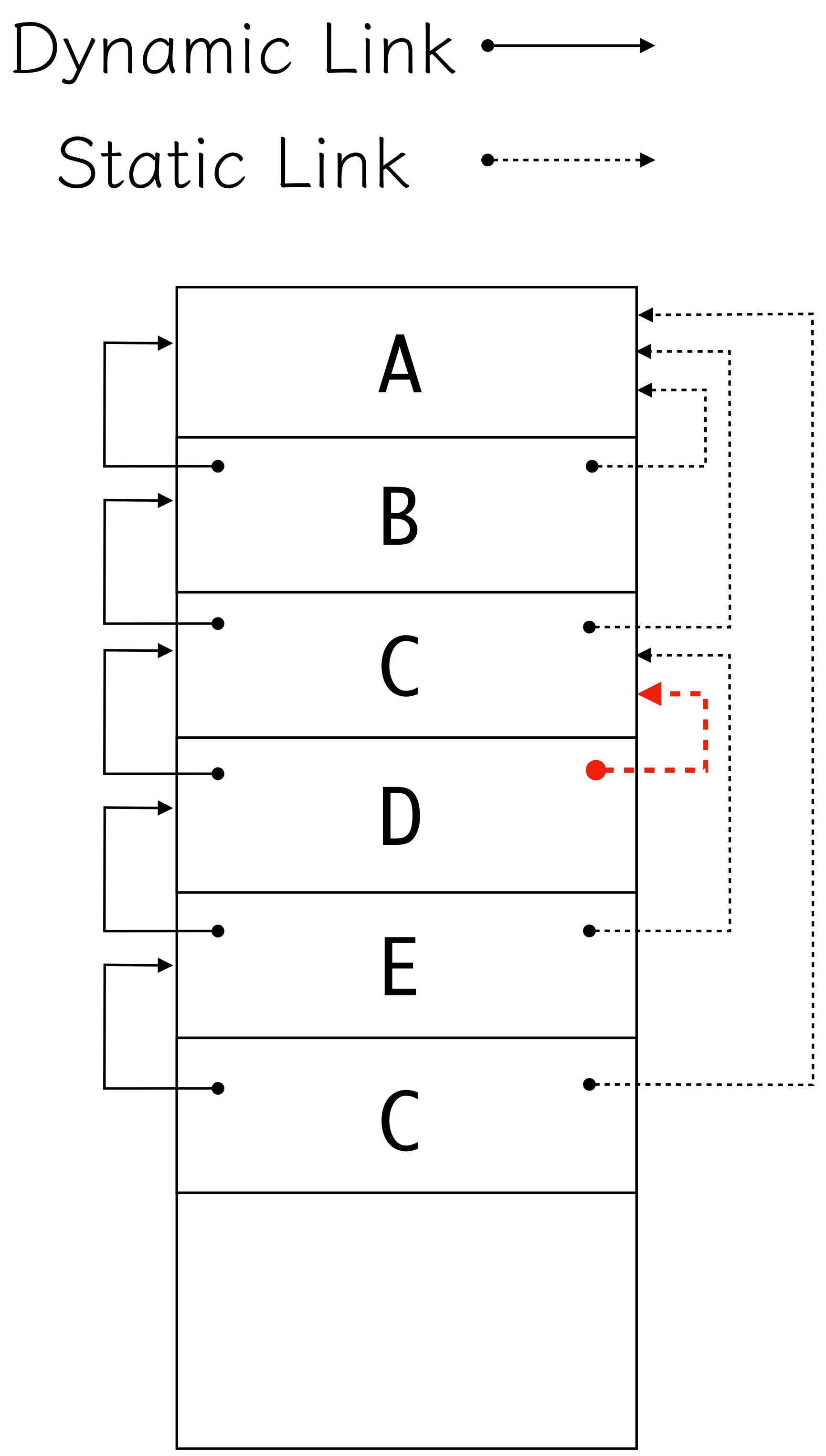
- **Case 1**: Callee is outside of the caller (e.g. E calls C).
    - Callee must be in an outer block of the caller based on visibility rules.
    - Hence the activation record of the callee must be in the stack already.
    - So we can backtrace the static links to find a new link for the new block.



Static Link

- **Case 2**: Callee is inside the caller (e.g. C calls D).
  - Visibility rules guarantee that the callee is declared in the same block which the call is occurred.
  - Hence we can simply use a static link to the caller.



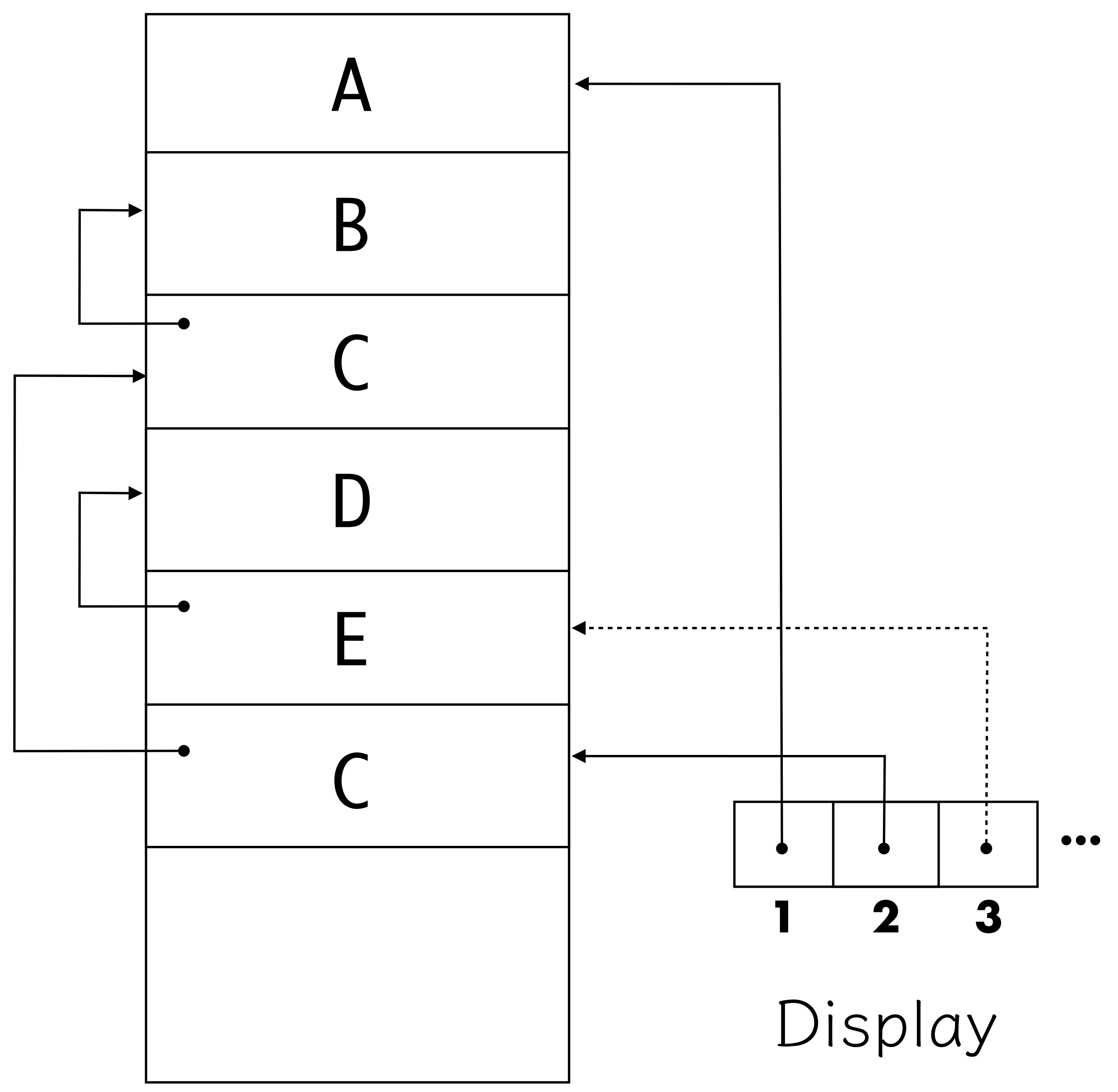


The Display

- Static Link requires several memory access for each procedure call.
- If a non-local name is declared *k* levels of block away, we need *k* memory accesses to follow the static links.
- Although we usually don't have too much nesting (i.e. k is not big), we can do better.
- The Display technique only requires ***constant memory accesses (twice)***.

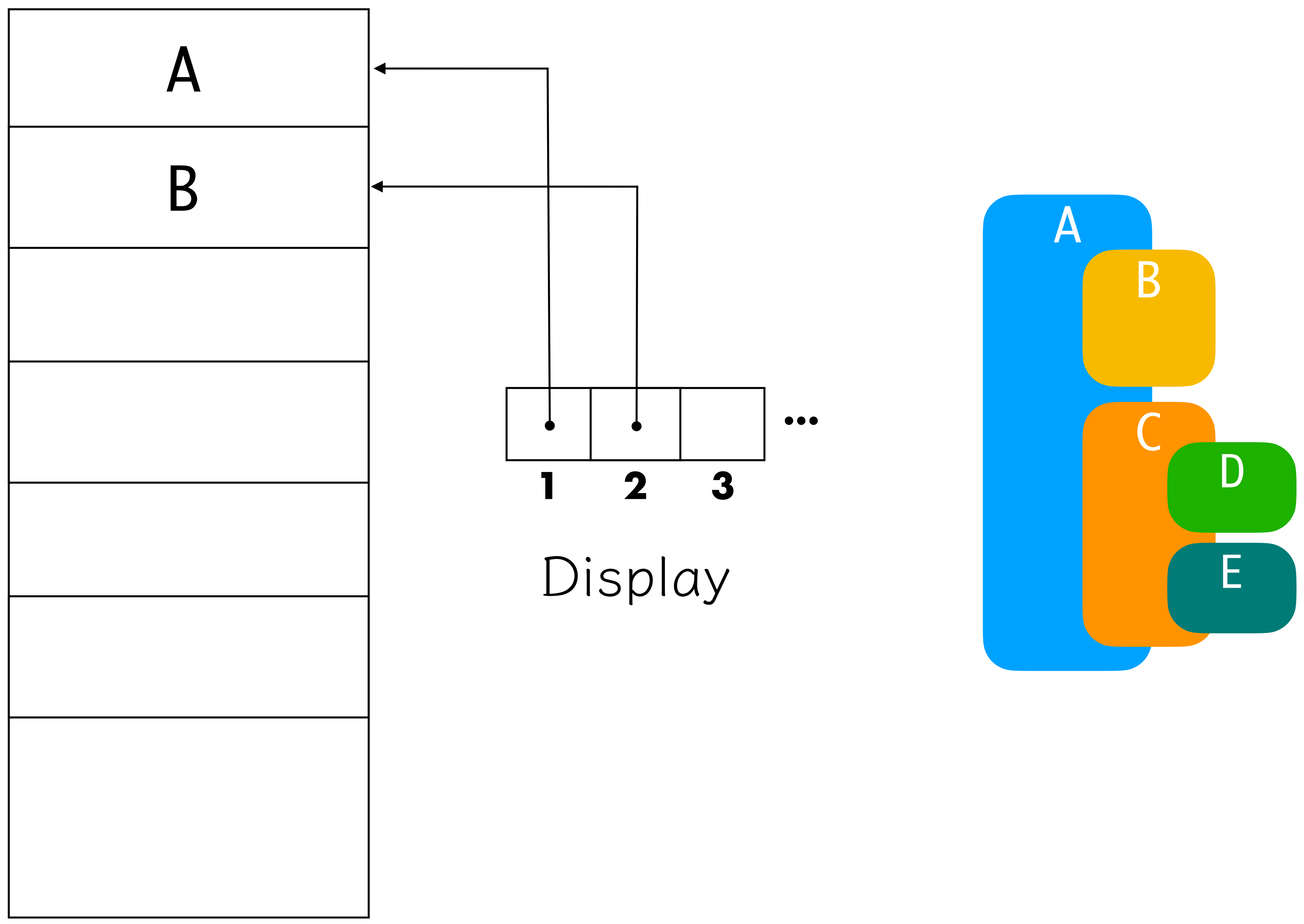
The Display

- The technique employs a vector called ***display***,
  - whose k-th element contains the pointer to the current activation record of k-th nesting level.
- To find a non-local name declared in a block at level n,
  - we can follow the pointer at element n,
  - then use local offset to find the name.



The Display

- Display processing is simple.
- When a procedure is called at level k, display's k-th element is updated as the pointer to the activation record of callee.
- Let's consider the sequence of calls A, B, C, D, E, C again.

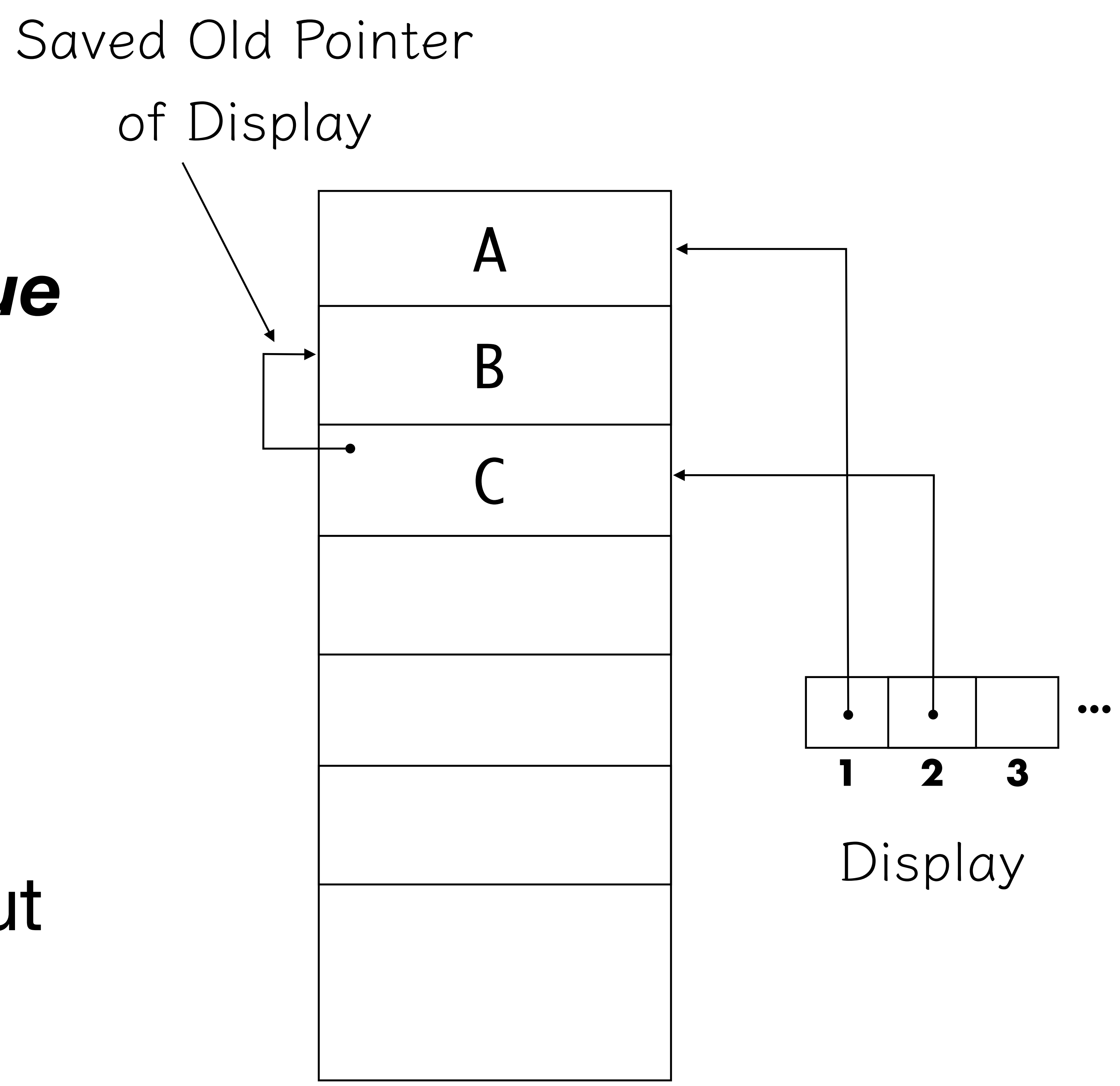


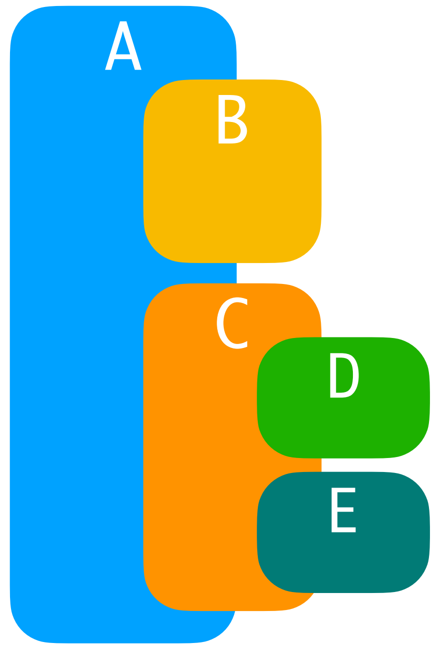
The Display

- When a procedure is called at the same level, ***it is necessary to store the old value*** in display.

- Since Block C is also at level 2, the old pointer to Block B is saved as a link from Block C to B.

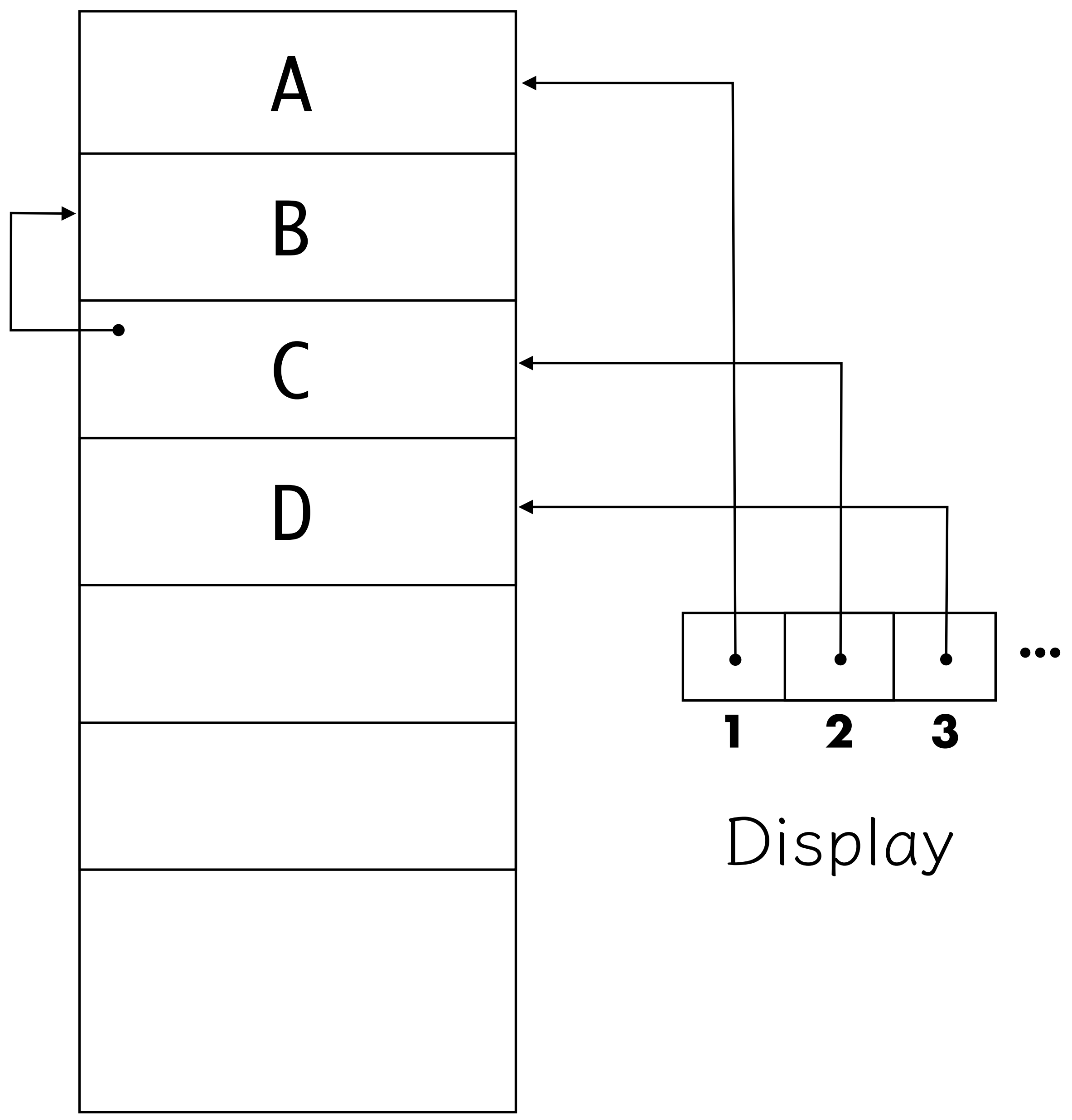
- This old value can be restored after C is out of the stack.

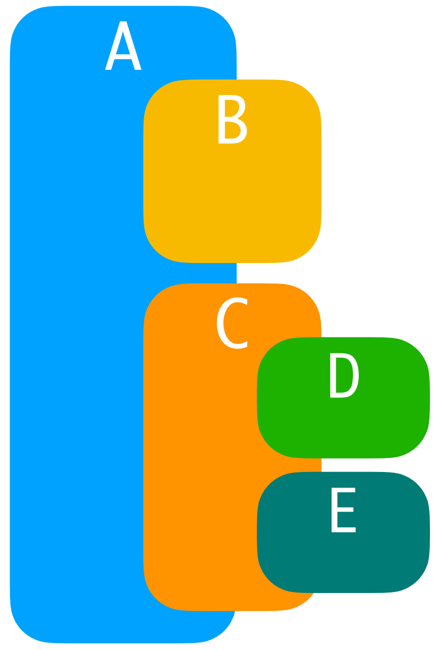




The Display

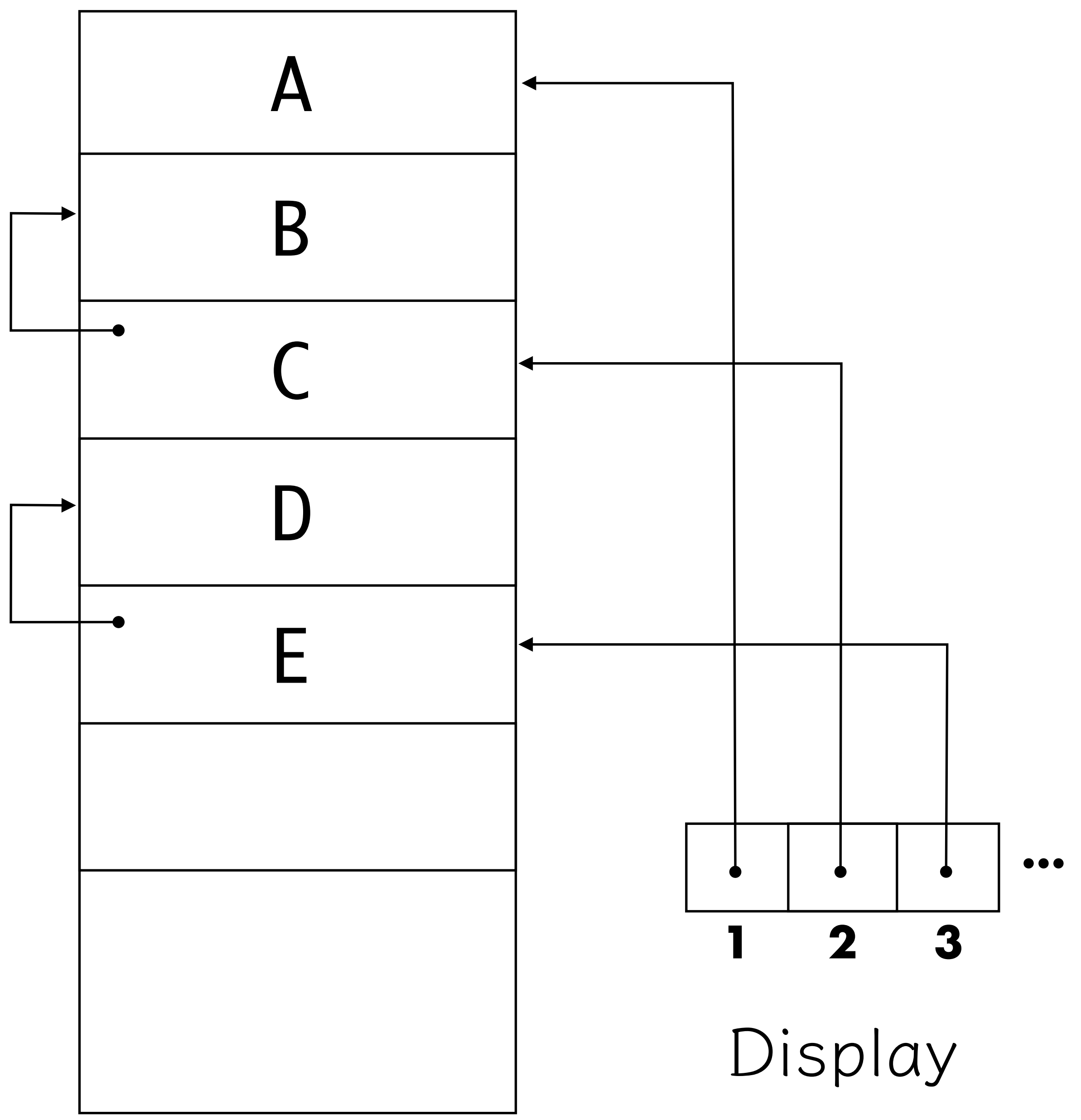
- If a callee (Block D) is inside the caller (Block C), we can increase the display length.
  - then we can put the pointer in the next element (3rd element).

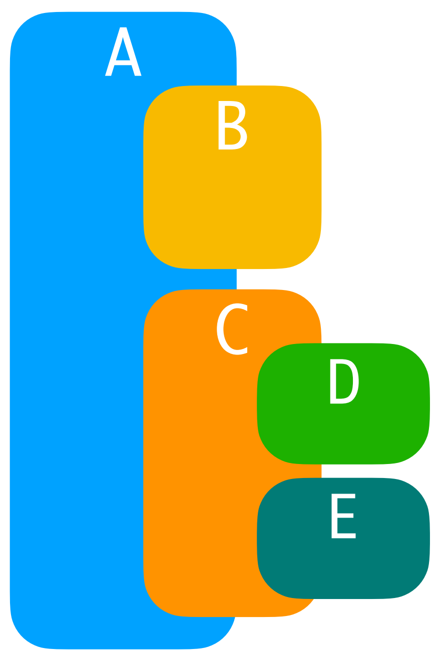




The Display

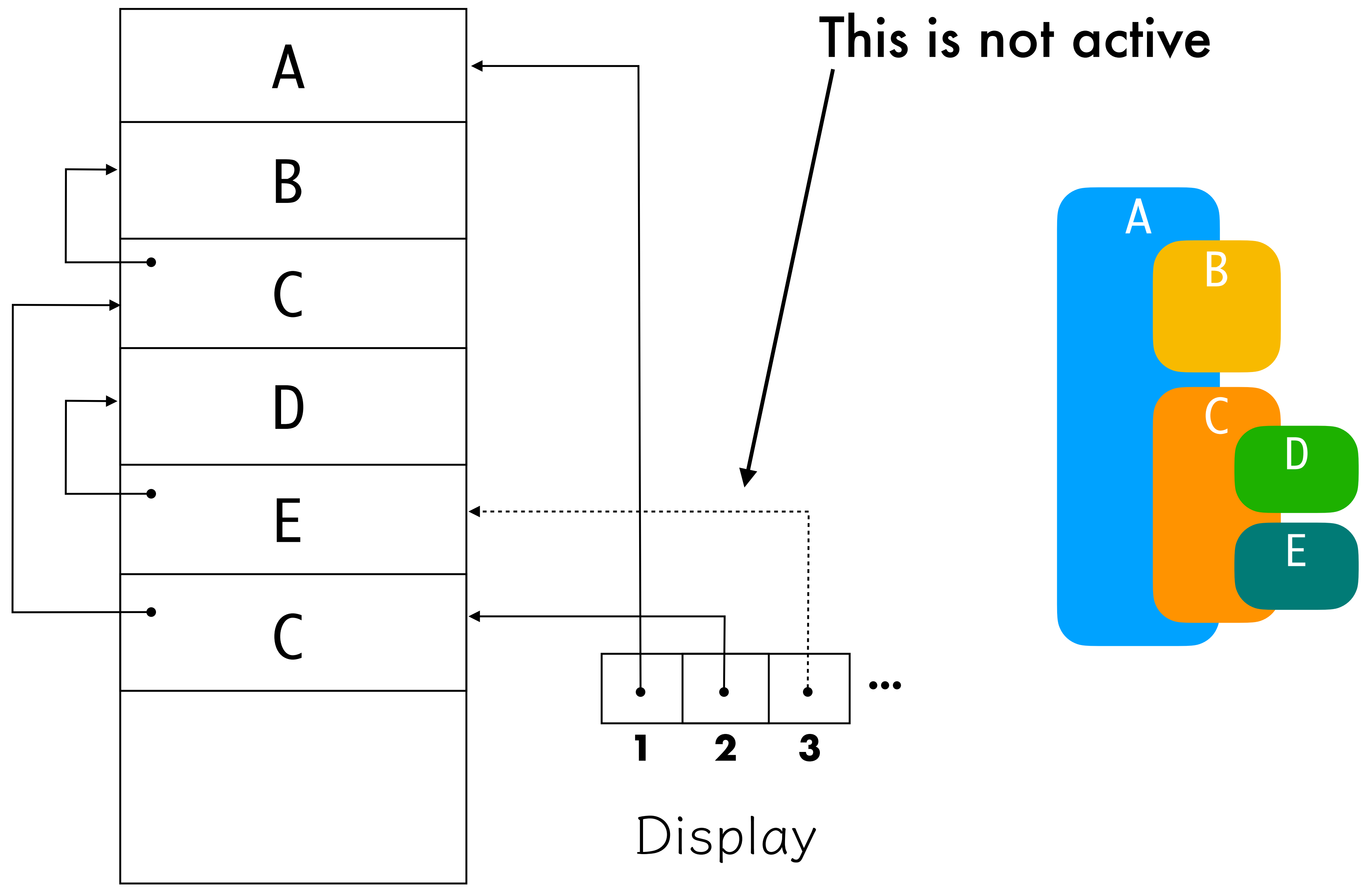
- As Block E is called, old value of element 3 is stored.
- Then the pointer is updated for Block E.
- Suppose variable x is declared in Block C and used in Block E.
    - We know that x is declared in C at compile time (static scope).
    - Block C is at level 2, hence check the display element 2.
    - Then use the local offset to find the binding of x.





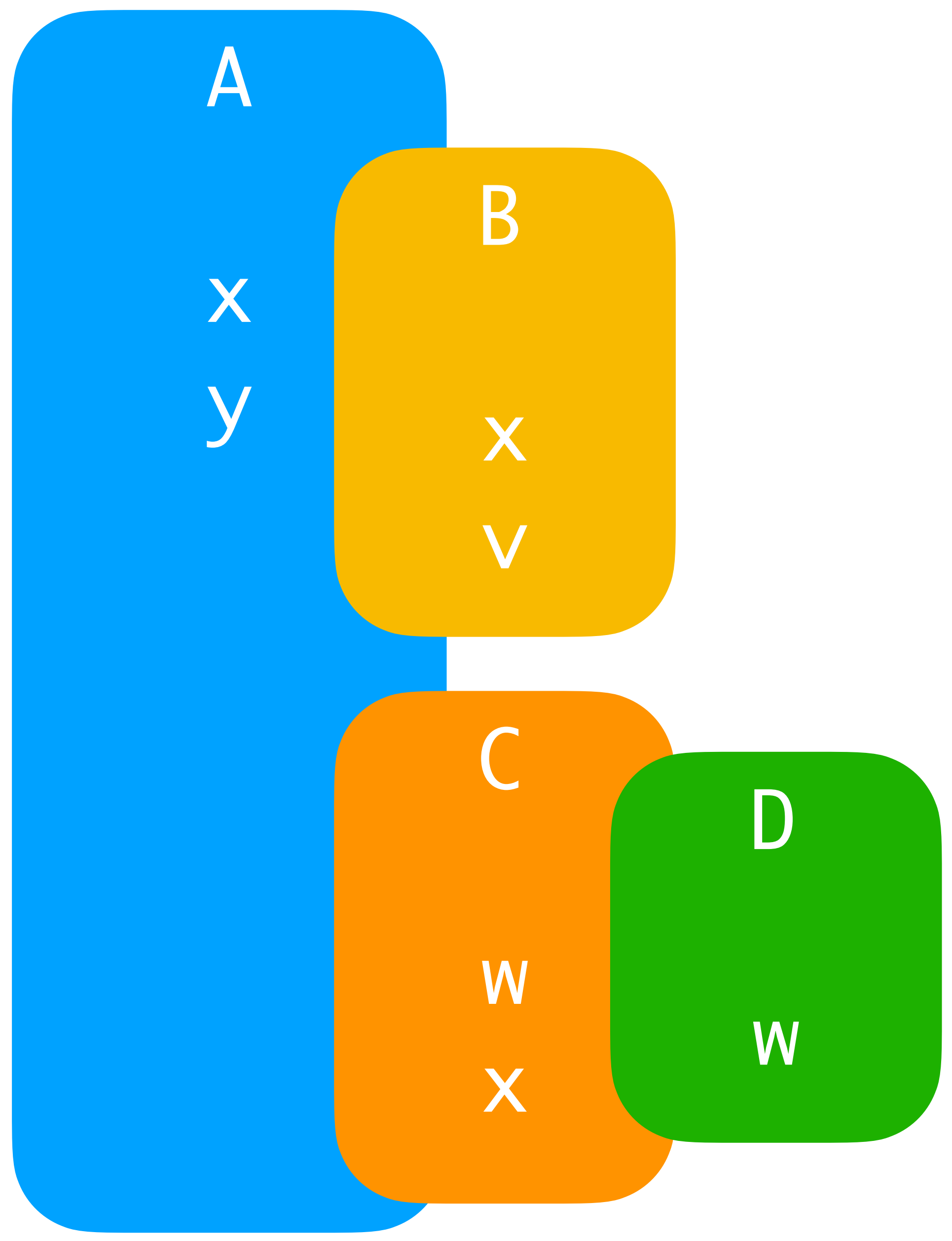
The Display

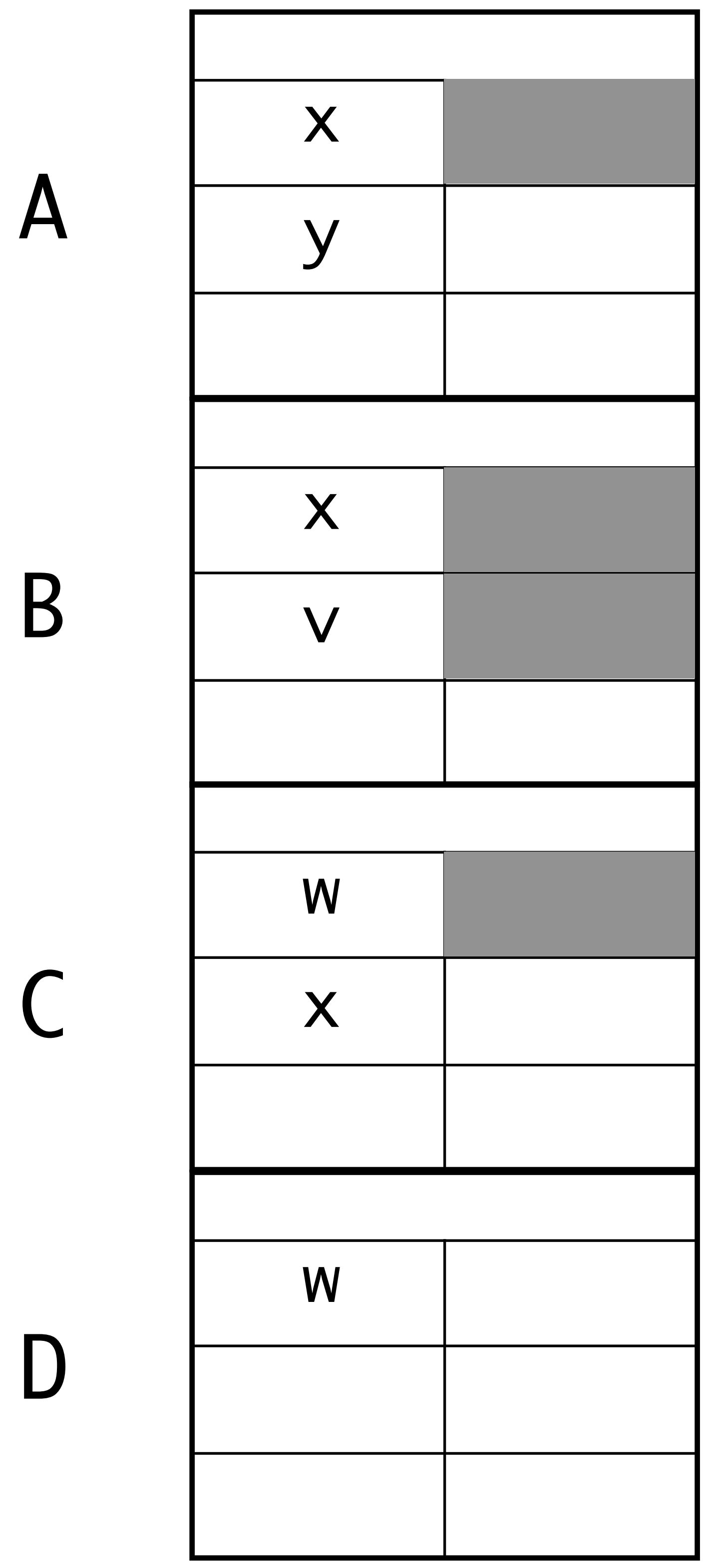
- Block C is called in Block E.
    - We cannot use local names declared in Block E in Block C.
    - Display contains the pointer to C at element 2, hence elements behind this are deactivated.



Dynamic Scope Rule Implementation

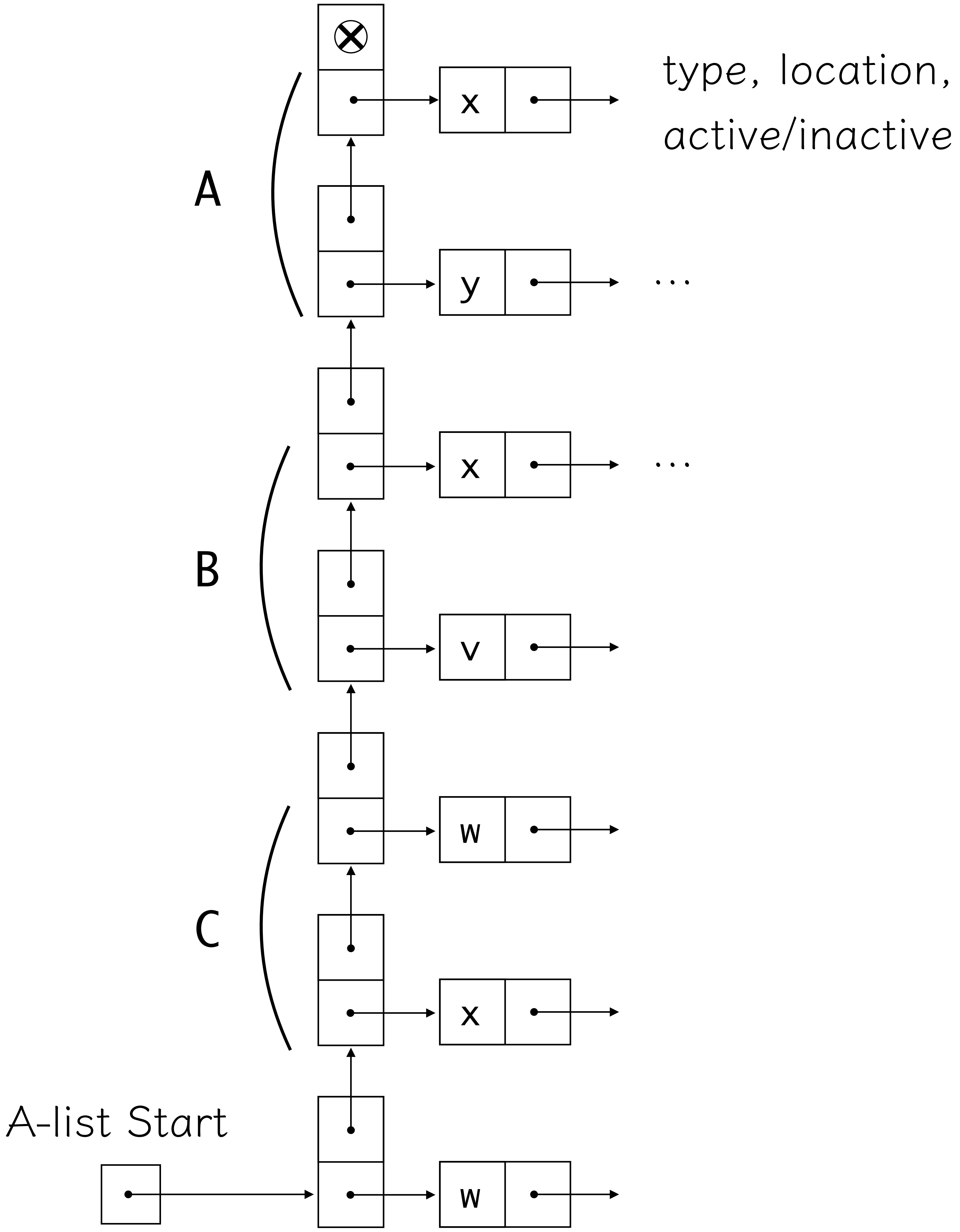
- In dynamic scope, non-local environments are considered in the order of their activations.
- Hence we need to go backward in the stack to find a proper binding.
- Let's consider calls A, B, C, D.
- Grey color means deactivated bindings.





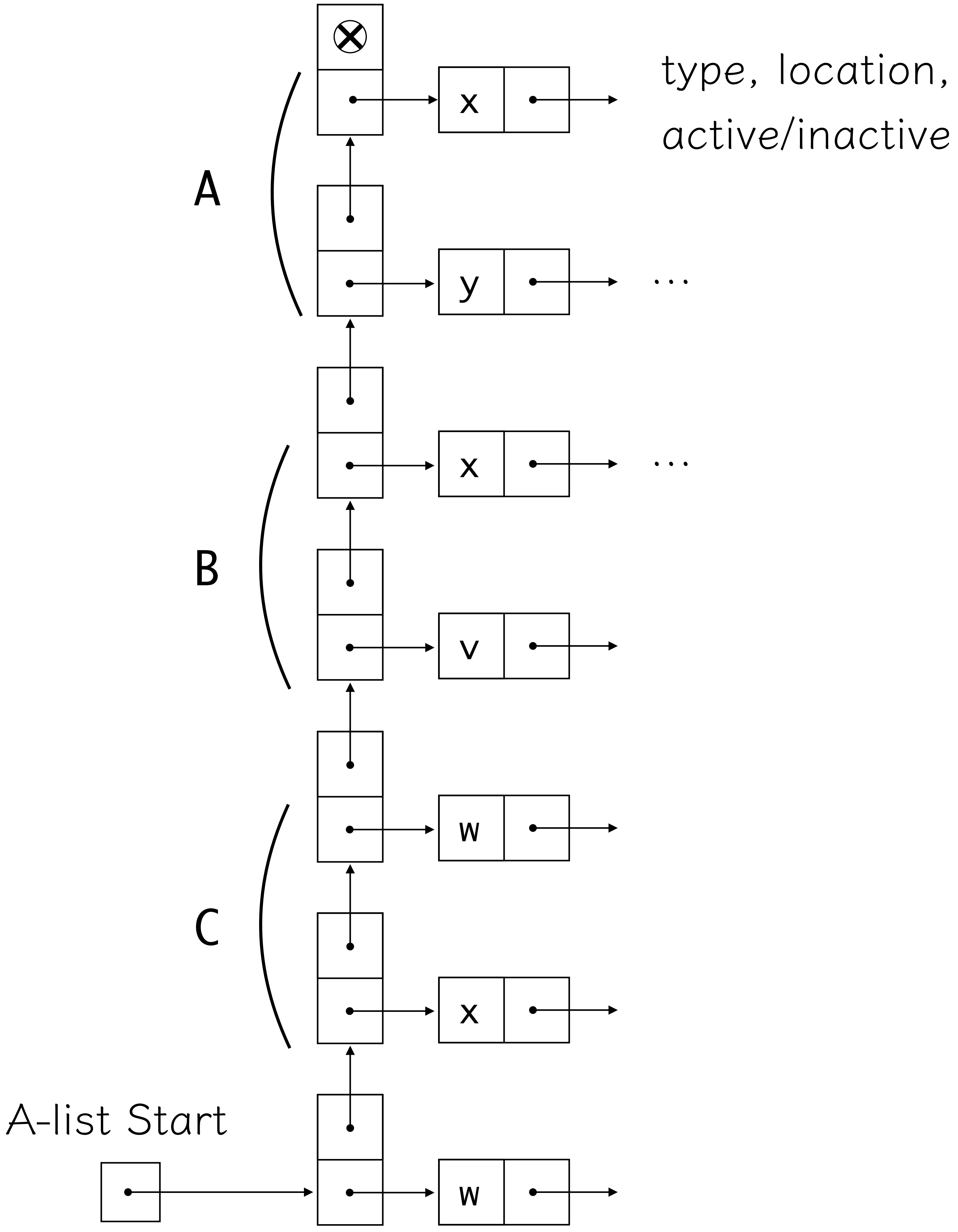
Association List

- Other than direct storage in activation records,
  - bindings can be stored separately in ***an association list (A-list)***.
- When a program enters a new environment,
  - bindings are inserted to A-list, and removed when the program exits the environment.



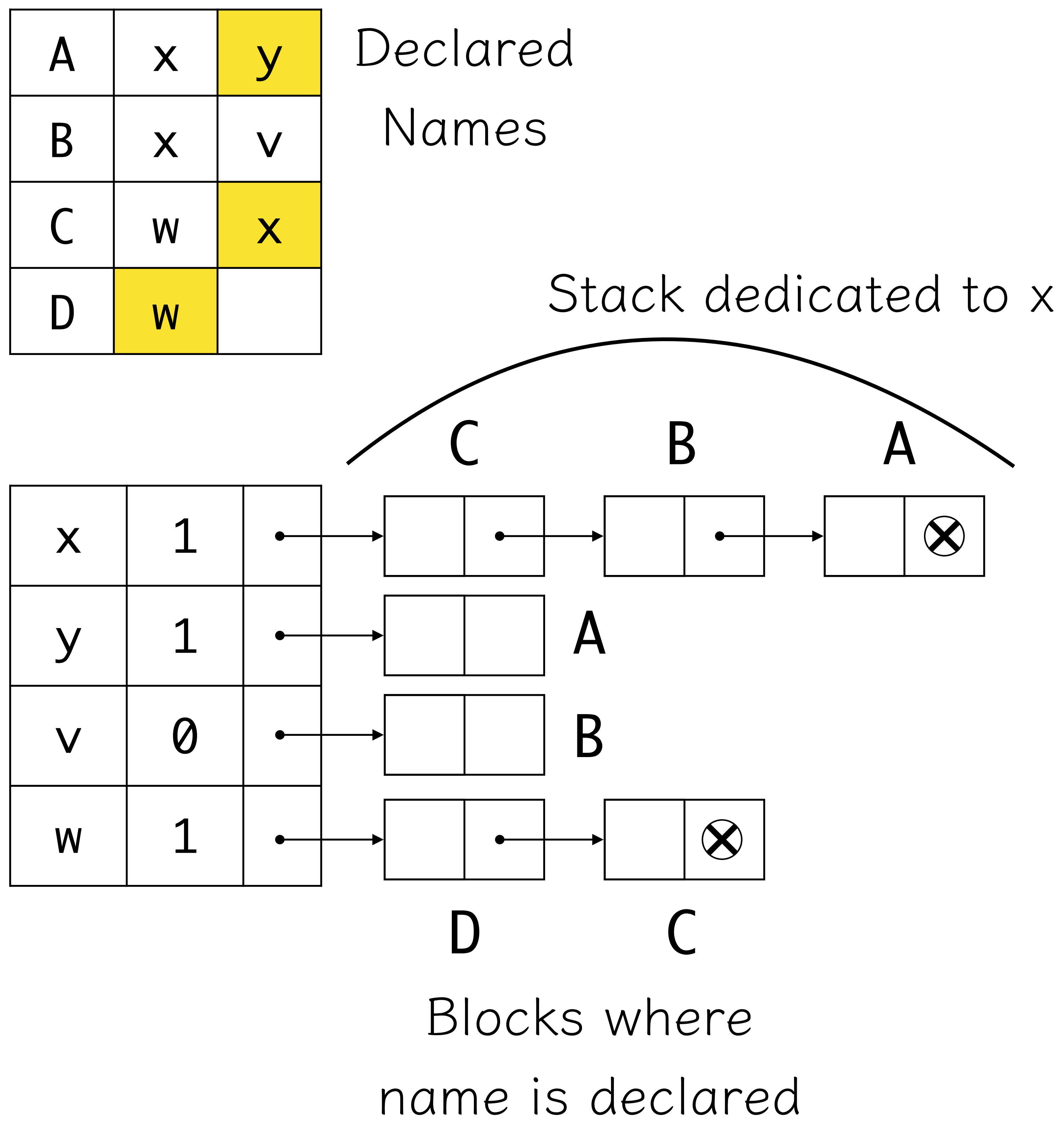
Association List

- Two disadvantages
  - *Names must be stored in structures at runtime.*
    - We cannot trace their locations at compile time.
  - *Runtime search of names is inefficient*.
    - We might need to check all the list.



Central Referencing Table

- To address the disadvantages, we can employ ***Central Referencing Table (CRT)***.
- All the names used in a program are stored in CRT.
- For each name, there is a flag indicating whether it is active / inactive.
- Dedicated stack contains the valid binding for the name at the top, and deactivated bindings under it.



Central Referencing Table

- We can also use a hidden stack to store all deactivated bindings.
- 3rd column contains the reference to the denotable object for the name.
- Deactivated bindings stored in the hidden stack, which will be restored when it becomes active again.

Declared Names

A	x	y
B	x	v
C	w	x
D	w	

reference to the object

A

x	1	a1
y	1	a2
v	0	
w	0	

AB

x	1	b1
y	1	a2
v	1	b2
w	0	

ABC

x	1	c1
y	1	a2
v	0	b2
w	1	c2

ABCD

x	1	c1
y	1	a2
v	0	b2
w	1	d1

hidden stack

x	a1

x	b1
x	a1

w	c2
x	b1
x	a1

Summary

- Stack and Heap
- Static Memory Management
- Dynamic Memory Management w/ Stack and Heap
- Scope Rule Implementation
  - Static: Static Link / Display
  - Dynamic: A-list / CRT