Note 7

Block Structures

Yunheung Paek
Associate Professor
Software Optimizations and Restructuring Lab.
Seoul National University

so&r

Topics

- Definition of a block
- Block structures
- Implementation of block structures

Blocks

- A block
  - is a section of code that consists of *a set of declarations* and *a sequence of statements*.
  - provides its own environment or scope for variables.
  - allocates storage to variables local to a block when execution enters the block; the storage is deallocated when the block is exited.
  - is delimited by keywords or special characters
    - *procedure bodies* — *ex: Fortran* → `function . . . end`
    - *begin/end* — *ex: Algol* → `begin . . end`
    - *special characters* — *ex: C* → `{ . . . }`
- The programming languages that allow programs to define blocks are called **block-structured** languages.
  - block-structured: Pascal, PL/I, Algol, C/C++, Scheme
  - non-block-structured: Cobol, Basic, Assembly

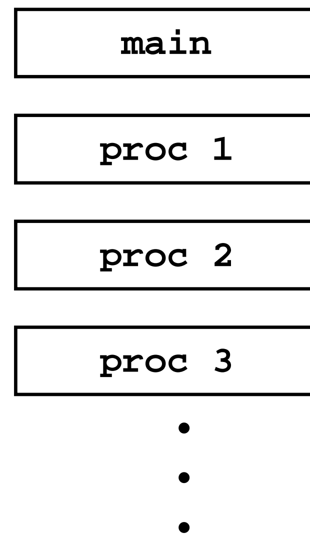
Terminology for blocks

- A block enclosed by other blocks is called a **nested** block. A block enclosing other blocks is called a **nesting** block.
- The variables **declared** (or **bound**) in a block are called **local** variables. Bindings of local variables of a block are visible only inside the block.
- The declarations of local variables in a block are **implicitly inherited** by nested blocks. It is not allowed to export a declaration to nesting blocks.
- **Non-local** variables of a block are those whose declarations are implicitly inherited from nesting blocks. They are not bound in the block, but bound in one of the nesting blocks.
- **Global** variables are those bound in the outermost nesting block; thereby, their bindings are visible in entire program, and they are accessible anywhere in a program.

Types of blocks

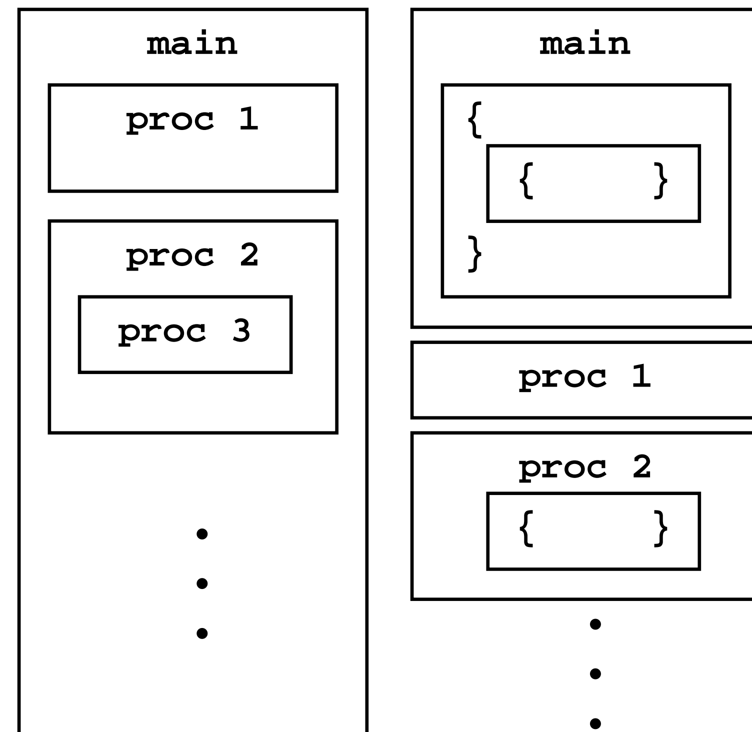
- Disjoint block structure
  - The body of a procedure is a block.
  - There is no nesting of blocks.

ex: Fortran

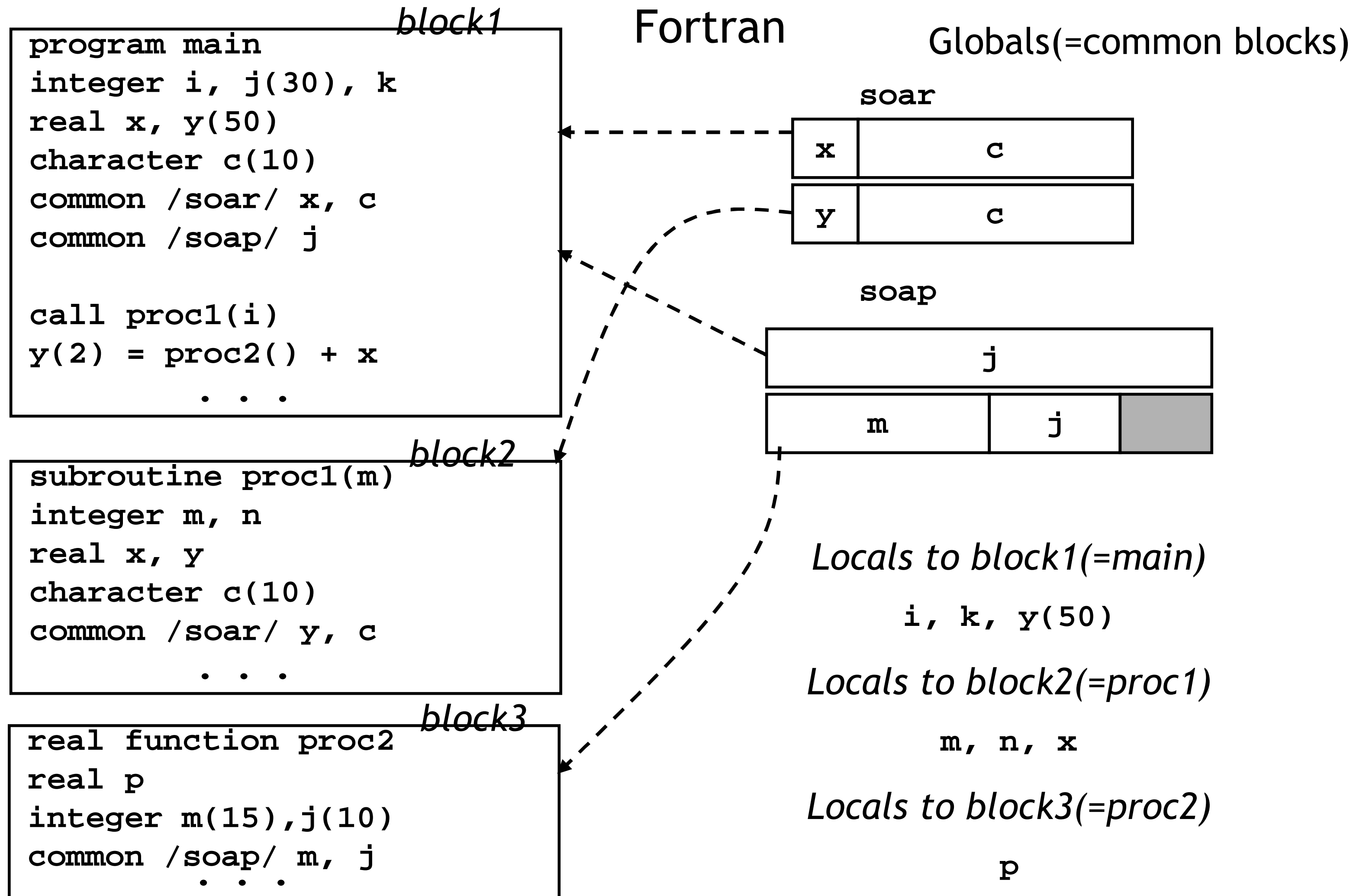


- Nested block structure
  - A block contains other blocks nested inside it.

ex: Pascal, Algol, C, Scheme

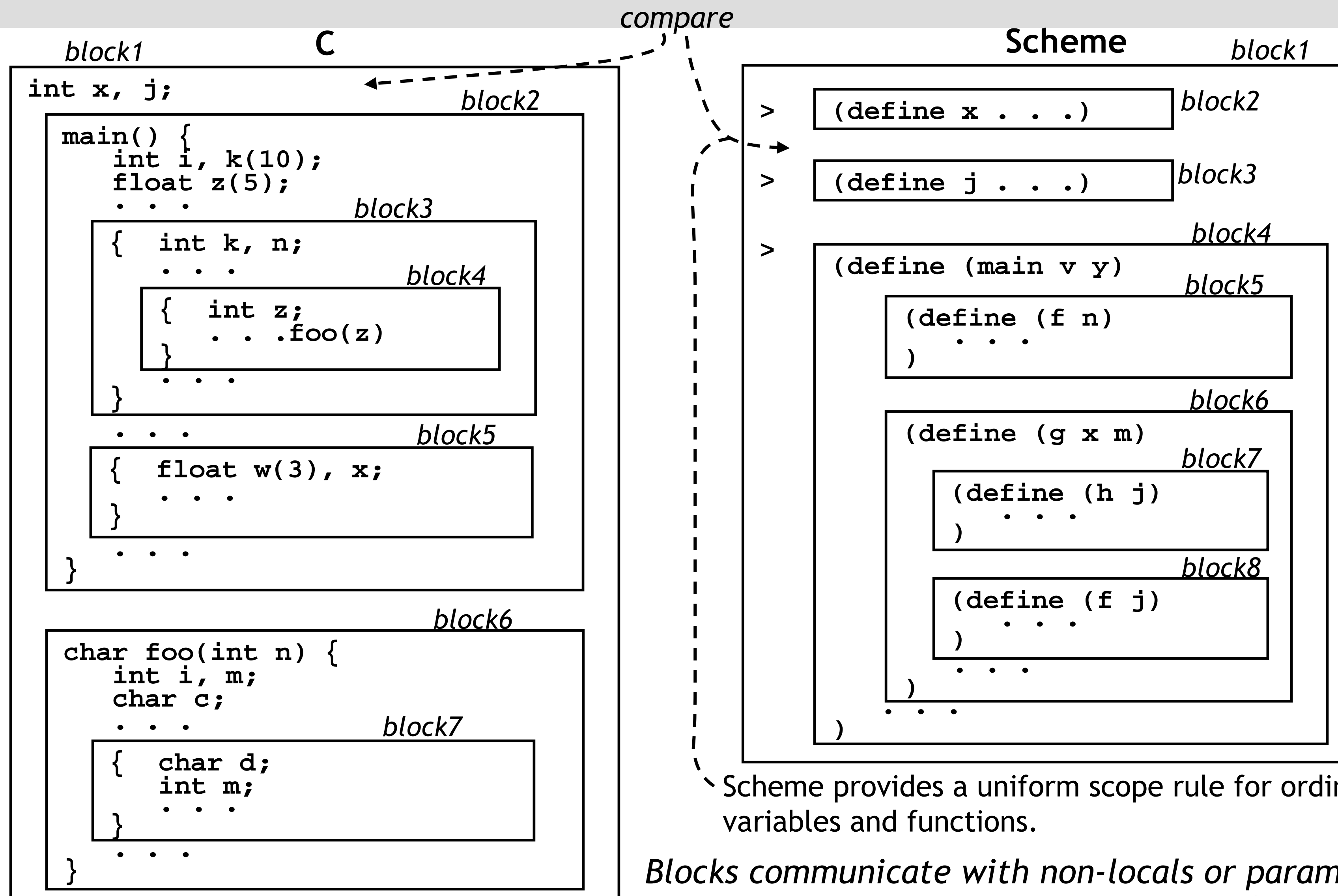


Disjoint block structure



Procedures communicate with common variables or parameters using call-by-reference

Nested block structure



Scheme provides a uniform scope rule for ordinary variables and functions.

Blocks communicate with non-locals or parameters.

Advantages of block structure

- The block structure improves *readability* of programming by delimiting the scope of a binding, whiling nested blocks allow some bindings to be shared.
  - → The storage location of shared bindings can be used for communication between different blocks.
- It saves *storage* because the binding of a variable needs to be remembered only as long as the innermost nesting block is executed.
  - → Upon return of a block, the storage for the local variables can be deallocated unless a variable is static.
- It provides a mechanism for structuring programs, which may improve *writability* of programming.
  - → For instance, a given task is decomposed to several subtasks. A main procedure performs the whole task by distributing the subtasks to its sub-procedures within it.

Problems w/ globals in block structure

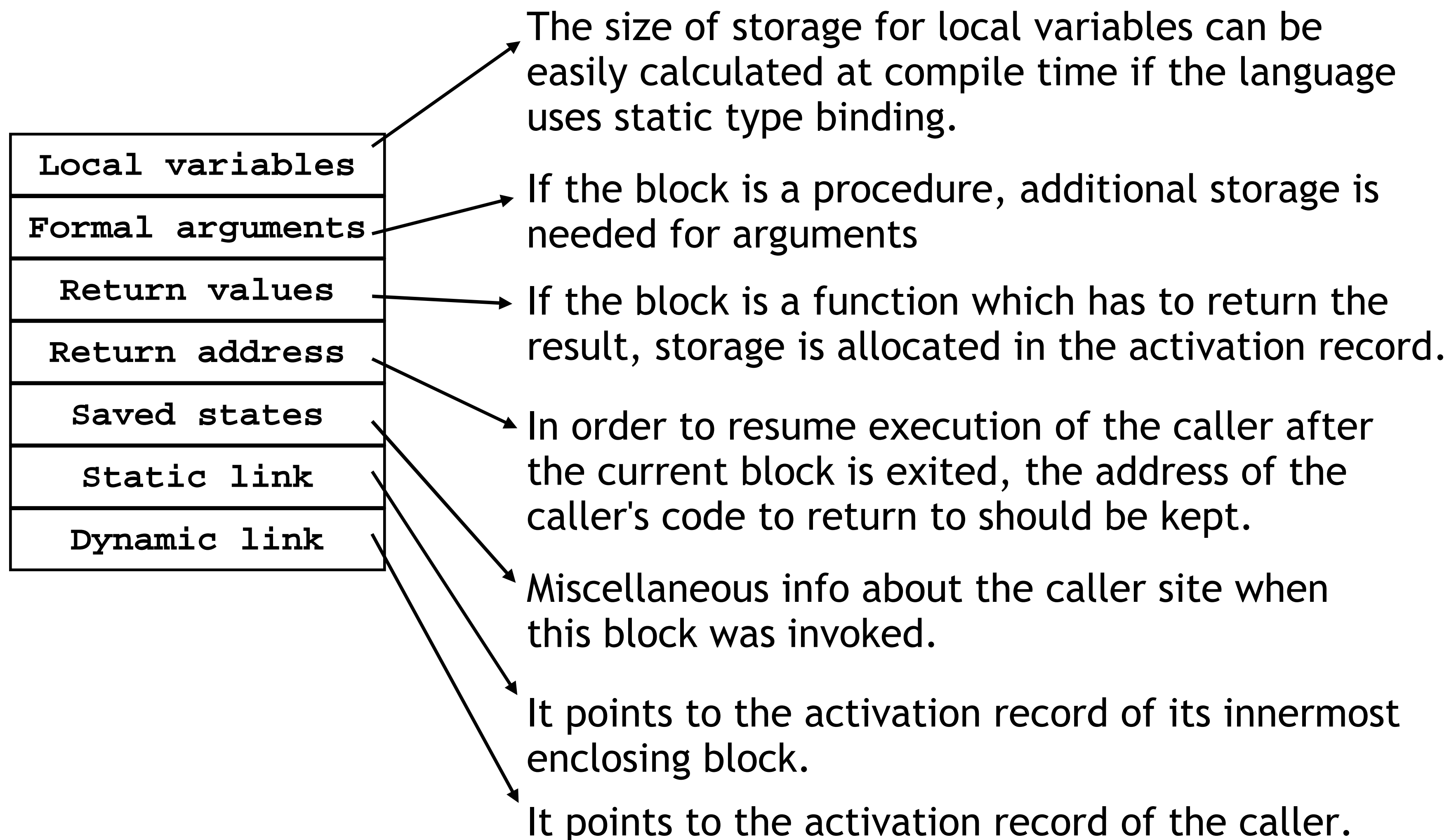
- It is generally difficult to exercise sharing bindings (or declarations) effectively.
- So, there is a tendency to move the declarations to the outermost block, which results in many global variables in a program. This exacerbates the following problems:
  - **Side-effects:** Debugging/maintaining programs are more difficult
  - **Indiscriminate accesses:** Due to implicit inheritance of bindings, all bindings in a block can be accessed by all nested blocks even when they are not supposed to. This results in less secure code.

    e.g.) typos in a nested block may not be recognized, and yet producing incorrect output.
  - **Screening problems:** The visibility of a declaration in a block can be accidentally lost when a variable with the same name is re-declared in an intervening nested block. This often happens when a program is large.

Implementation of block structure

- When a program block is invoked in a block-structured language, the body of the block is executed.
- Each execution of the body is called an **activation** of the block.
- Associated with each activation of a block is storage of the variables declared in the block and any additional information needed for the activation.
- The storage associated with an activation is called an **activation record** (AR).
- Components of an AR may vary depending on languages.

Activation record



Storage types for the implementation

- Static location
  - The addresses of static variables are fixed before run time.
  - Some storage is reserved for the variables at compile time.
- Stack
  - A stack is used to manage allocation/deallocation of ARs.
  - A language that holds ARs in a stack is said to obey a **stack-discipline**.

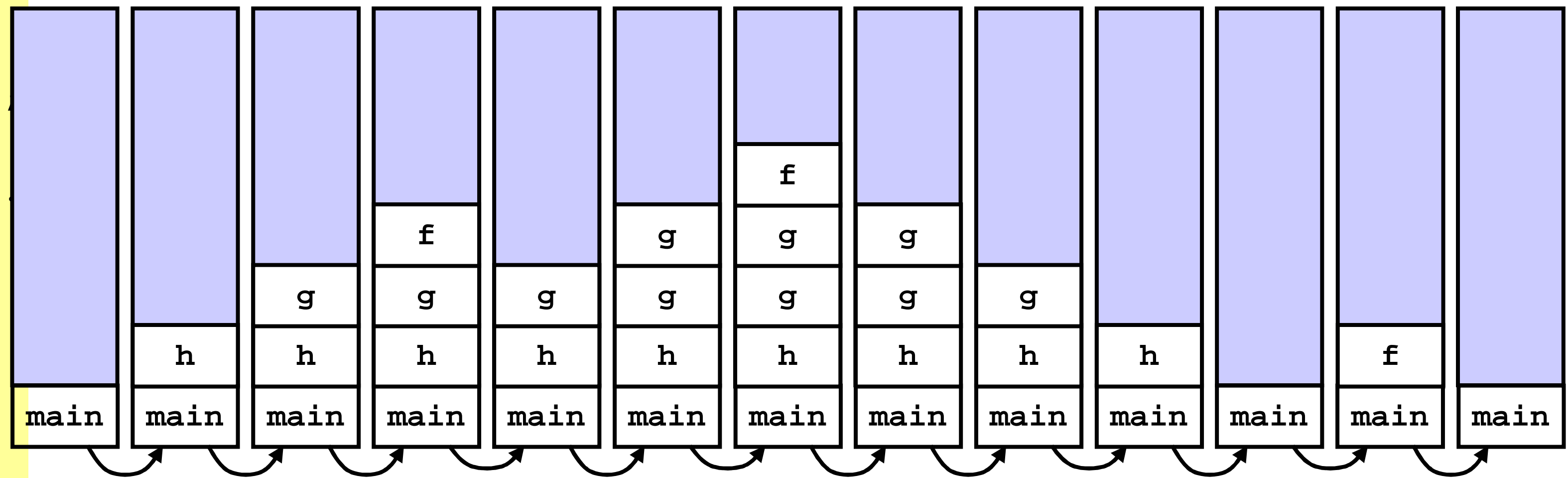
    → Most traditional imperative programming languages such as C and Pascal obey the discipline.
- Heap
  - A heap is more expensive but more flexible than a stack.

    → Typically, it is used for dynamic/pointer variables.
  - Functional languages use a heap for activation record allocation in order to treat functions/procedures as first-class citizens. (*why?*)
  - Also, some newer imperative programming languages such as Modula-3 and Oberon use a heap for AR allocation.

AR implemented in a stack

- Some observations on uses of ARs
  - Recursion has significant implications for language implementations of block structure. To support recursion, a separate AR has to be allocated for each procedure block invocation (*why?*)
  - When a block is exited, the life-time of the local variables ends. The AR is no longer needed after returning from the block.
- ARs can be efficiently managed with an **LIFO stack.**

```
f() { . . .}
g() {
  static int i=0
  ..f()..
  if(i++==0) g()
}
h() {
  ..g()..
}
main() {
  ..h()..f()..
}
```

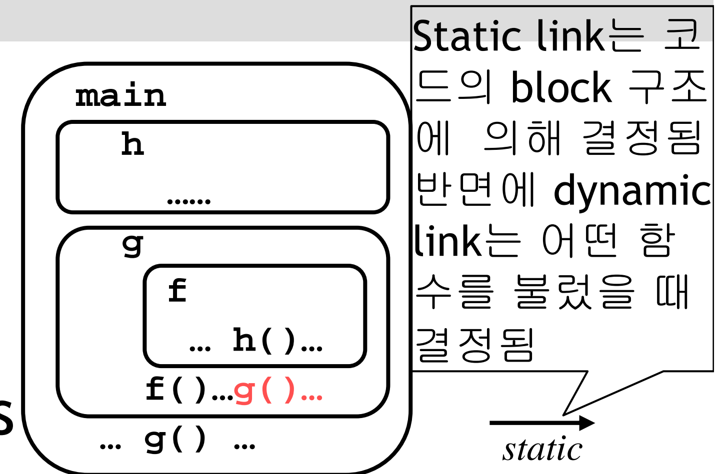


Static and dynamic links in a stack

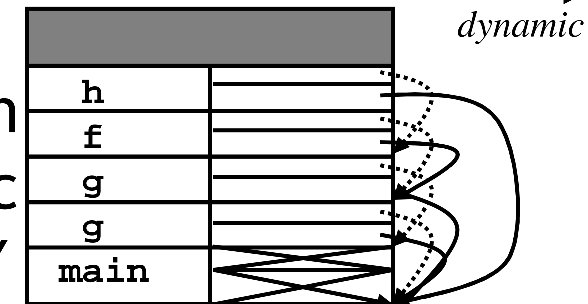
- A dynamic link is used to restore access to the AR where the current block is activated: that is, *the AR of the caller of the block*.
- A static link in an AR of a block points to the AR of the next nesting block.
- Assume that X is a block whose AR is currently on the top of the stack when a new block Y is invoked. The dynamic and static link values of a new AR of Y are: Dynamic = *address of the base of the AR of X*

  Static = { *address of the AR of d+1′ th outer nesting block of X if d ≥ 0*
  address of the base of the AR of X if d = −1

  where d = nesting level of X – nesting level of Y.



When h() is called twice



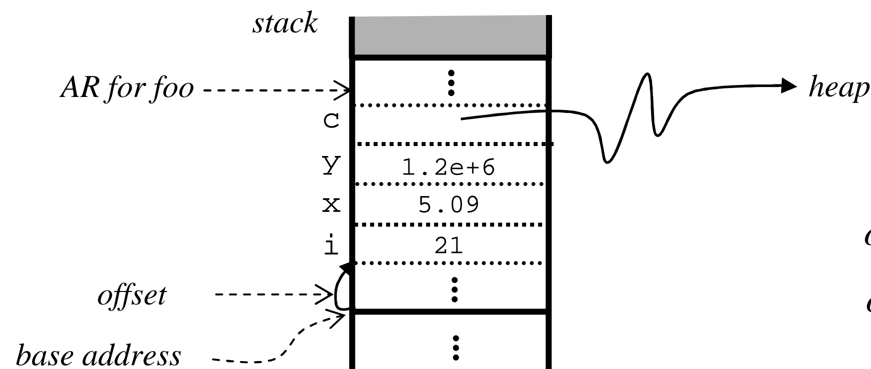
Note: in case of the C language, d = 0

Access to (non)-local data in a stack

- Local accesses are fast:

  address of a local variable = address of base of current AR + an offset

```
foo (int i) {
    double x, y;
    char* c;
    …
}
```



$offset(x) = offset(i) + 4$

$offset(y) = offset(x) + 8$

$offset(c) = offset(y) + 8$

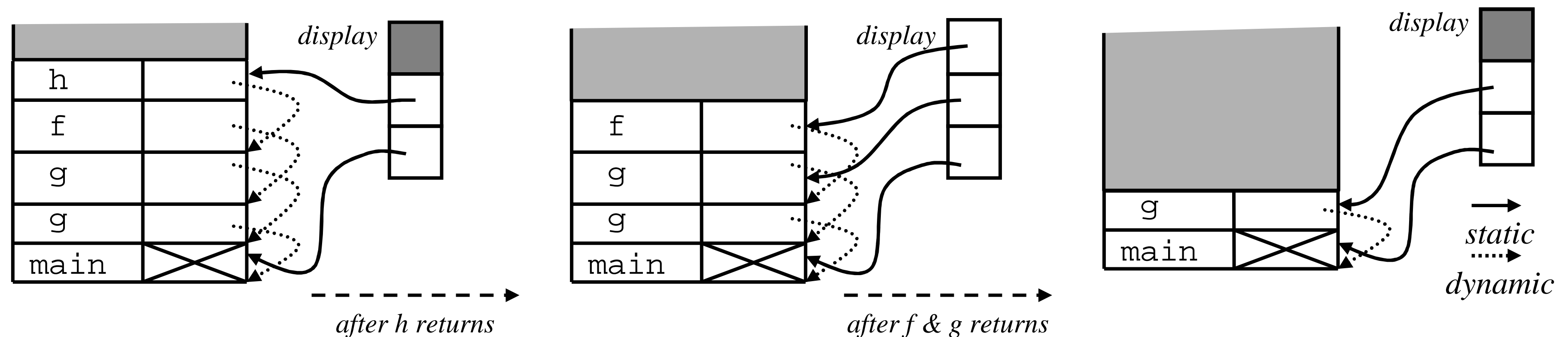
$\ldots = offset(c) + 4$

- Nonlocal accesses are slower because they require extra pointers chasing following static or dynamic links.
  - If **dynamic scoping** is used, follow the dynamic links until the nonlocal variable is found. → *static links can be removed from ARs*
  - If **static scoping** is used, follow the static links until the nonlocal variable is found. → *this is generally more efficient*

Displays

- The problem with static links: nonlocal accesses are costly when the nesting level of the current AR is deep because it should chase several links.

  → *Solution: use a* **display**, *a single array of static links.*

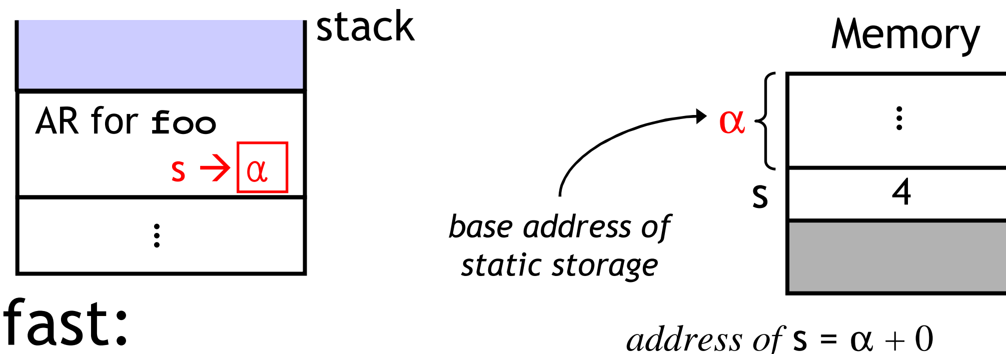


- Pros and cons of displays
  - The memory access time is equal for all nonlocal accesses; nonlocal accesses cost only one more memory access than local.
  - The current display must be updated at every block invocation and return.
  - Local accesses always need an extra step to read display.

Storage allocation for static variables

- A static variable declared in a block should retain its value between activations of the block.
  - If static variables are stored in ARs, this requirement cannot be met because the AR for each activation is removed after the activation is killed and, thereby, the values of all the variables in the AR is lost.
  - One solution is to store static variables in separate memory space with fixed addresses. For this, the compiler reserves some static storage space for static variables when it compiles the program.

```
foo (int i) {
    static int s = 0;
    …
    s++;
    …
}
```



- Access to static data is fast:

  address of static data = base address of static storage + offset

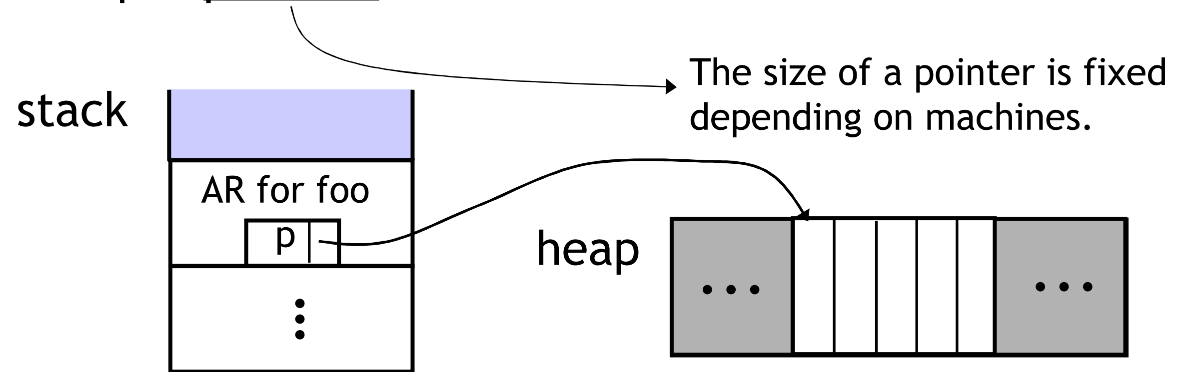
  → the base address and offsets can be determined at compile-time.

Heap allocation/deallocation

- If ARs are managed with a heap(*the area of memory used for dynamic memory allocation*), life times of the ARs need not be tied to the LIFO flow of control between activation.
- Even after control returns from a procedure block, an AR for the block can stay in storage. That is, the local variables are bound as long as needed.
  - This is useful for functional languages that provide **thunks**.
  - Even in imperative languages, the size of an AR may not be determined when the AR is created because of **dynamic arrays**.
  - So, languages that use a stack for AR allocation still need a heap to allocate dynamic structures and to put pointers to them in the AR.

Related to delay or force in Scheme

```
foo() {
   int* p;
    …
   p = new int[5];
    …
}
```



Allocation of dynamic arrays/lists

- Most languages support dynamic allocation primitives.

Pascal

```
type item = ^list;
     list = record
                 head : integer;
                 tail : item
               end;
var p : item;
begin
   new(p);
   p^.head = 3;
   p^.tail = nil;                    // p = {3}
```

C++

```
list* p = new list;
p->head = 3;
p->tail = '\0';
```

Ada

```
item p = new list(3, nil);
```

Scheme

```
(define p (cons 3 '()))
```

CLU

```
p : array[int] := {}
array[int]$addh(p, 3)                // p = {3}
```

→ The primitives allocate storage for a list/struct/record on a heap and store a pointer to it in `p` that is located in the AR on a stack.

How to deallocate dynamic data?

- implicitly at every block exit

```
f() { int* q = new int[10]; ... return q; }
g() { ... int* p = f(); ... }
```

→ *What would p point to if q is deallocated when f returns? Nothing*

- implicitly at program termination

```
h() { ... int* p = f(); ... p = f(); ... }
```

→ *If p is not deallocated before the second call to f, memory leak occurs due to* **garbage**.

- use deallocation primitives: **`dispose`** (Pascal), **`delete`** (C++)

```
h() { ... int* p = f(); ... delete p; p = f(); … }
```

→ *versatile and flexible, but more difficult and less secure because the user must deallocate dynamic arrays explicitly.*

- use garbage collection: Ada, CLU, Scheme, Emacs (Lisp)
  - A background process monitors all the objects in the heap and deallocate garbage if it is found.

→ *more secure and convenient but expensive because of run-time overhead*

Common errors w/ dynamic allocation

- Explicit deallocation may cause **dangling pointers**.

```
void f () {
  char* d;
  char* c = d = "this is a list";
  delete c;
  ...
  cout << d;
}
```

//Error! The string may no longer exist

- Mixing stack-allocated variables and pointers may cause errors.

```
float* g() {
  float* s = new float;
  float t;
  ...
  return &t;
}
void h() {
  float* r = g();
  ...
}
```

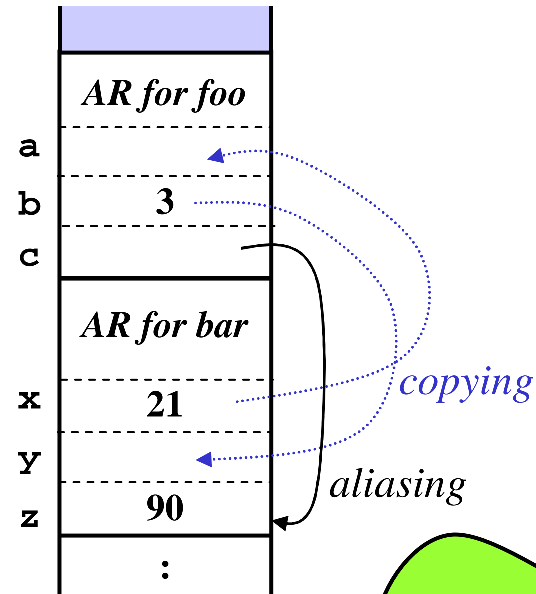
// s is garbage if it is not explicitly deallocated in g

// no syntax error but r is a dangling pointer

Parameter passing in block structure

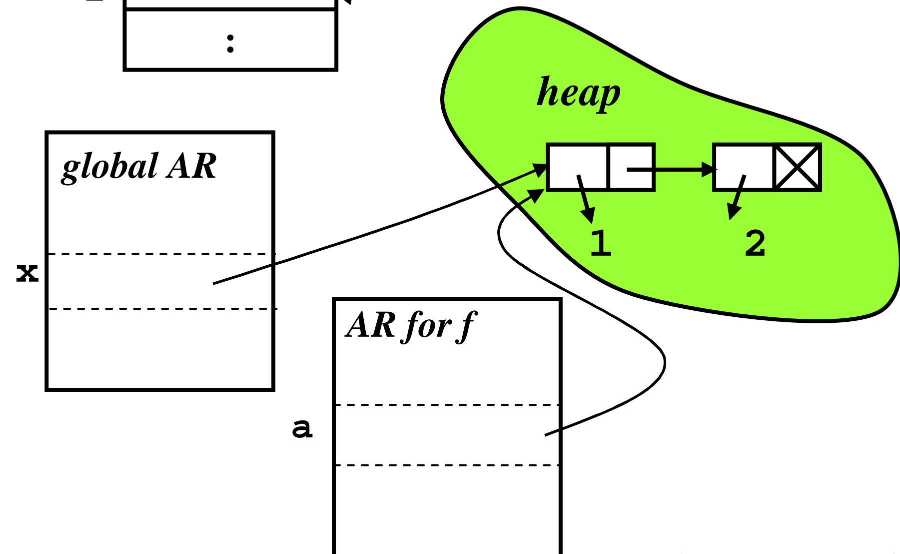
- Implementing call-by-value/result/reference

```
foo(value a, result b, ref c) {
      ...
}
bar() {
      ... foo(x,y,z) ...
}
```



- Implementing call-by-sharing

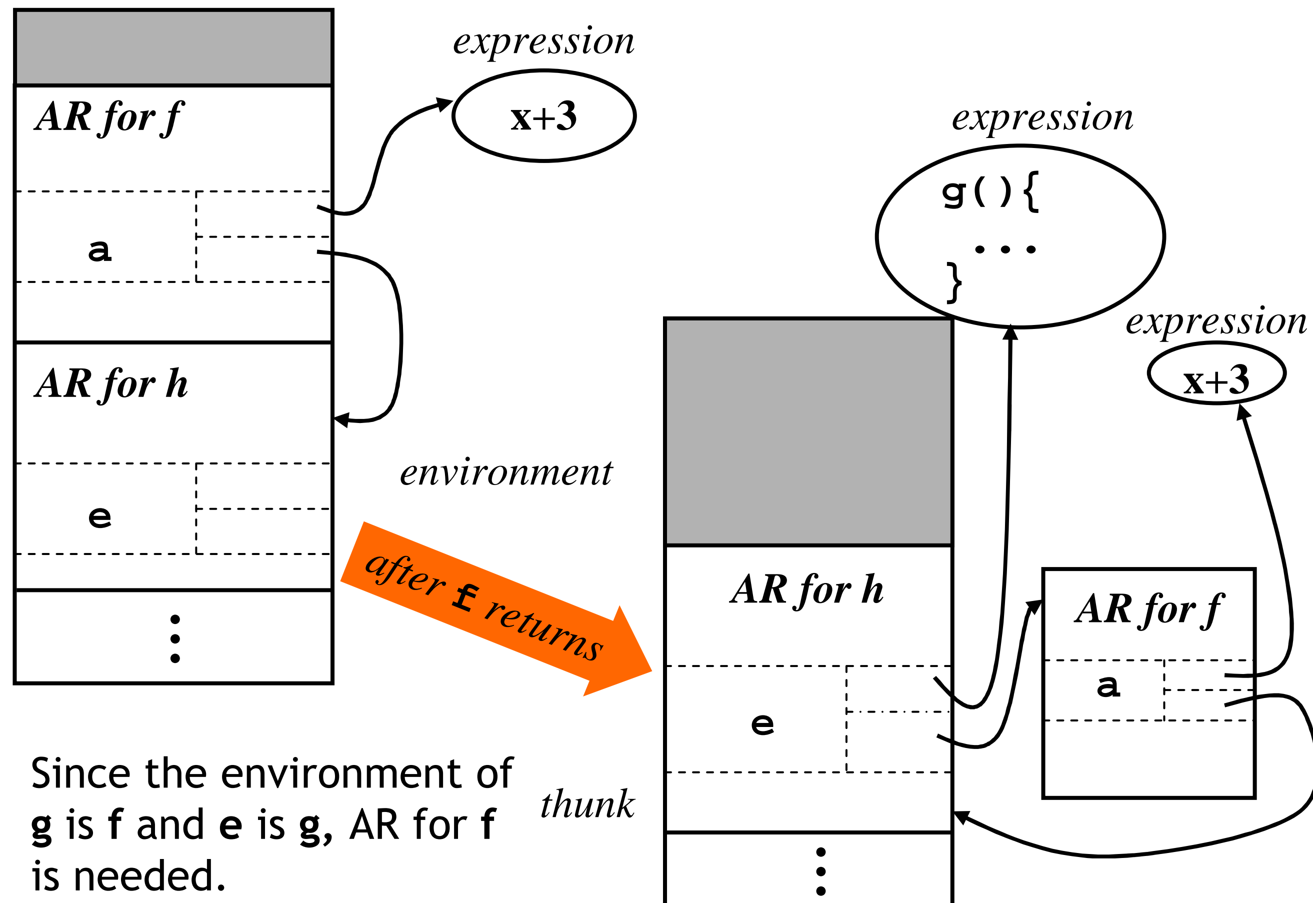
```
> (define x '(1 2))
> (define f (lambda (a) ...))
> (f x)
...
```



Parameter passing in block structure

- Implementing thunks as arguments (call-by-name/functions as results)

```
f(name a) {
    g() {
        ...
    }
    ...
    return g;
}
h(){
    ...
    e = f(x+3);
    ...
}
```



Since the environment of **g** is **f** and **e** is **g**, AR for **f** is needed.