Programming Methodology

Spring 2009

Blocks

Topics

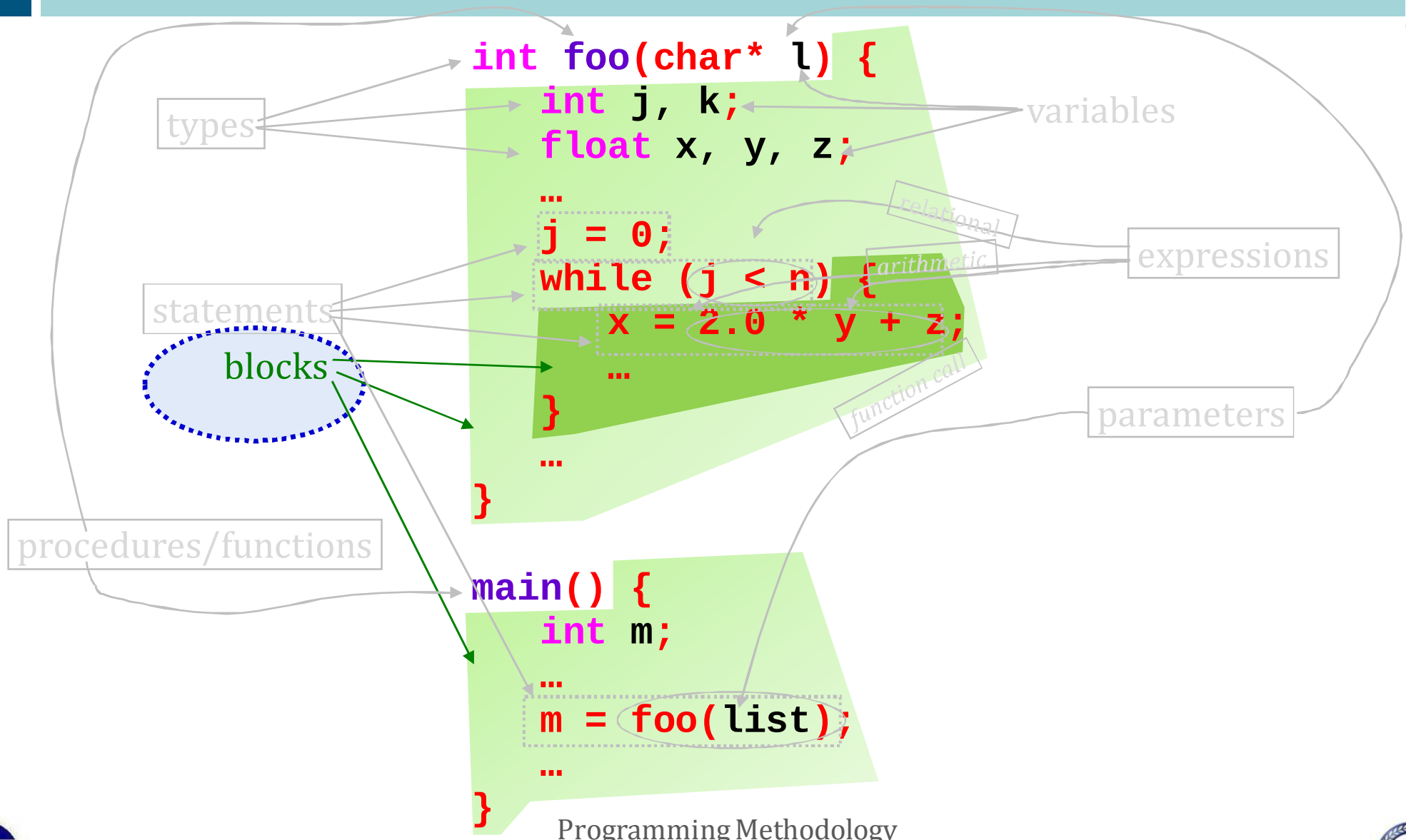
Definition of a block

Block structures

Implementation of block structures

- **Activation Records**
- **Runtime stacks**
- **Heaps**
- **Parameter passinng**

Programming language constructs



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

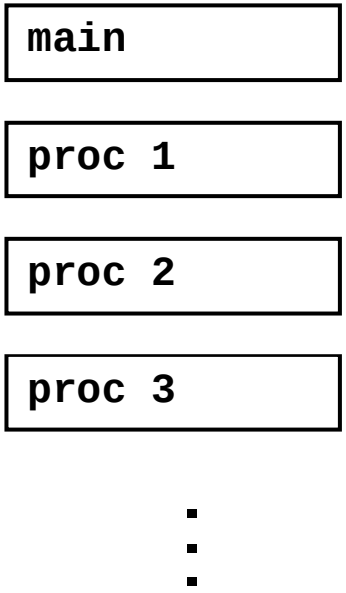
Terminology for blocks

- A block enclosed by other blocks is called a **nested** block.
- A block enclosing other blocks is called a **nesting** block.
- **Local** variables are the variables **declared/bound** in a block.
  - ➔ 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 bound in the outermost nesting block.
  - ➔ Their bindings are visible in entire program, and they are accessible anywhere in a program.

Types of blocks

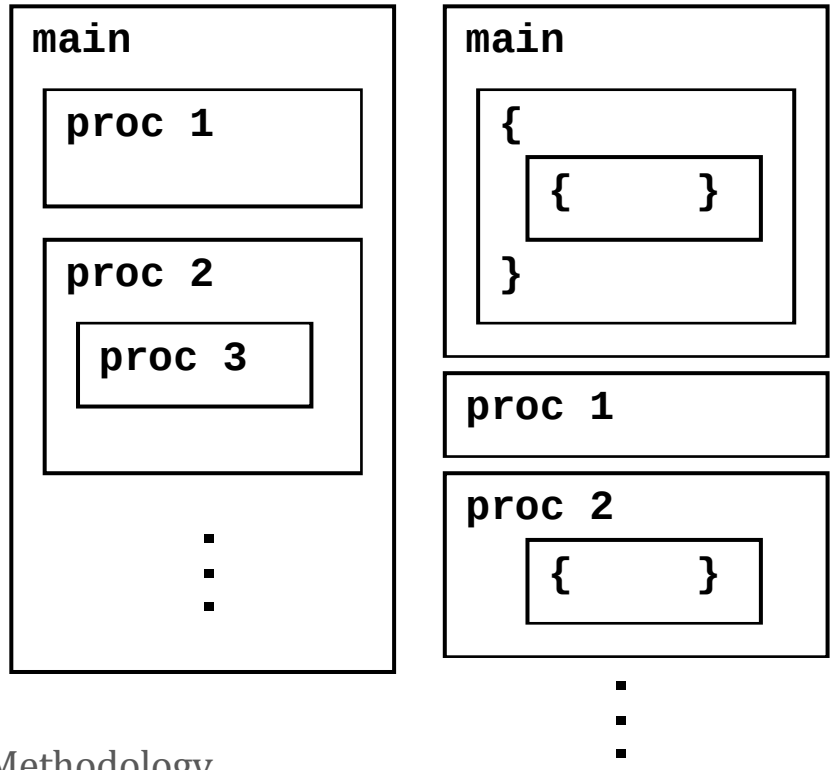
- Disjoint block structure
  - The body of a procedure is a block.
  - There is no nesting of blocks.

ex: Fortran



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

ex: Pascal, Algol, C, Scheme



Disjoint block structure (Fortran)


```
program main
integer i, j(30), k
real x, y(50)
character c(10)
common /soar/ x, c
common /soap/ j

call proc1(i)
y(2) = proc2() + x
        . . .
```

block1

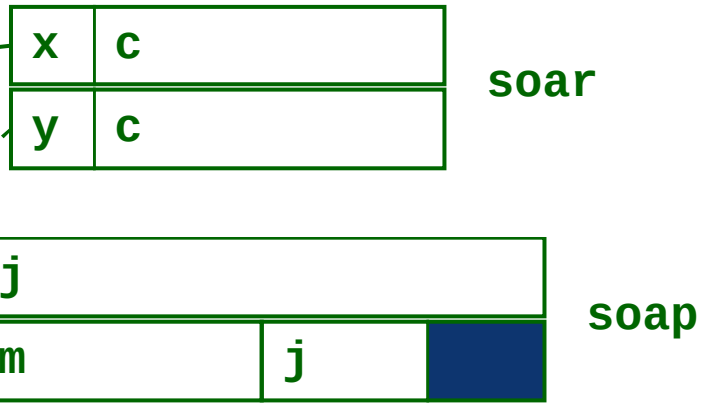
```
subroutine proc1(m)
integer m, n
real x, y
character c(10)
common /soar/ y, c
        . . .
```

block2

```
real function proc2
real p
integer m(15),j(10)
common /soap/ m, j
        . . .
```

block3

Globals(=common blocks)



Locals to block1(=main)
i, k, y(50)

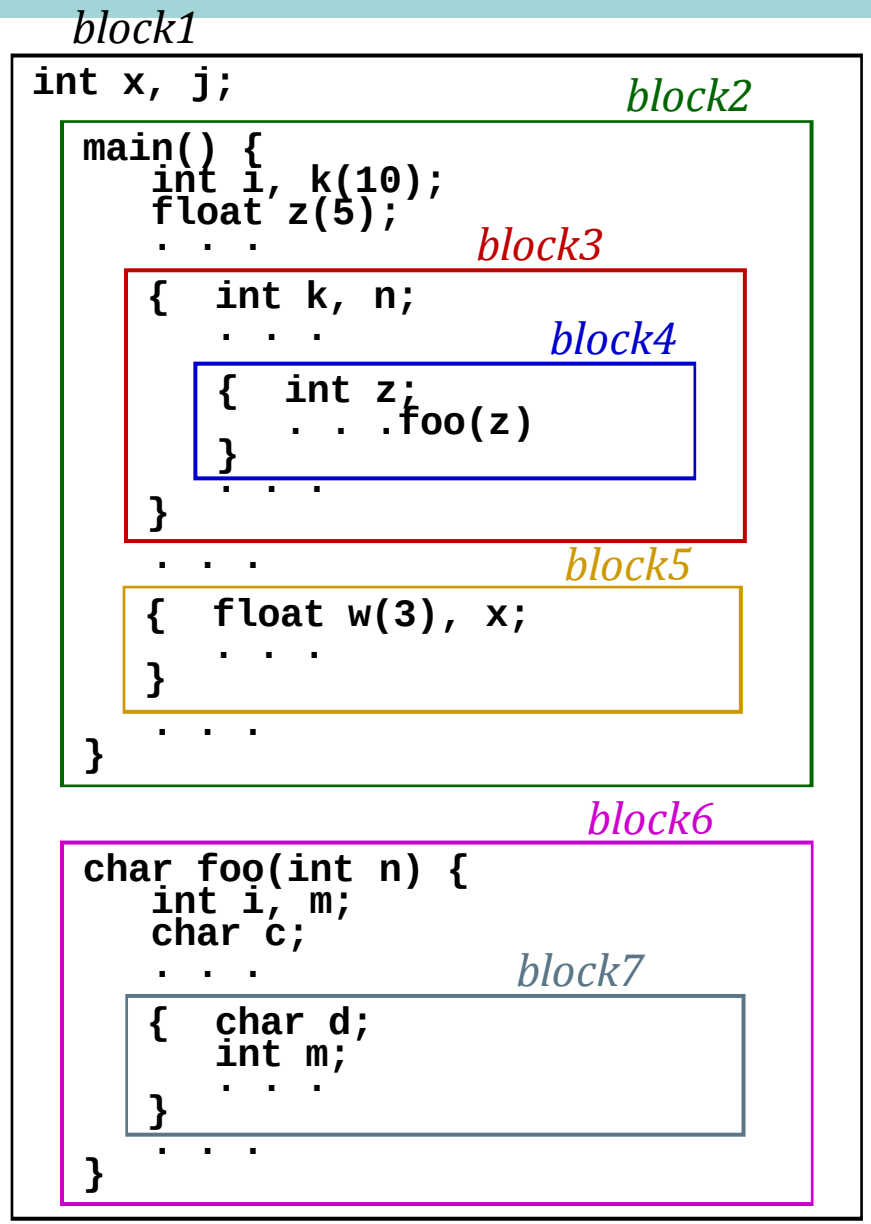
Locals to block2(=proc1)
m, n, x

Locals to block3(=proc2)
p

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

Nested block structure (C/C++)

- *Blocks communicate with non-locals or parameters.*



Advantages of block structure

- The block structure 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.
- It improves program *readability* by delimiting the scope of a binding, while 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* since we need remember the binding of a variable only when 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.

Problems with global 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
  - Indiscriminate accesses
  - Screening problems

Problems with global in block structure

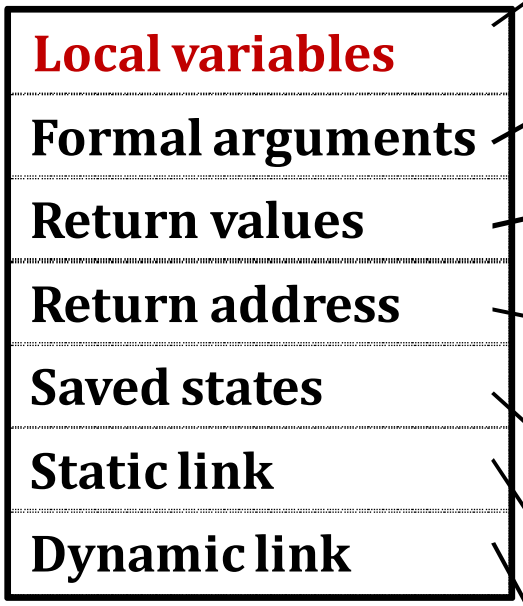
- Side-effects
  - Debugging/maintaining programs are more difficult
- Indiscriminate accesses
  - Due to implicit inheritance of bindings, all bindings in a block can be accessed by all nested blocks even when they are not supposed to.
  - This results in less secure code.
  - e.g.) typos in a nested 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) or a **frame**.
  - AR is defined for each function invocation at run time.
  - AR represents execution environment of the function.
  - AR includes local variables, parameters, return value, etc.

    ➔ Its component may vary depending on languages.

AR(activation record)/Frame



The size of storage for locals can be easily calculated at compile time if the language uses static type binding.

If the block is a procedure, additional storage is needed for arguments

If the block is a function which has to return the result, storage is allocated in the activation record.

In order to resume execution of the caller after the current block is exited, the address of the caller's code to return to should be kept.

Miscellaneous info about the caller site when this block was invoked.

It points to the activation record of its innermost enclosing block.

It points to the activation record of the caller.

Local variables in AR

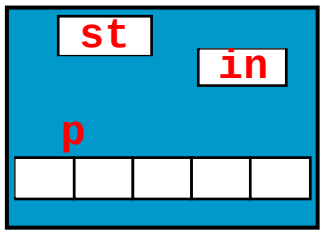
- In C/C++ code, variables refer to abstract memory locations.
- In assembly, they refer to actual locations in physical memory
  - The compiler usually divides physical memory into multiple regions.
  - Variables are stored to different regions according to their **storage types**.
  - So, how to access these variables in different regions must be specified.
- Ex:

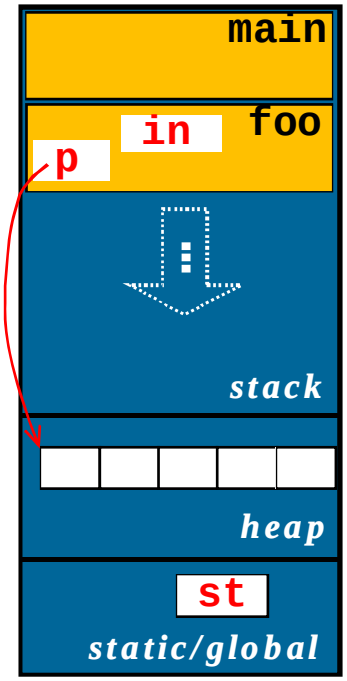
```
int foo(int* p) {
 int in;
 static int st;
 p = new int [5];
  …
 st = st - n[in];
  …
}

void main() {
  … foo(…) …
}
```

assembly

```
(allocate p[5])
 …
load r0,⟨in⟩
mult r1,r0,4
load r0,⟨p⟩
add r0,r1,r0
load r1,⟨r0⟩
load r0,⟨st⟩
sub r0,r0,r1
str ⟨st⟩,r0
 …
```





Storage types for the implementation

- Inside AR ← automatic variables: `int in;`
  - Local variables in a function store their r-values in the AR for the function.
  - In many languages, ARs are managed in a **stack** at run time.
    - When a function is invoked, its AR is created and pushed into the stack.
    - When the current function exits, its AR is popped and deleted.
  - So, local variables with their r-values in a stack are called **stack variables**.
- Static location ← static variables: `static int st;`
  - The addresses of static variables are fixed before run time since their life times persist during the whole execution of the code.
  - So, some storage is reserved for r-values of these variables at compile time.
- Heap ← dynamic variables: `int* p = new int [5];`
  - Typically, a heap is used for dynamic/pointer variables.
  - It is very flexible but expensive to maintain.

Run-time stack

- Languages that hold ARs in a stack are said to a **stack-discipline**.
  - ➔ Most existing languages like C/C++ obey the discipline.
- Operations on the ARs in the stack
  - Push the AR of function **f** on the stack when **f** is called.
  - Pop the AR when **f** returns.
  - Top AR = the AR of currently executed function
- This stack mechanism is necessary to support *recursion.*
  - 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
  - With the LIFO stack implementation, ARs can be created and stacked up for different activations of the same recursive function.

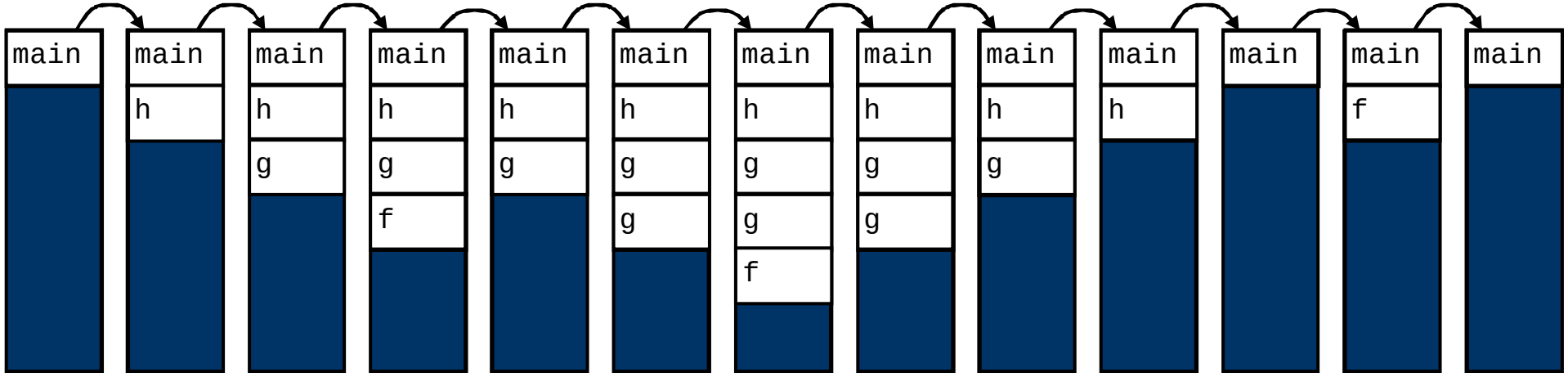
Stack operations with ARs

- Example:

```
f() { … }
g() {  … f() … g() …  }
h() {  … g() … }
main() {  … h() … f() …  }
```

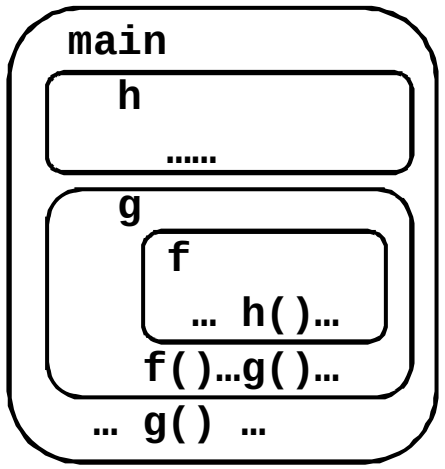
- ARs can be efficiently managed with an **LIFO stack.**
- Stack AR configurations at run time

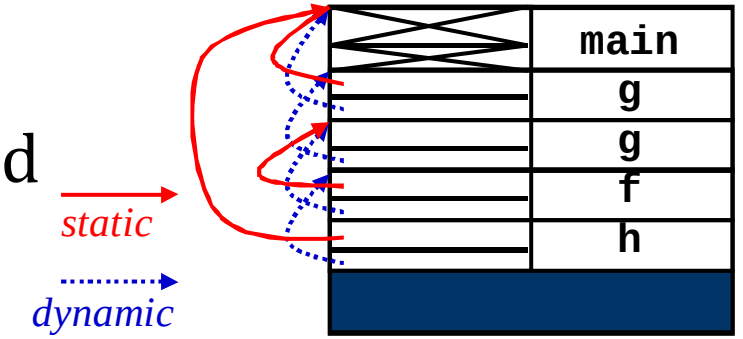
  ➔ Usually the stack grows downwards from high to low address.



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 AP of* X

Static = $\begin{cases} \textit{address of the AR of } d+1\textit{'th outer nesting block of X if } d \geq 0 \\ \textit{address of the base of the AP of X if } d = -1 \end{cases}$

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

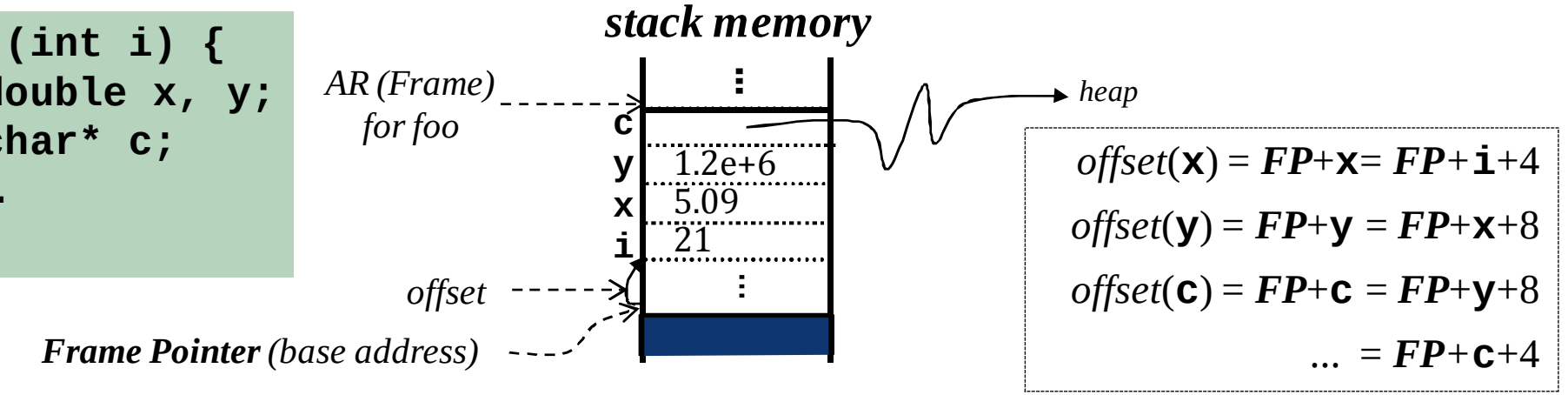
Note: in case of the C language, d = 0

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;
   …
}
```

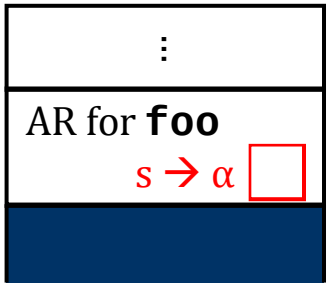


- Nonlocal accesses are slower because they require extra pointers chasing following static or dynamic links.
  - If **dynamic scoping** is used, follow the dynamic links until the nonlocal variable is found. → *static links can be removed from ARs*
  - If **static scoping** is used, follow the static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

Storage allocation for static variables

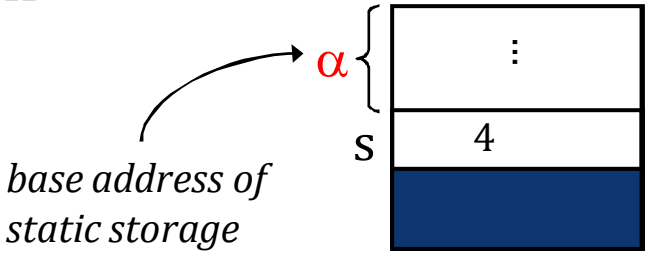
- A static variable declared in a block should retain its value between activations of the block.
  - If static variables are stored in ARs, this requirement cannot be met because the AR for each activation is removed after the activation is killed and, thereby, the values of all the variables in the 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++;
}
```



stack

Memory



address of s = α + 0

- Access to static data is fast:

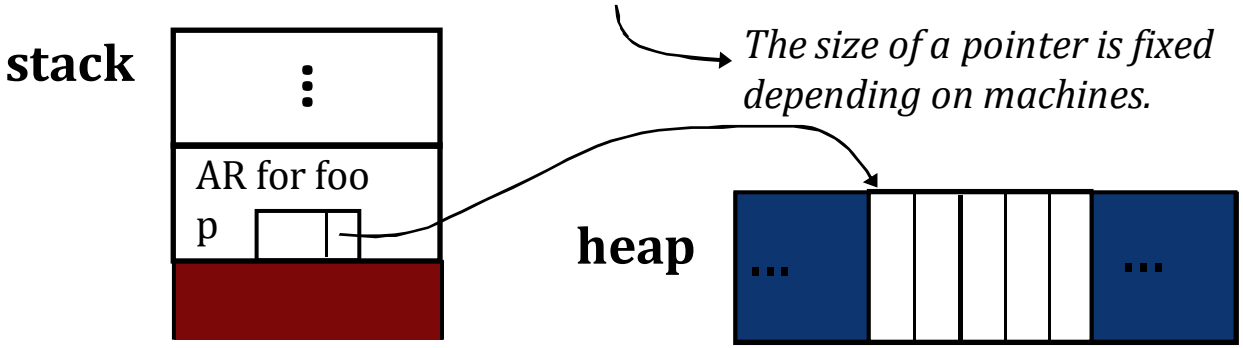
address of static data = base address of static storage + offset

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

Heap allocation/deallocation

- If ARs are managed with a heap, 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. ➔ *So, local variables are bound as long as needed.*
  - This is useful for functional languages that provide **thunks** for a function returned as a result.
  - 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.

```
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 '()))
```

- The primitives allocate storage for a 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?*

- 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: Java, 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* c = d = "this is a list";
  delete c;
  ...
  cout << d;        //Error! The string may no longer exist
}
```

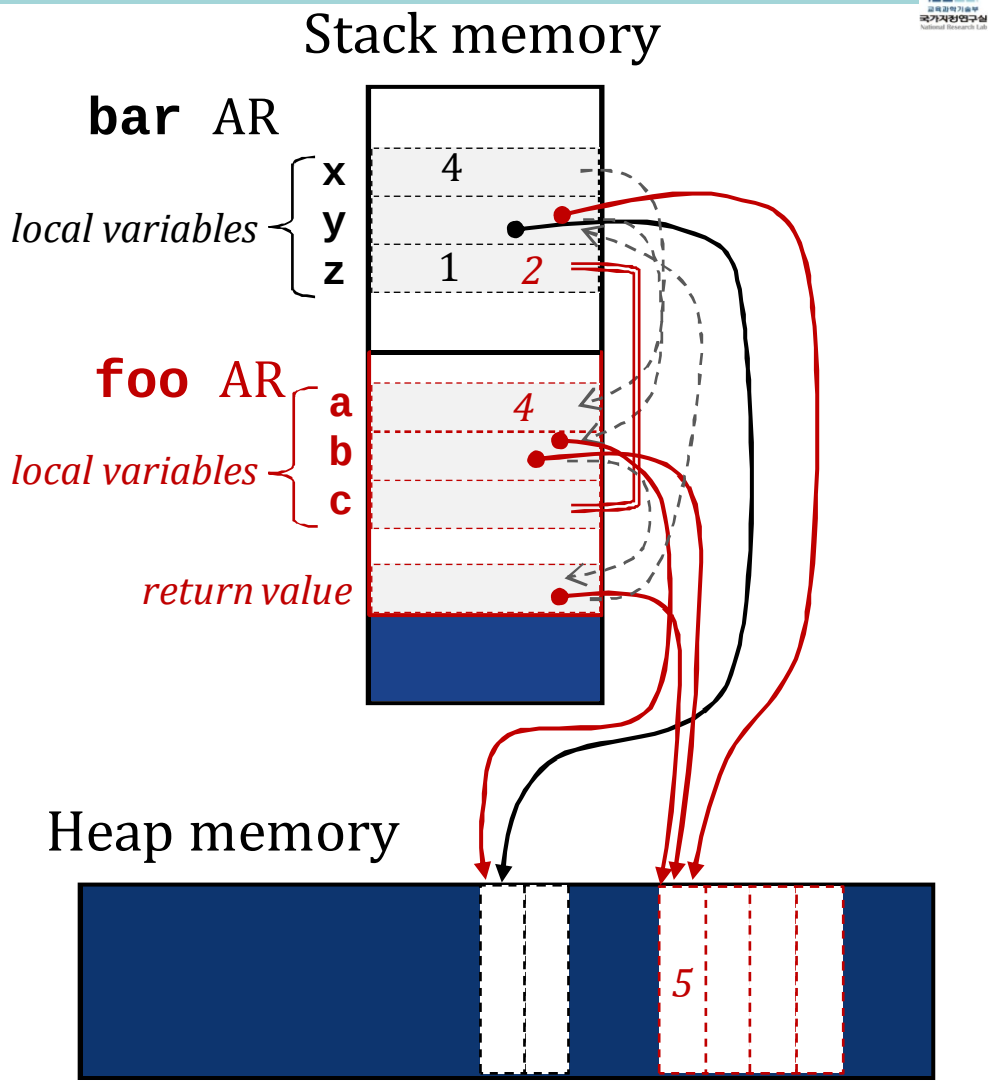
- Mixing stack-allocated variables and pointers may cause errors

```
float* g() {
  float* s = new float;
  float t;
  ...
  return &t;
}                   // s is garbage if it is not explicitly deallocated in g
void h() {
  float* r = g();   // no syntax error but r is a dangling pointer
  ...
}
```

Parameter passing in block structure

- Implementing call-by-value/reference

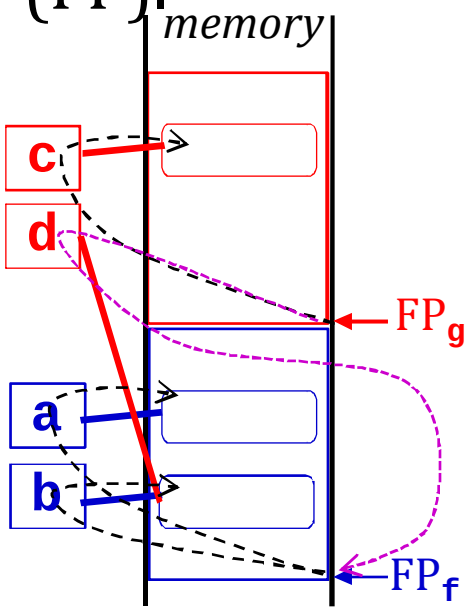
```
int* foo(int a, int* b, int& c)
{
        …
        c = c * 2;
        delete b;
        b = new int[4];
        …
        b[0] = a + 1;
        return b;
}
void bar()
{
        int x = 4;
        int *y = new int[2];
        int z = 1;
        y = foo(x, y, z);
        …
}
```



Recall this slide from Note 4!

- Call-by-reference causes *aliasing*, which makes the code …
  - generally more efficient (ex: long arrays); but
  - error prone due to *side effects*, and
  - in some cases, even less efficient because call-by-reference is often implemented with an extra level of indirection thru a Frame Pointer (FP).

```
g(int c, int& d) {
   … = c + d
   …
}
f() {
  int a, b;
  … g(a, b);
  …
}
```




```
…
load r1, [FPg+<c>]    ← c
load r2, [FPg+<FPf>] ← FPf
load r3, [r2+<b>]     ← d = b
add r4, r1, r3        ← c + d
…
```

```
…
load r14,[FPf+<a>]    ← a
load r15,[FPf+<b>]    ← b
call g
…
```