CS 335: Runtime Environments

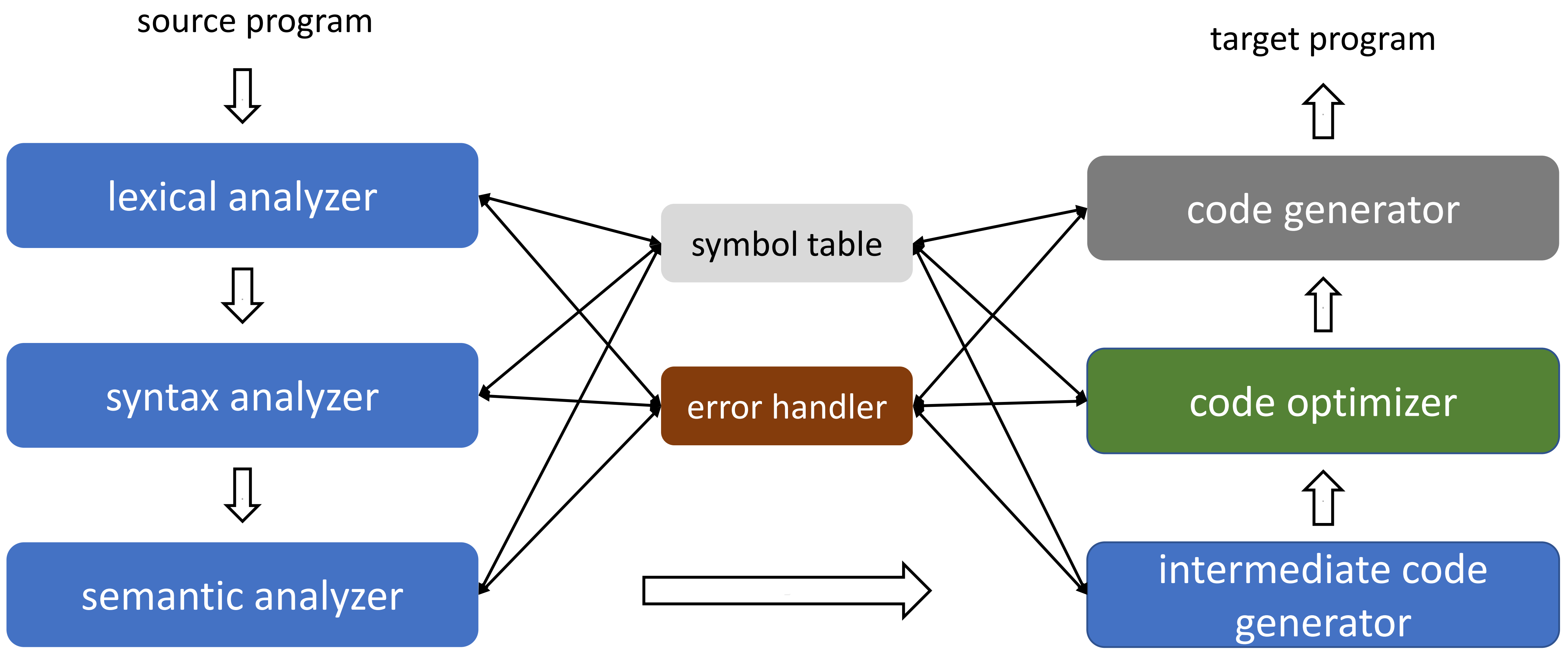
Swarnendu Biswas

Semester 2019-2020-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



Abstraction Spectrum

- Translating source code requires dealing with all programming language abstractions
  - For example, names, procedures, objects, flow of control, and exceptions
- Physical computer operates in terms of several primitive operations
  - Arithmetic, data movement, and control jumps
- It is not enough to just translate intermediate code to machine code
  - For e.g., need to manage memory when a program is executing

Runtime Environment

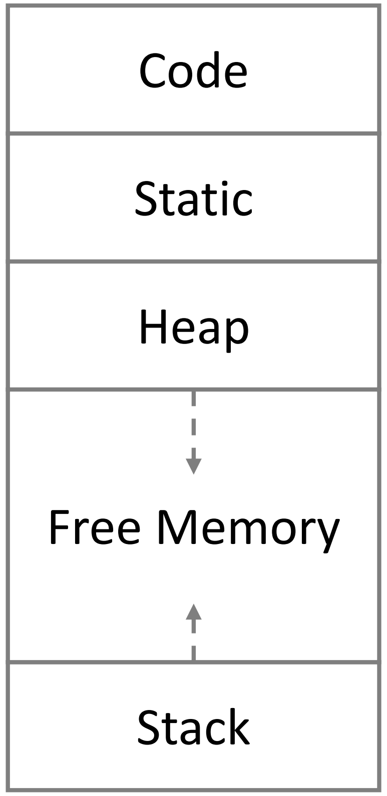
- A runtime environment is a **set of data structures** maintained at run time to implement high-level structures
  - For example, stack, heap, static area, and virtual function tables
  - Depends on the features of the source and the target language
- Compilers create and manage a runtime environment in which the target programs are executed
- Runtime deals with the layout, allocation, and deallocation of storage locations, linkages between procedures, and passing parameters among other concerns

Issues Dealt with Runtime Environments

- How to pass parameters when a procedure is called?
- What happens to locals when procedures return from an activation?
- How to support recursive procedures?
- Can a procedure refer to nonlocal names? If yes, then how?
- …

Storage Organization

- Target program runs in its own logical space
- Size of generated code is usually fixed at compile time
  - Unless code is loaded or produced dynamically
- Compiler can place the executable at fixed addresses
- Runtime storage can be subdivided into
  - Target code
  - Static data objects such as global constants
  - Stack to keep track of procedure activations and local data
  - Heap to keep all other information like dynamic data



Strategies for Storage Allocation

- Static allocation – Lay out storage at compile time only by studying the program text
  - Memory allocated at compile time will be in the static area
- Dynamic allocation – Storage allocation decisions are made when the program is running
  - Stack allocation – Manage run-time allocation with a stack storage
    - Local data are allocated on the stack
  - Heap allocation – Memory allocation and deallocation can be done at any time
    - Requires memory reclamation support

Static Allocation

- Names are bound to storage locations at compilation time
  - Bindings do not change, so no run time support is required
  - Names are bound to the same location on every invocation
  - Values are retained across activations of a procedure
- Limitations
  - Size of all data objects must be known at compile time
  - Data structures cannot be created dynamically
  - Recursive procedures are not allowed

Stack vs Heap Allocation

Stack	Heap
• Allocation/deallocation is automatic	• Allocation/deallocation is explicit
• Less expensive	• More expensive
• Space for allocation is limited	• Challenge is fragmentation

Static vs Dynamic Allocation

Static	Dynamic
• Variable access is fast • Addresses are known at compile time • Cannot support recursion	• Variable access is slow • Accesses need redirection through stack/heap pointer • Supports recursion

Procedure Abstraction

Procedure Calls

- Procedure definition is a declaration that associates an identifier with a statement (procedure body)
  - Formal parameters appear in declaration
  - Actual parameters appear when a procedure is called
- Important abstraction in programming
  - Defines critical interfaces among large parts of a software

Procedure Calls

- Creates a controlled execution environment
  - Each procedure has its own private named storage or name space
  - Executing a call instantiates the callee’s name space
- Abstractions provided by procedures
  - Control abstraction
  - Name space
  - External interface

Control Abstraction

- Each language has rules to
  - Invoke a procedure
  - Map a set of arguments from the caller’s name space to the callee’s name space
  - Return control to the caller, and continue execution after the call
- Linkage convention standardizes the actions taken by the compiler and the OS to make a procedure call

Procedure Calls

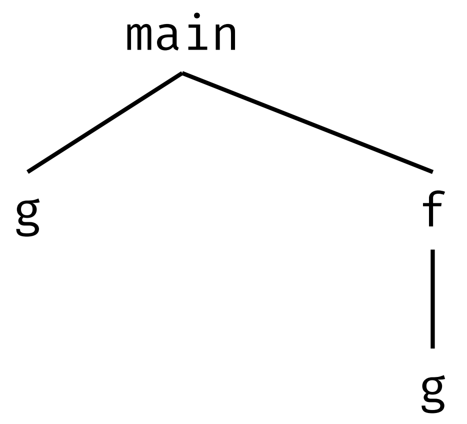
- Each execution of a procedure P is an **activation** of the procedure P
- A procedure is recursive if an activation can begin before an earlier activation of the same procedure has ended
  - If procedure is recursive, several activations may be alive at the same time

- The **lifetime** of an activation of P is all the steps to execute P including all the steps in procedures that P calls
- Given activations of two procedures, their lifetimes are either non-overlapping or nested

Activation Tree

- Depicts the way control enters and leaves activations
  - Root represents the activation of main
  - Each node represents activation of a procedure
  - Node a is the parent of b if control flows from a to b
  - Node a is to the left of b if lifetime of a occurs before b
- Flow of control in a program corresponds to depth-first traversal of activation tree

```
int g() { return 42; }
int f() { return g(); }
main() {
  g();
  f():
}
```



Quicksort Code

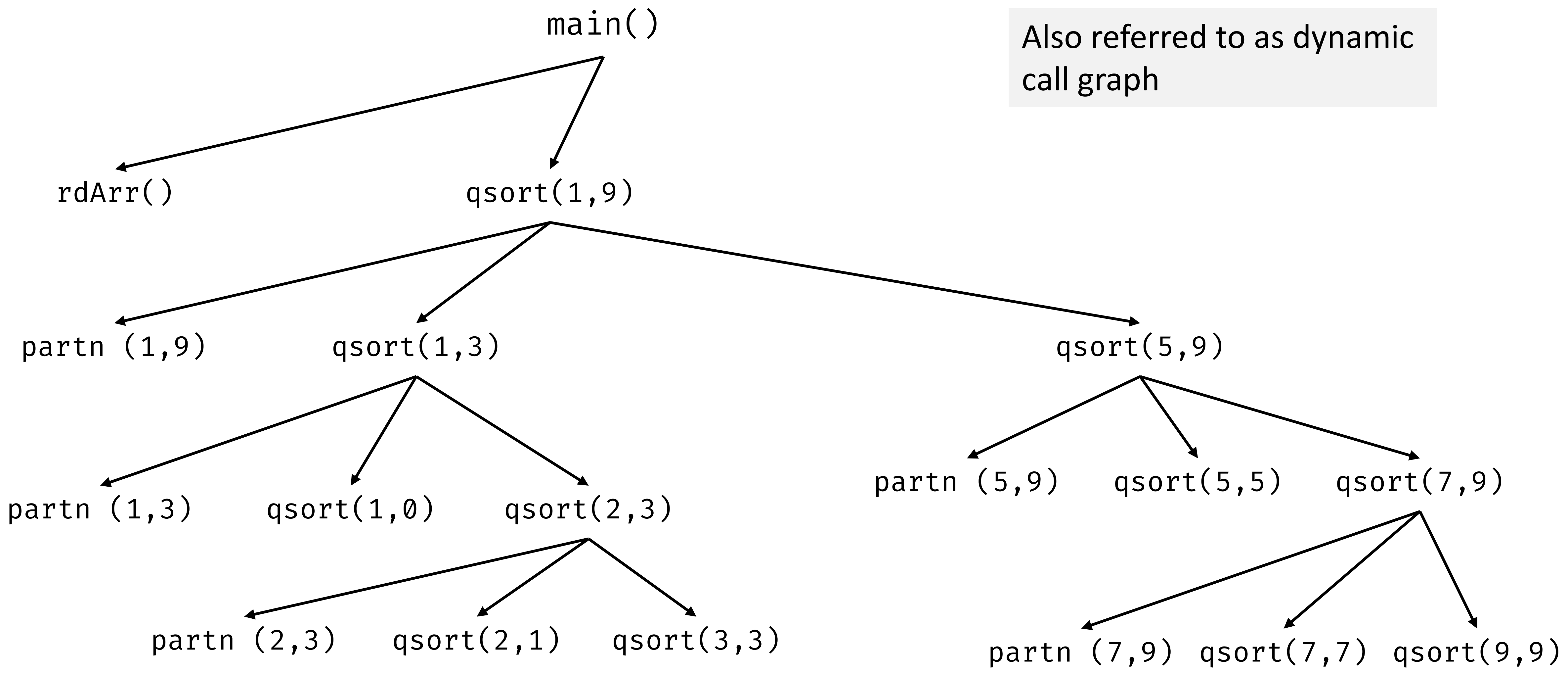
```
int a[11];
void readArray() {
  int i;
  …
}

int partition(int m, int n) {
  …
}
```

```
void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quicksort(m, i-1);
    quicksort(i+1, n);
  } }

int main() {
  readArray();
  a[0] = -99999;
  a[10] = 99999;
  quicksort(1, 9);
}
```

Activation Tree



Also referred to as dynamic call graph

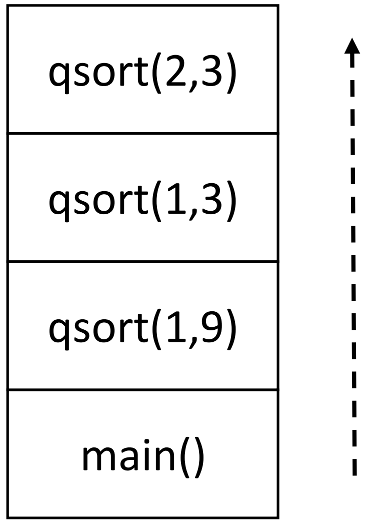
Example of Procedure Activations

$$\begin{tikzcd}[column sep=tiny]
 & & & \text{main()} \arrow[dlll] \arrow[d] & & & & \\
\text{rdArr()} & & & \text{qsort(1,9)} \arrow[dlll] \arrow[dl] \arrow[drrr] & & & & \\
\text{partn (1,9)} & & \text{qsort(1,3)} \arrow[dll] \arrow[dl] \arrow[d] & & & & \text{qsort(5,9)} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{partn (1,3)} & \text{qsort(1,0)} & \text{qsort(2,3)} \arrow[dl] \arrow[d] \arrow[dr] & & & \text{partn (5,9)} & \text{qsort(5,5)} & \text{qsort(7,9)} \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & \text{partn (2,3)} & \text{qsort(2,1)} & \text{qsort(3,3)} & & & \text{partn (7,9)} & \text{qsort(7,7)} & \text{qsort(9,9)}
\end{tikzcd}$$

```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
    ...
    leave quicksort(1,3)
    enter quicksort(5,9)
    ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

Control Stack

- Procedure calls and returns are usually managed by a run-time stack called the control stack
- Each live activation has an activation record on the control stack
  - Stores control information and data storage needed to manage the activation
  - Also called a frame
- Frame is pushed when activation begins and popped when activation ends
- Suppose node n is at the top of the stack, then the stack contains the nodes along the path from n to the root



Is a Stack Sufficient?

When will a control stack work?

When will a control stack not work?

Is a Stack Sufficient?

When will a control stack work?

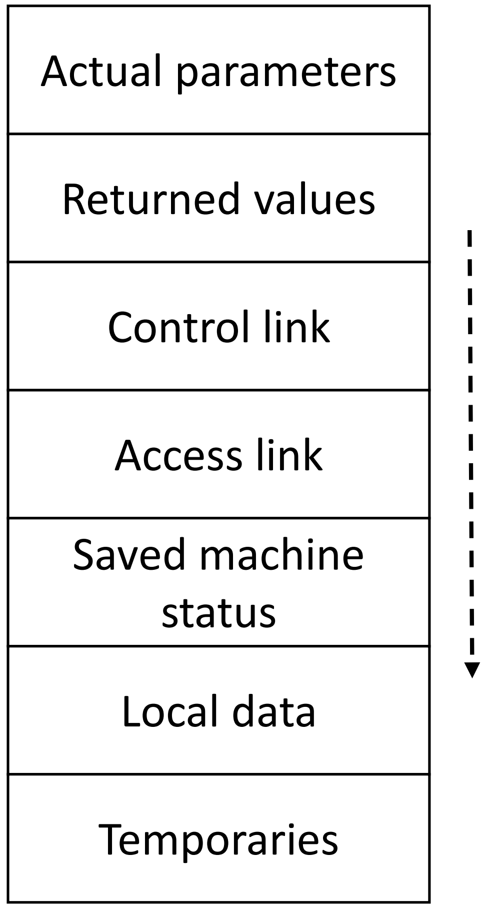
- Once a function returns, its activation record cannot be referenced again
- We do not need to store old nodes in the activation tree
- Every activation record has either finished executing or is an ancestor of the current activation record

When will a control stack not work?

- Once a function returns, its activation record cannot be referenced again
- Function closures – procedure and run-time context to define free variables

Activation Record

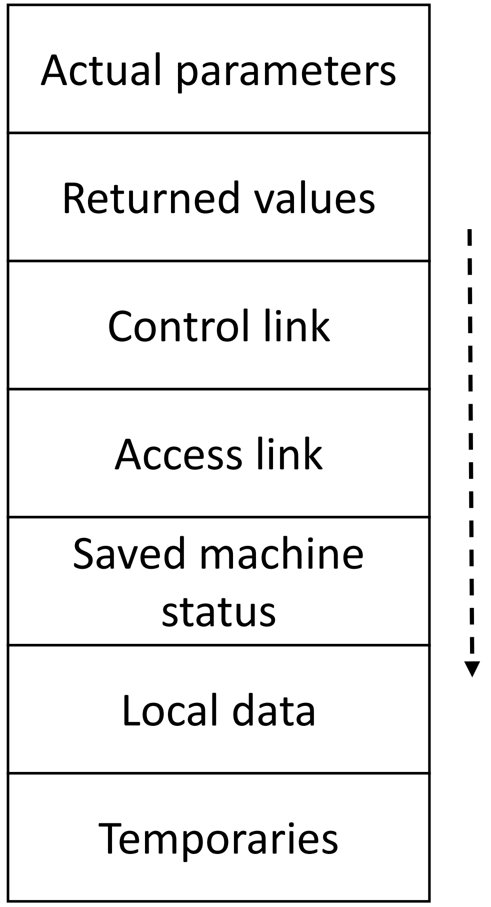
- A pointer to the current activation record is maintained in a register
- Fields in an activation record
    - Temporaries – evaluation of expressions
    - Local data – field for local data
    - Saved machine status – information about the machine state before the procedure call
        - Return address (value of program counter)
        - Register contents
    - Access link – access nonlocal data



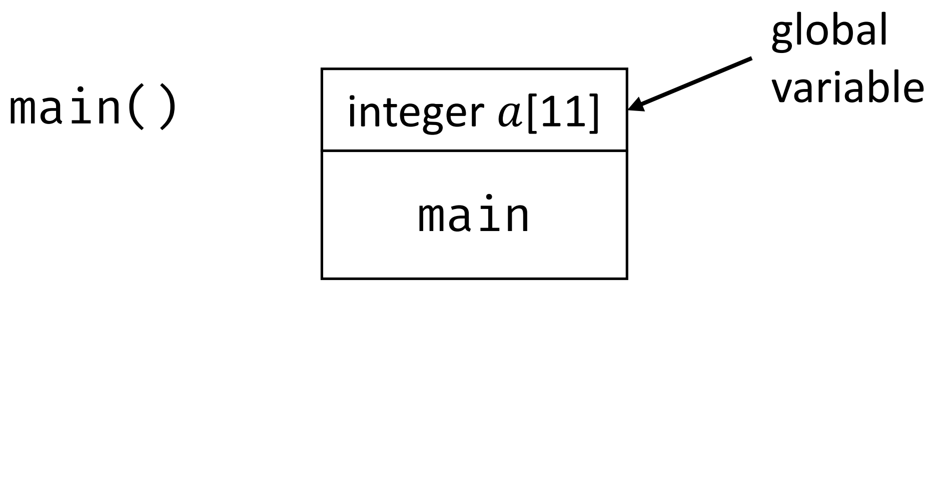
Activation Record

- Fields in an activation record
  - Control link – Points to the activation record of the caller
  - Returned values – Space for the value to be returned
  - Actual parameters – Space for actual parameters

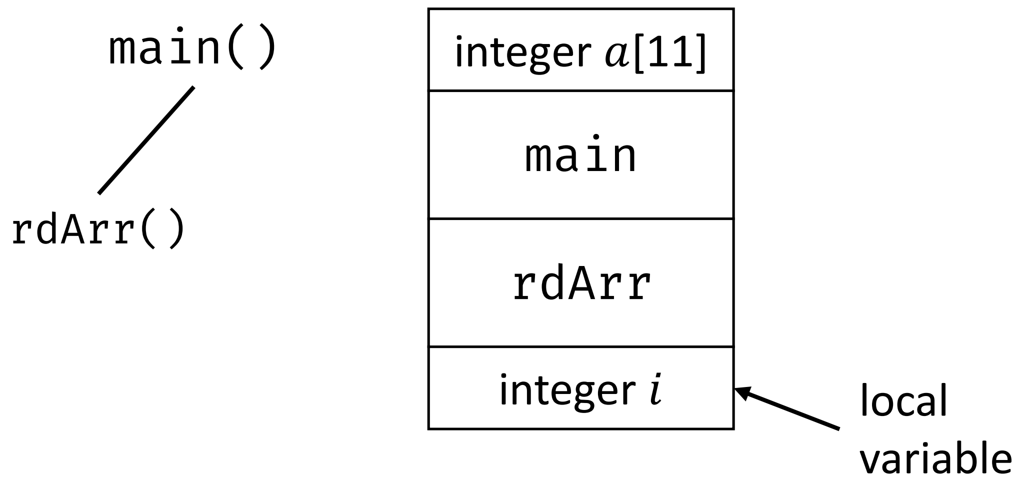
- Contents and position of fields may vary with language and implementations



Example of Activation Records

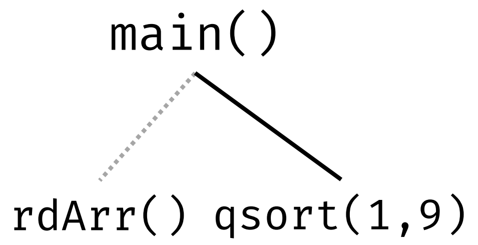


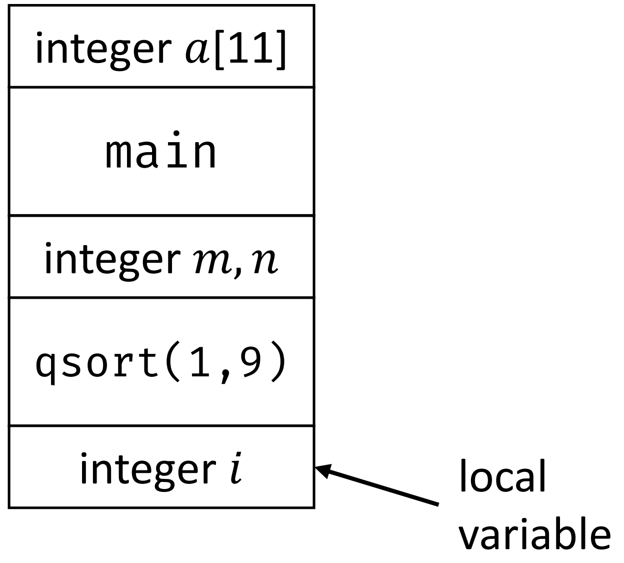
Frame for main()



rdArr() is activated

Example of Activation Records





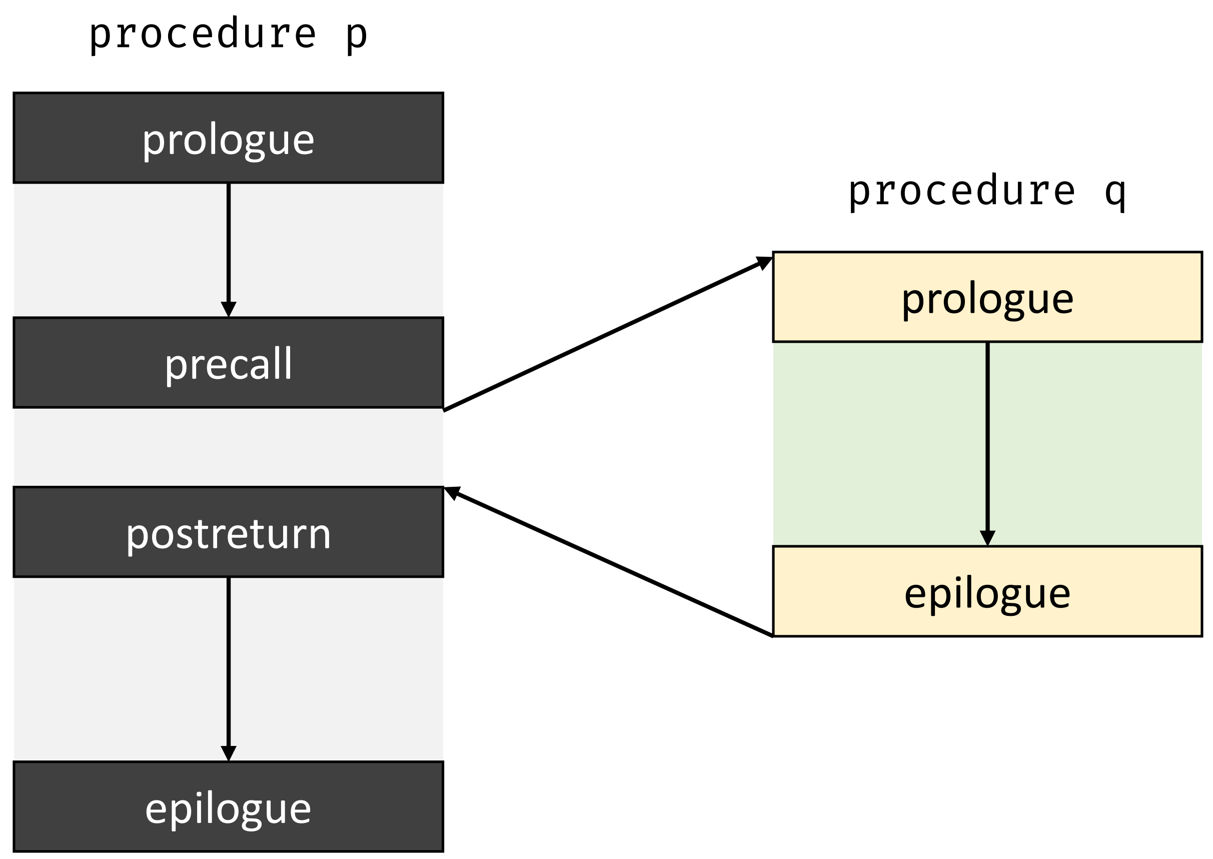
`rdArr()` is popped, `qsort(1,9)` is pushed

What is in G’s Activation Record when F( ) calls G( )?

- If a procedure F calls G, then G’s activation record contains information about both F and G
- F is suspended until G completes, at which point F resumes
  - G’s activation record contains information needed to resume execution of F
- G’s activation record contains
  - G’s return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G’s local variables

A Standard Procedure Linkage

- Procedure linkage is a contract between the compiler, the OS, and the target machine
- Divides responsibility for naming, allocation of resources, addressability, and protection



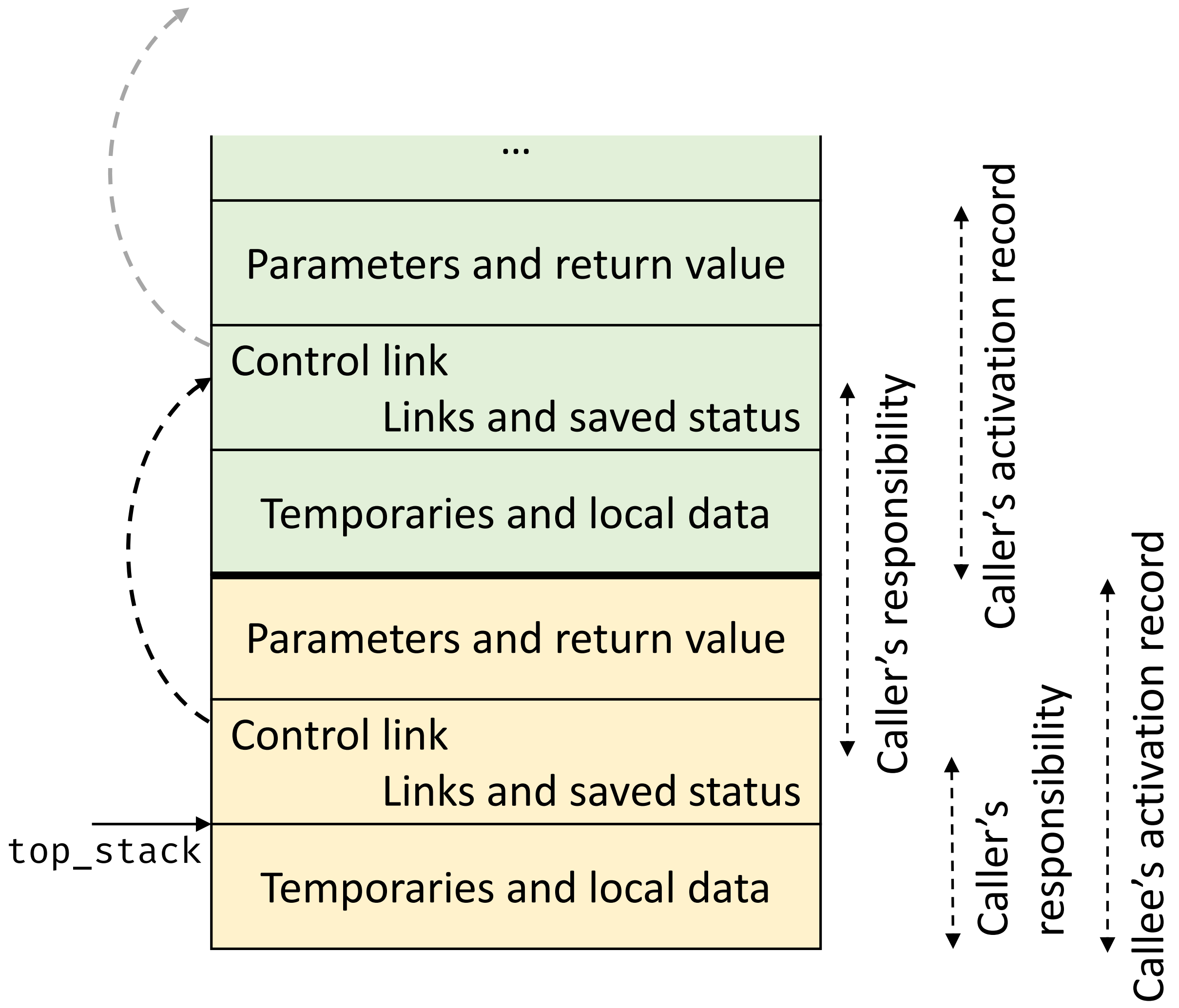
Calling Sequence

- Calling sequence allocates an activation record on the stack and enters information into its fields
  - Responsibility is shared between the caller and the callee
- Return sequence is code to restore the state of the machine so the calling procedure can continue its execution after the call

Calling Sequence

- Policies and implementation strategies can differ
    - Place values communicated between caller and callee at the beginning of the callee’s activation record, close to the caller's activation record
    - Fixed-length items are placed in the middle
    - Data items whose size are not known during intermediate code generation are placed at the end of the activation record
    - Top-of-stack points to the end of the fixed-length fields
        - Fixed-length data items are accessed by fixed offsets from top-of-stack pointer
        - Variable-length fields records are actually "above" the top-of-stack

Division of Tasks Between Caller and Callee


...
Parameters and return value
Control link
Links and saved status
Temporaries and local data
Parameters and return value
Control link
Links and saved status
top_stack
Temporaries and local data
Caller's responsibility
Caller's activation record
Caller's responsibility
Callee's activation record

Division of Tasks Between Caller and Callee

Call sequence

- Caller evaluates the actual parameters
- Caller stores a return address and the old value of `top_stack` into the callee's activation record
- Caller then increments `top_stack` past the caller's local data and temporaries and the callee's parameters and status fields
- Callee saves the register values and other status information
- Callee initializes its local data and begins execution

Division of Tasks Between Caller and Callee

Return Sequence

- Callee places the return value next to the parameters
- Callee restores `top_stack` and other registers
- Callee branches to the return address that the caller placed in the status field
- Caller copies return value into its activation record

Communication between Procedures

- Calling convention is an implementation-level detail to specify how callees receive parameters from their caller and how callees return a result
- Parameter binding maps the actual parameters at a call site to the callee's formal parameters
- Types of mapping conventions
  - Pass by value
  - Pass by reference
  - Pass by name

Call by Value and Call by Reference

Call by Value

- Convention where the caller evaluates the actual parameters and passes their r-values to the callee
- Formal parameter in the callee is treated like a local name
- Any modification of a value parameter in the callee is not visible in the caller
- Example: C and Pascal

Call by Reference

- Convention where the compiler passes an address for the formal parameter to the callee
    - Any redefinition of a reference formal parameter is reflected in the corresponding actual parameter
- A formal parameter requires an extra indirection

Call by Name

- Reference to a formal parameter behaves as if the actual parameter had been textually substituted in its place
  - Renaming is used in case of clashes
  - Can update the given parameters
- Actual parameters are evaluated inside the called function
- Example: Algol-60

```
procedure double(x);
  real x;
begin
  x := x*2
end;
```

`double(c[j])` ➡ `c[j] := c[j]*2`

https://www2.cs.sfu.ca/~cameron/Teaching/383/PassByName.html

Challenges with Call by Name

```
procedure swap(a, b)
integer a, b, temp;
begin
    temp := a
    a :=  b
    b := temp
end;
```

- What will happen when you call `swap(i, x[i])`?

Name Spaces

- Scope is the part of a program to which a name declaration applies
- Scope rules provide control over access to data and names
- Lexical scope – a name refers to the definition that is lexically closest to the use
- Compilers can use a static coordinate for a name for lexically-scoped languages
  - Consider a name x declared in a scope s
  - Static coordinate is a pair $<l, o>$
    - l is the lexical nesting level of s and o is the offset where x is stored in the scope's data area

Nested Lexical Scopes in Pascal

```
program Main_0(inp, op);
  var x_1, y_1, z_1: integer;
  procedure Fee_1;
    var x_2: integer;
    begin { Fee_1 }
      x_2 := 1;
      y_1 := x_2*2+1
    end;
  procedure Fie_1;
    var y_2: real;
    procedure Foe_2;
      var z_3: real;
      procedure Fum_3;
        var y_4: real;
        ...
```

Scope	x	y	z
Main	<1,0>	<1,4>	<1,8>
Fee	<2,0>	<1,4>	<1,8>
Fie	<1,0>	<2,0>	<2,8>
Foe	<1,0>	<2,0>	<3,0>
Fum	<1,0>	<4,0>	<3,0>

Lexical and Dynamic Scoping

- A variable that a procedure refers to and that is declared outside the procedure’s own scope is called a free variable
- With lexical scoping, a free variable is bound to the declaration for its name that is lexically closest to the use
- With dynamic scoping, a free variable is bound to the variable most recently created at runtime
- Lexical scoping is more popular
    - Dynamic scoping is relatively challenging to implement
    - Both are identical as far as local variables are concerned

Lexical and Dynamic Scope

```
int x = 1, y = 0;
int g(int z) {
 return x + z;
}
int f(int y) {
  int x;
  x = y + 1;
  return g(x * y);
}

y=f(3);
```

- What is the value of `x` after the call to `g( )`?
  - Lexical scope: `x = ?`
  - Dynamic scope: `x = ?`

Lexical and Dynamic Scope

```
int x = 1, y = 0;
int g(int z) {
 return x + z;
}
int f(int y) {
  int x;
  x = y + 1;
  return g(x * y);
}

y=f(3);
```

- What is the value of x after the call to g( )?
  - Lexical scope: `x = 1`
  - Dynamic scope: `x = 4`

Lexical and Dynamic Scoping in Perl

```
$x = 10;
sub f
{
   return $x;
}
sub g
{
  # If local is used, x uses dynamic scoping
  # If my is used, x uses lexical scoping
  local $x = 20;
  # my $x = 20;
  return f();
}
print g()."\n";
```

```
Dynamic scope

$ perl scope.pl
?
```

```
Lexical scope

$ perl scope.pl
?
```

Lexical and Dynamic Scoping in Perl

```
$x = 10;
sub f
{
   return $x;
}
sub g
{
  # If local is used, x uses dynamic scoping
  # If my is used, x uses lexical scoping
  local $x = 20;
  # my $x = 20;
  return f();
}
print g()."\n";
```

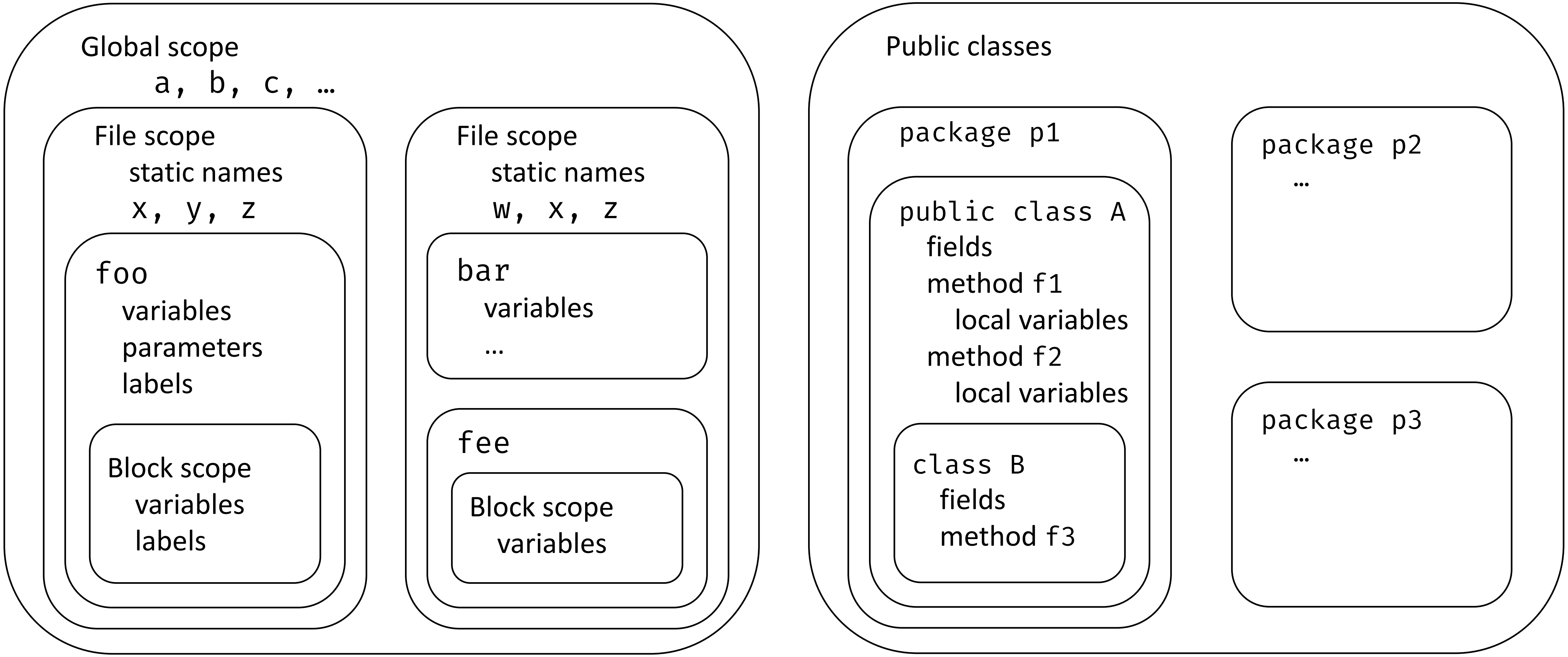
```
Dynamic scope

$ perl scope.pl
20
```

```
Lexical scope

$ perl scope.pl
10
```

Scoping Rules for C and Java Languages

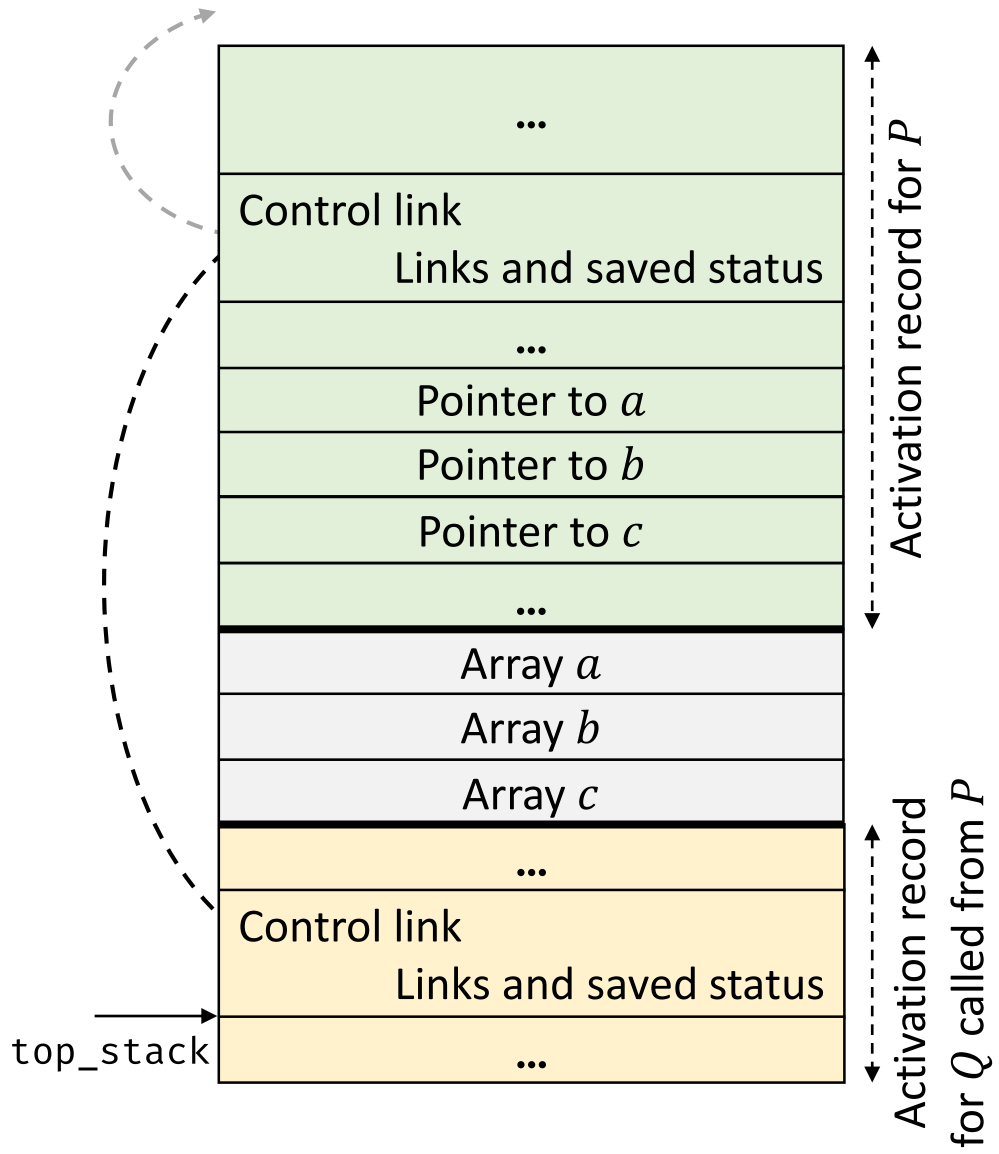


Allocating Activation Records

- Stack allocation
  - Activation records follow LIFO ordering
  - E.g.: Pascal, C, and Java
- Heap allocation
  - Needed when a procedure can outlive its caller
  - Garbage collection support eases complexity
  - E.g.: Implementations of Scheme and ML
- Static allocation
  - Procedure P cannot have multiple active invocations if it does not call other procedures

Variable Length Data on the Stack

- Data may be local to a procedure but the size may not be known at compile time
  - For example, a local array whose size depends upon a parameter
- Data may be allocated in the heap but would require garbage collection
- Possible to allocate variable-sized local data on the stack



Data Access without Nested Procedures

- Consider C-family of languages
- Any name local to a procedure is nonlocal to other procedures

- Simple rules
  - Global variables are in static storage
    - Addresses are fixed and determined at compile time
  - Any other name must be local to the activation at the top of the stack

Access to Nonlocal Data in Nested Procedures

- This is challenging for nested procedures

- Suppose procedure p at lexical level m is nested in procedure q at level n, and x is declared in q
- Our aim is to resolve a nonlocal name x in p

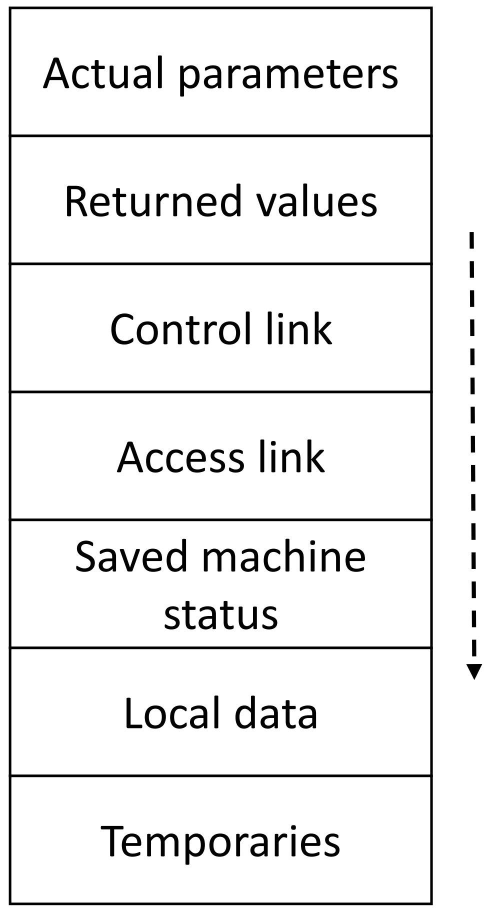
- Compiler models the reference by a static distance coordinate $<m - n, o>$ where o is x's offset in the activation record for q
  - Compiler needs to translate $<m - n, o>$ into a runtime address

Access to Nonlocal Data in Nested Procedures

- Finding the relevant activation of q from an activation of p is a dynamic decision
  - We cannot use compile-time decisions since there could be many activation records of p and q on the stack
- Two common strategies: access links and displays

Access Links

- Suppose procedure p is nested immediately within procedure q
- Access link in any activation of p points to the most recent activation of q
- Access links form a chain up the nesting hierarchy
  - All activations whose data and procedures are accessible to the currently executing procedure



Nesting Depth

- Procedures not nested within other procedures have nesting depth 1
  - For example, all functions in C have depth 1
- If p is defined immediately within a procedure at depth i, then p is at depth $i + 1$

Quicksort in ML using Nested Procedures

```
1) fun sort (inputFile, outputFile) =
      let
2)        val a = array(11,0);
3)        fun readArray(inputFile) = ... ;
4)          ...a... ;
5)        fun exchange(i, j) =
6)          ...a... ;
```

```
7)      fun quicksort(m,n) =
          let
8)          val v= ... ;
9)          fun partition(y,z) =
10)           ...a...v...exchange...
          in
11)         ...a...v...partition...quicksort
          end
        in
12)     ...a...readArray...quicksort...
        end;
```

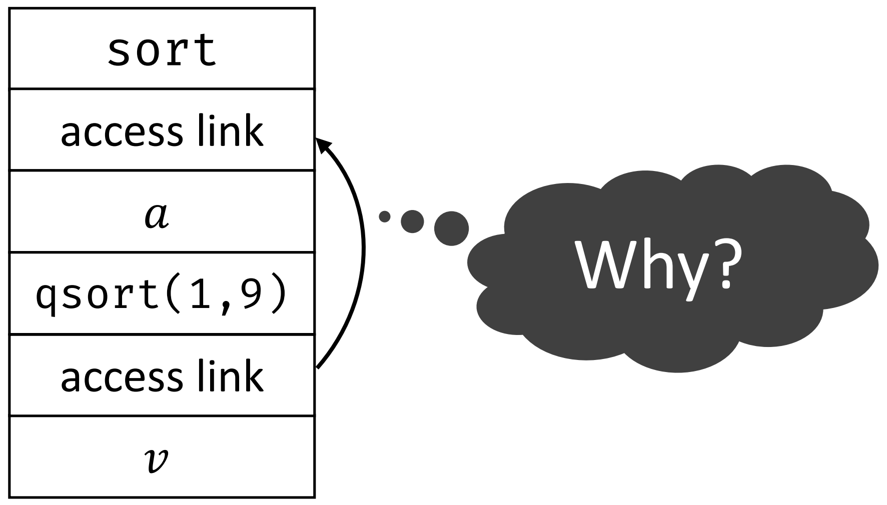
Procedure	Nesting Depth
sort	1
readArray	2
exchange	2
quicksort	2
partition	3

How to find nonlocal x?

- Suppose procedure p is at the top of the stack and has depth n_p, and q is a procedure that surrounds p and has depth n_q
  - Usually $n_q < n_p$, $n_q == n_p$ only if p and q are the same

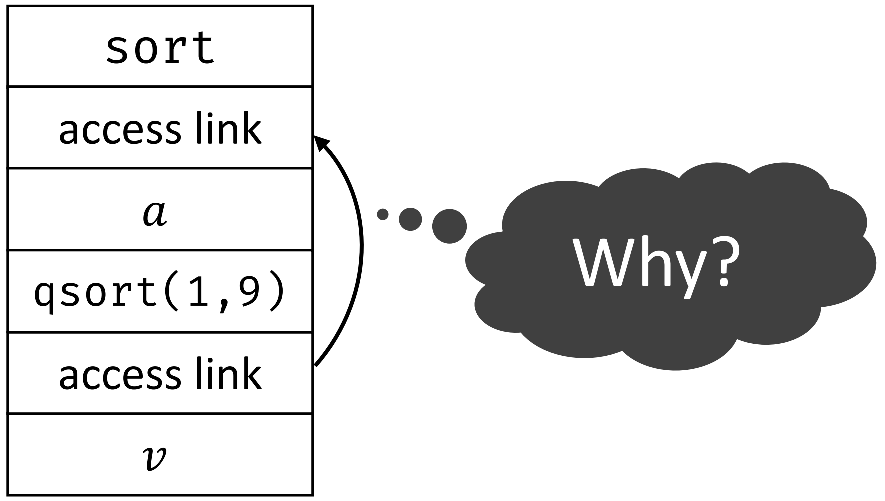
- Follow the access link $(n_p - n_q)$ times to reach an activation record for q
- That activation record for q will contain a definition for local x

Example of Access Links



Because sort called quicksort?

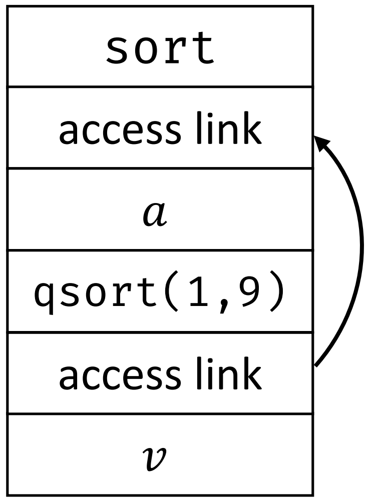
Example of Access Links

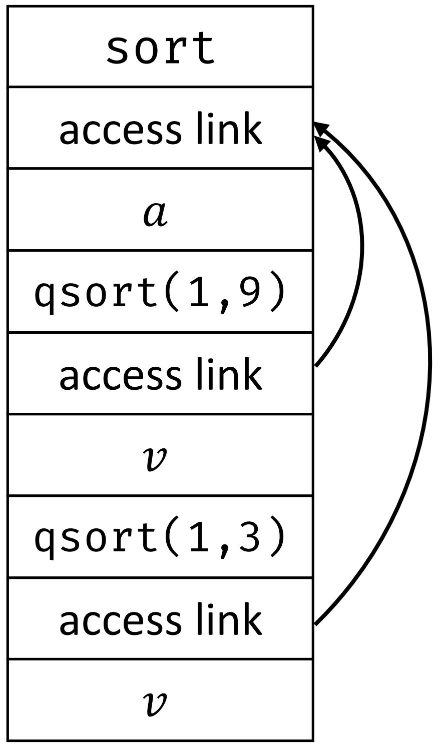


Because sort called quicksort?

No, because sort is the most closely nested function surrounding quicksort

Example of Access Links

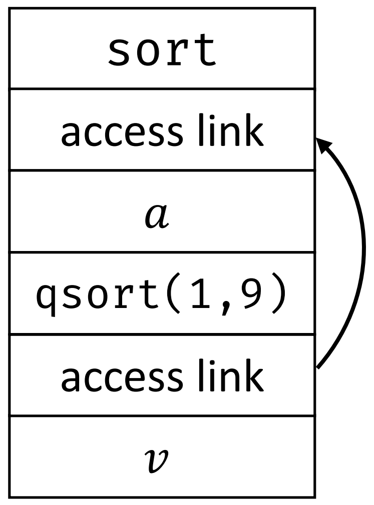


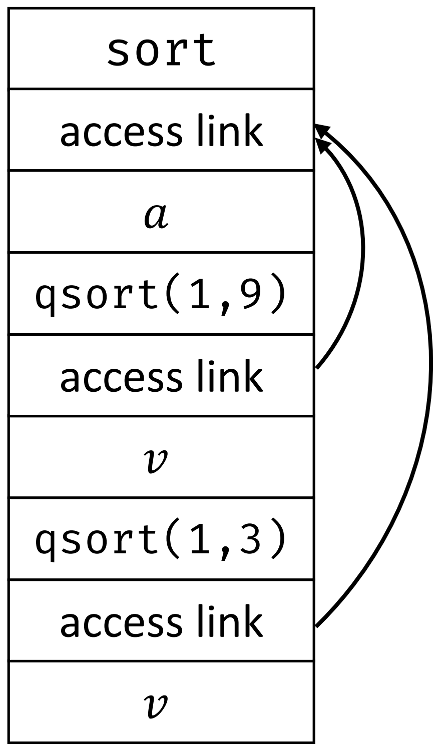


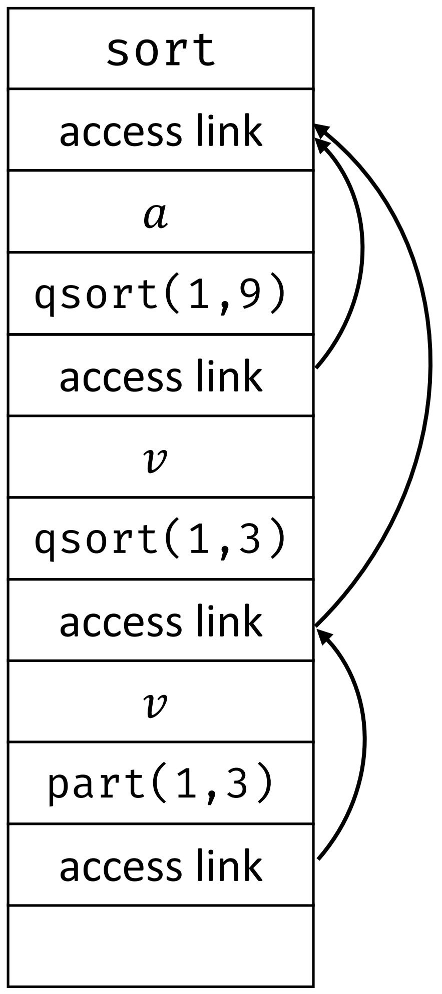
Because sort called quicksort?

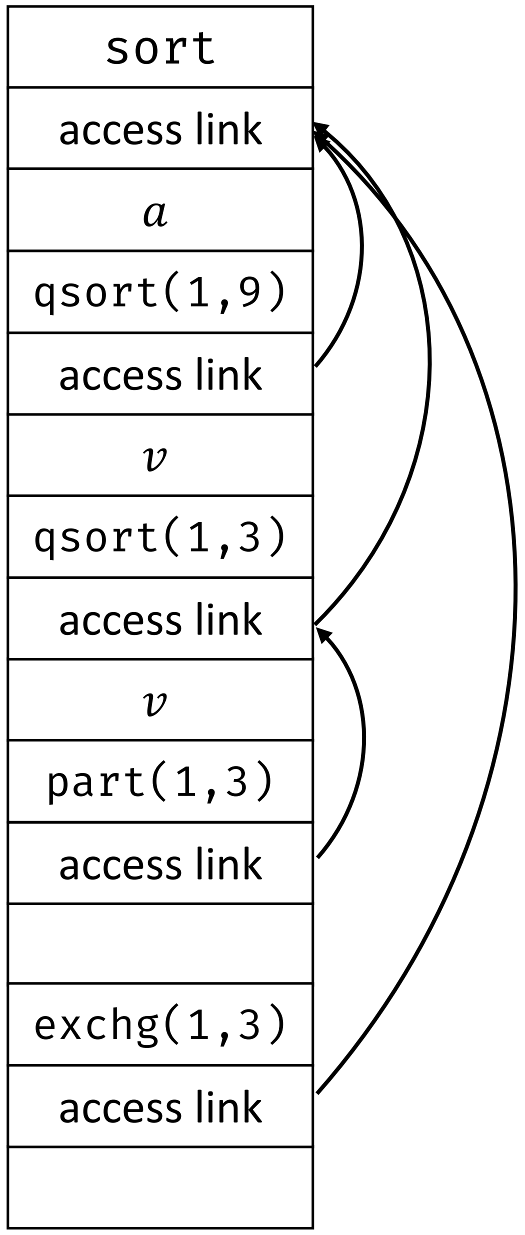
No, because sort is the most closely nested function surrounding quicksort

Example of Access Links

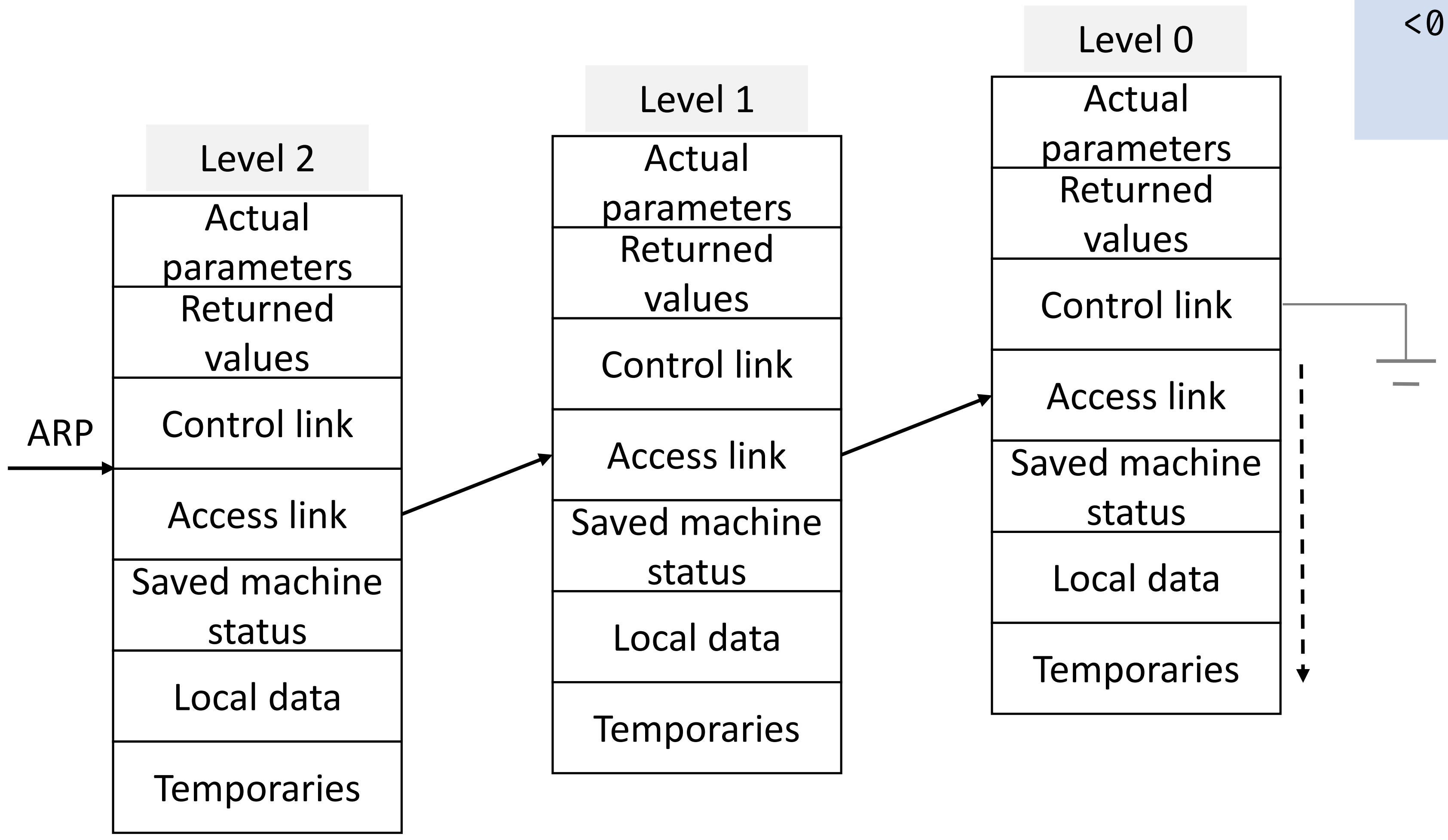








Manipulating Access Links



Coordinate	Code
<2, 24>	loadAI r_{arp}, 24 ⇒ r_2
<1, 12>	loadAI r_{arp}, aloff ⇒ r_1 loadAI r_1, 12 ⇒ r_2
<0, 16>	loadAI r_{arp}, aloff ⇒ r_1 loadAI r_1, aloff ⇒ r_1 loadAI r_1, 16 ⇒ r_2

Manipulating Access Links

- Code to setup access links is part of the calling sequence
- Suppose procedure q at depth n_q calls procedure p at depth n_p
- The code for setting up access links depends upon whether or not the called procedure is nested within the caller

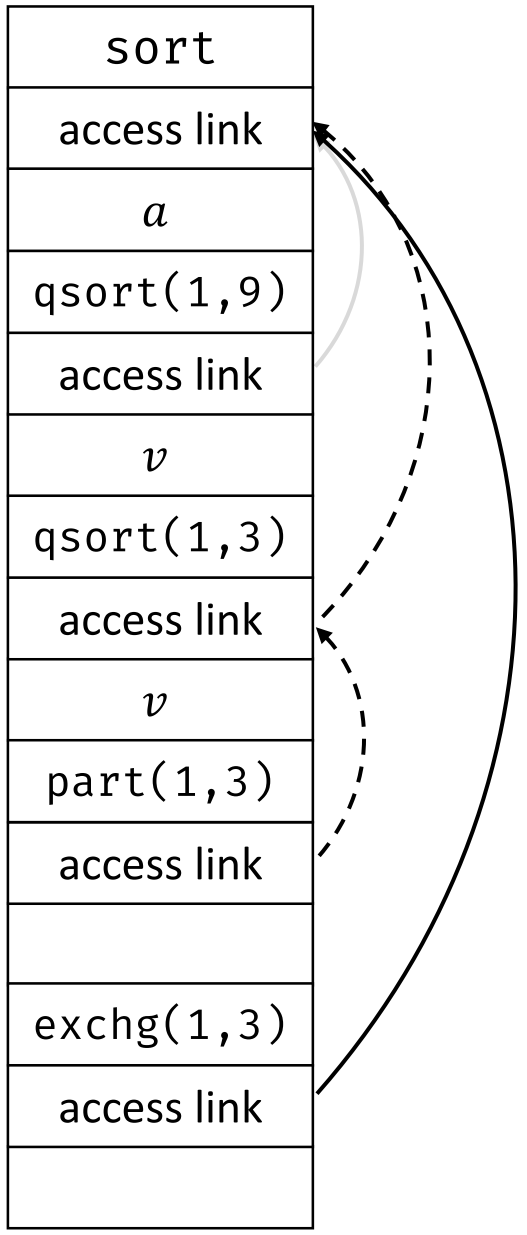
Manipulating Access Links

- Case 1: $n_q < n_p$
  - Called procedure p is nested more deeply than q
  - Therefore, p must be declared in q, or the call by q will not be within the scope of p
  - Access link in p should point to the access link of the activation record of the caller q
- Case 2: $n_p == n_q$
  - Procedures are at the same nesting level
  - Access link of called procedure p is the same as q

Manipulating Access Links

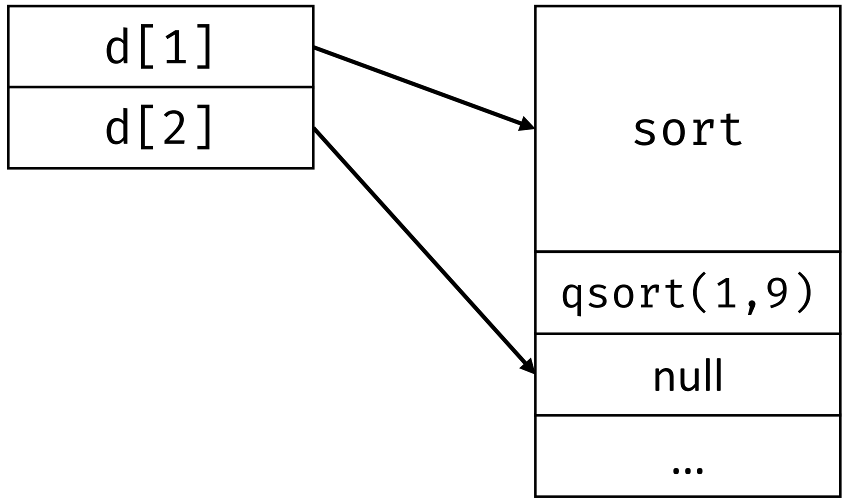
- Case 3: $n_q > n_p$
  - For the call within q to be in the scope of p, q must be nested within some procedure r, while p is defined immediately within r
  - Top activation record for r can be found by following chain of access links for $n_q - n_p + 1$ hops
    - Start in the activation record for q
  - Access link for q will go to the activation for r

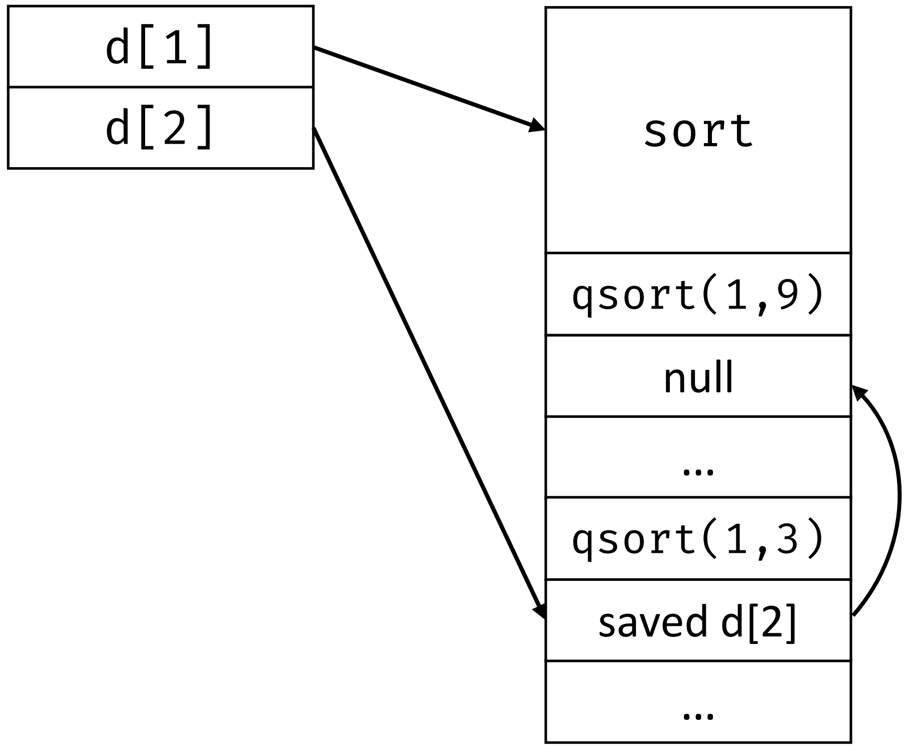
- Nesting depth of `exchange` is 2
- Nesting depth of `partition` is 3



Displays

- Display is a global array to hold the activation record pointers for the most recent activations of procedures at each lexical level

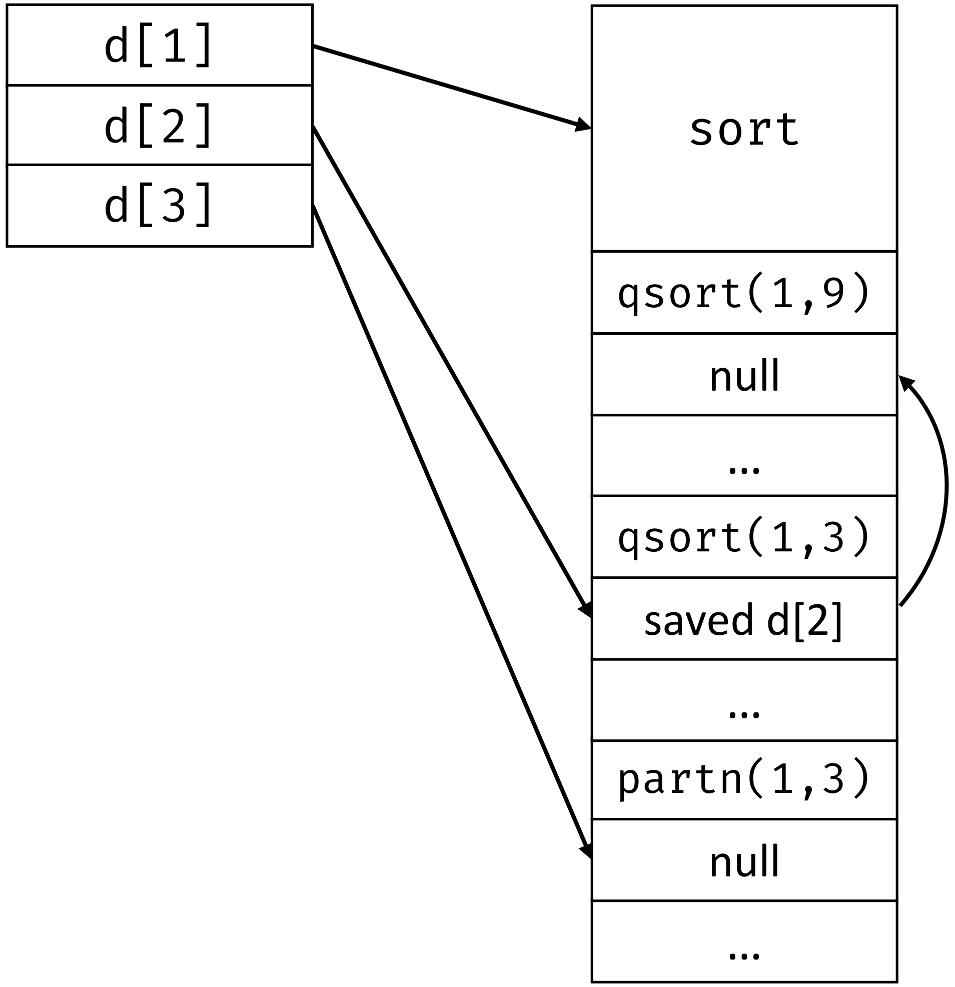


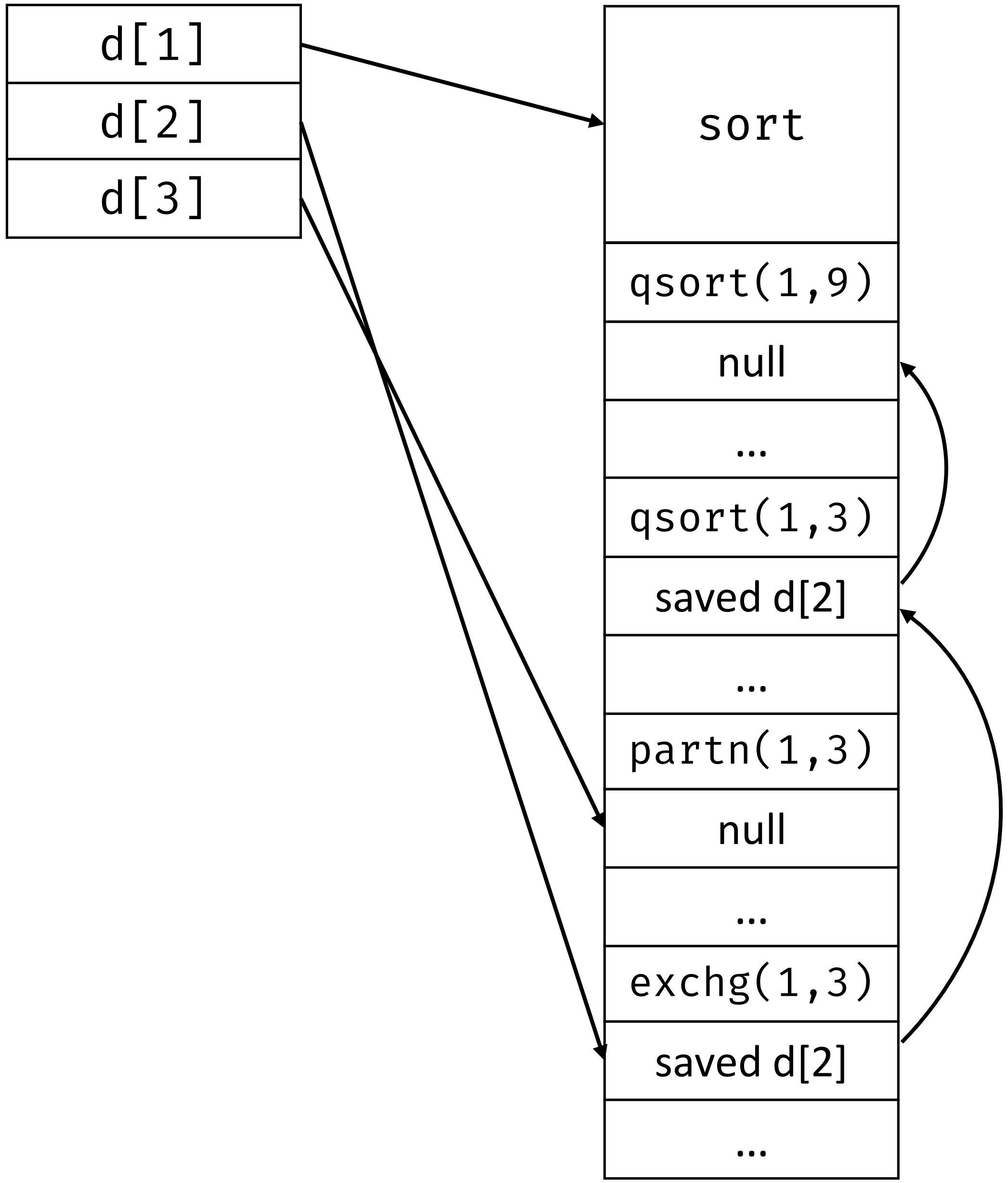


Insight in Using Displays

- Suppose a procedure p is executing and needs to access element x belonging to procedure q
- The runtime only needs to search in activations from $d[i]$, where i is the nesting depth of q
  - Follow the pointer $d[i]$ to the activation record for q, wherein x should be defined at a known offset

Displays


d[1]
d[2]
d[3]
sort
qsort(1,9)
null
...
qsort(1,3)
saved d[2]
...
partn(1,3)
null
...


d[1]
d[2]
d[3]
sort
qsort(1,9)
null
...
qsort(1,3)
saved d[2]
...
partn(1,3)
null
...
exchg(1,3)
saved d[2]
...

Access Links vs Displays

Access Links	Displays
• Cost of lookup varies • Common case is cheap, but long chains can be costly	• Cost of lookup is constant
• Cost of maintenance also is variable	• Cost of maintenance is constant

Heap Management

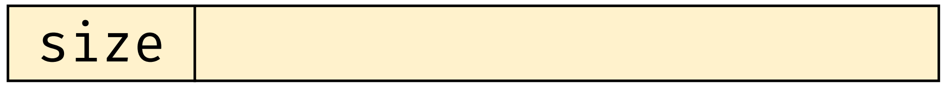
- Few runtime structures can outlive the called procedure
- Heap is used for allocating space for objects created at runtime
- Interface to the heap: `allocate(size)` and `free(addr)`
  - Commonly-used interfaces
    - malloc()/free() in C or new/delete in C++
- Allocation and deallocation may be completely manual (C/C++), semi-automatic (Java), or fully automatic (Lisp)

Heap Management

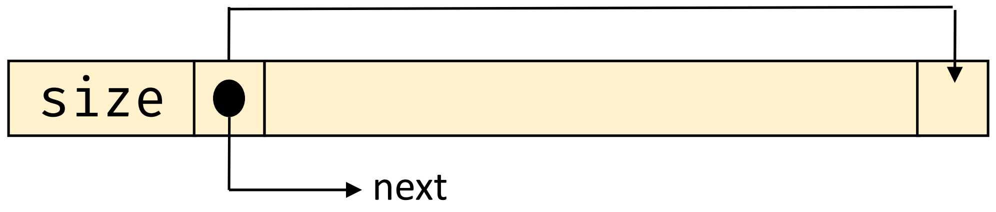
- Manages heap memory by implementing mechanisms for allocation and deallocation
  - Either manual or automatic strategies
- Goals
  - Space efficiency – minimize fragmentation
  - Program efficiency – take advantage of locality of objects in memory and make the program run faster
  - Low overhead – allocation and deallocation must be efficient

Heap Management

- First-fit allocation – emphasize speed over memory utilization
- Every block in the heap has a field for size



Allocated block



Free block

First-Fit Allocation

- `allocate(k)`
  - Traverse the free list to find a block b_i with size greater than k+1
  - If found, remove b_i from the free list and return pointer to the second word of b_i
    - If b_i is larger than k, then split the extra space and add to the free list
  - If not found, then request for more virtual memory
- `free(addr)`
  - Add b_j to the head of the free list

Reducing Fragmentation

- Merge free blocks
  - Check the preceding end-of-block pointer when processing b_j
  - Merge if both blocks are free
  - Can also merge with successor block
- Other variants – best-fit and next-fit allocation strategy
  - Best-fit strategy searches and picks the smallest (best) possible chunk that satisfies the allocation request
  - Next-fit strategy tries to allocate the object in the chunk that has been split recently

Problems with Manual Deallocation

- Common problems
  - Fail to delete data that is not required, called memory leak
    - Critical for performance of long-running or server programs
  - Reference deleted data, i.e., dangling pointer reference
  - These problems are hard to debug

- Possible solution is support for implicit deallocation
- Garbage collection is support for implicit deallocation of objects that reside on the runtime heap

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 7.1-7.4.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 6, 7.1-7.2.