Run-time Environments - Part 2

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

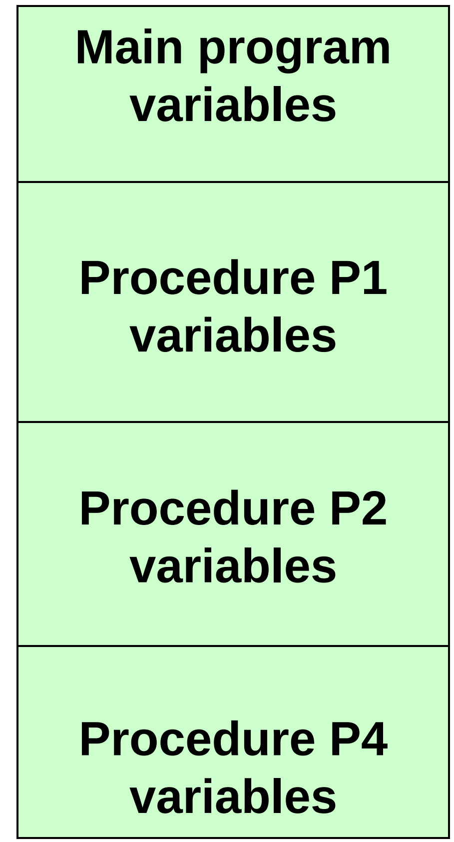
NPTEL Course on Compiler Design

Outline of the Lecture – Part 2

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile time
  - No stack/heap allocation; no overheads
  - Ex: Fortran IV and Fortran 77
  - Variable access is fast since addresses are known at compile time
  - No recursion



Main memory

Dynamic Data Storage Allocation

- Compiler allocates space only for golbal variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemened

Activation Record Structure

Return address
Static and Dynamic links (also called Access and Control link resp.)
(Address of) function result
Actual parameters
Local variables
Temporaries
Saved machine status
Space for local arrays

Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be at the top of the act. record.

Variable Storage Offset Computation

- The compiler should compute
  - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

Example of Offset Computation

P → Decl { Decl.inoffset↓= 0;}

$Decl \rightarrow T\ id\ ;\ Decl_1$
{enter(id.name↑, T.type↑, Decl.inoffset↓);
$Decl_1$.inoffset↓ = Decl.inoffset↓ + T.size↑;
Decl.outoffset↑ = $Decl_1$.outoffset↑; }

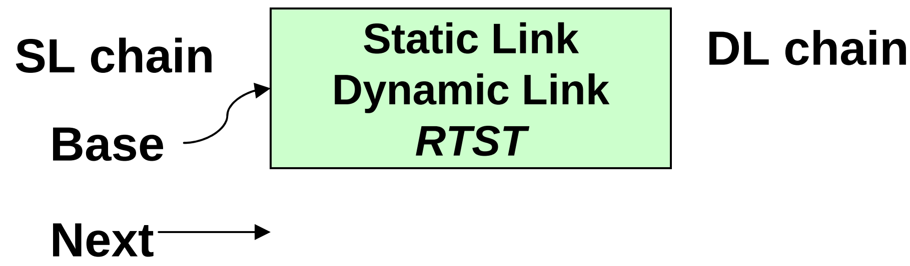
Decl → T id ; {enter(id.name↑, T.type↑, Decl.inoffset↓);
Decl.outoffset↑ = T.size↑;}

T → int {T.type↑ = inttype; T.size↑ = 4;}

T → float {T.type↑ = floattype; T.size↑ = 8;}

$T \rightarrow [num]\ T_1$ {T.type↑ = arraytype(T1.type↑, T1.size↑);
T.size↑ = T1.size↑ * num.value↑;}

Allocation of Activation Records



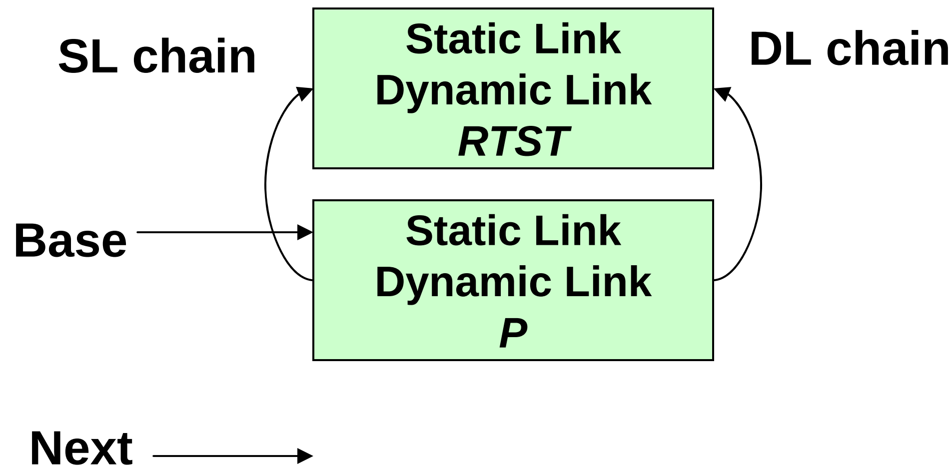
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end

Activation records are created at procedure entry time and destroyed at procedure exit time

RTST -> P -> R -> Q -> R

Allocation of Activation Records

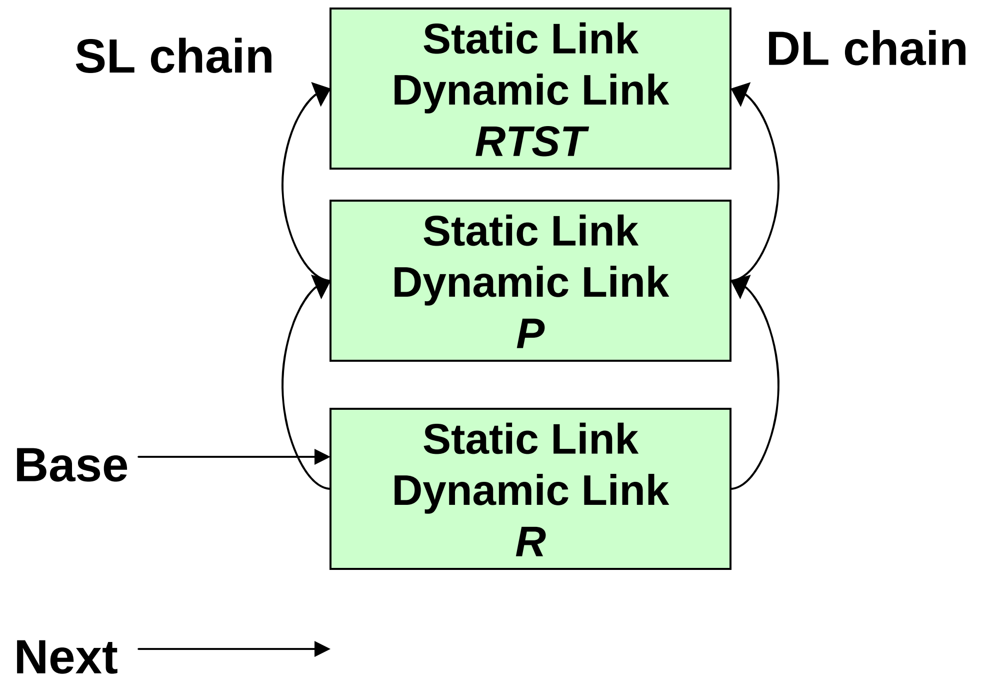
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> R -> Q -> R

Allocation of Activation Records

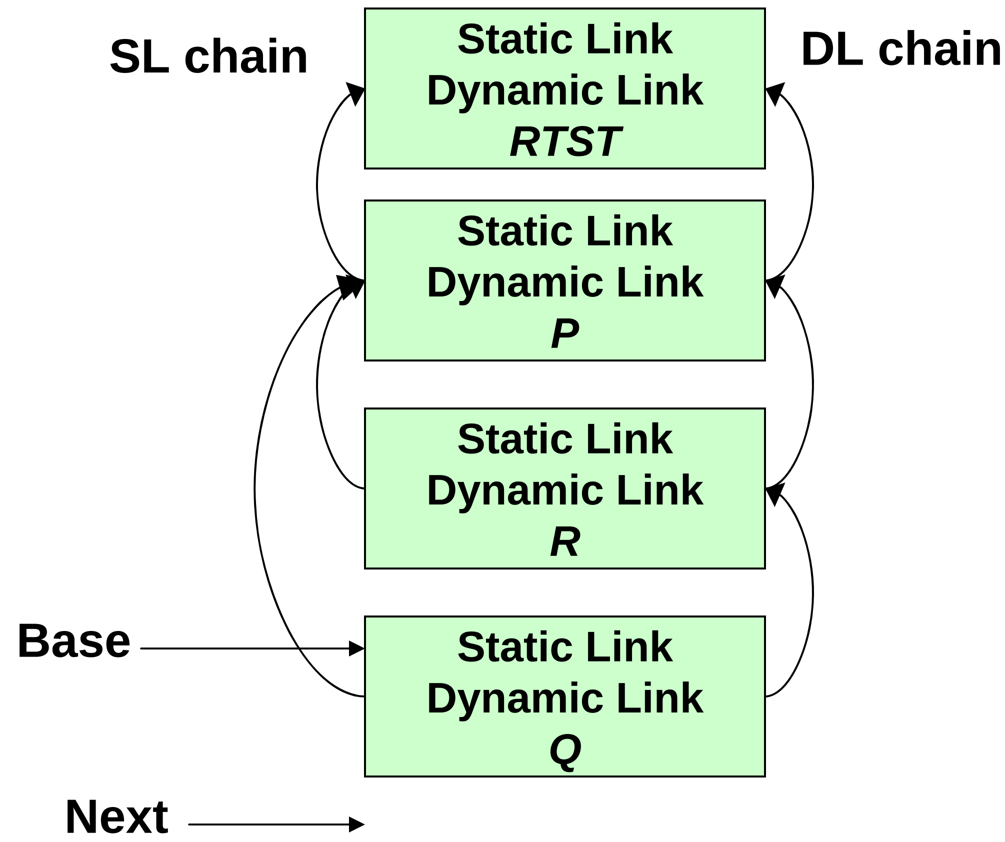
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> **R** -> Q -> R

Allocation of Activation Records

program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end

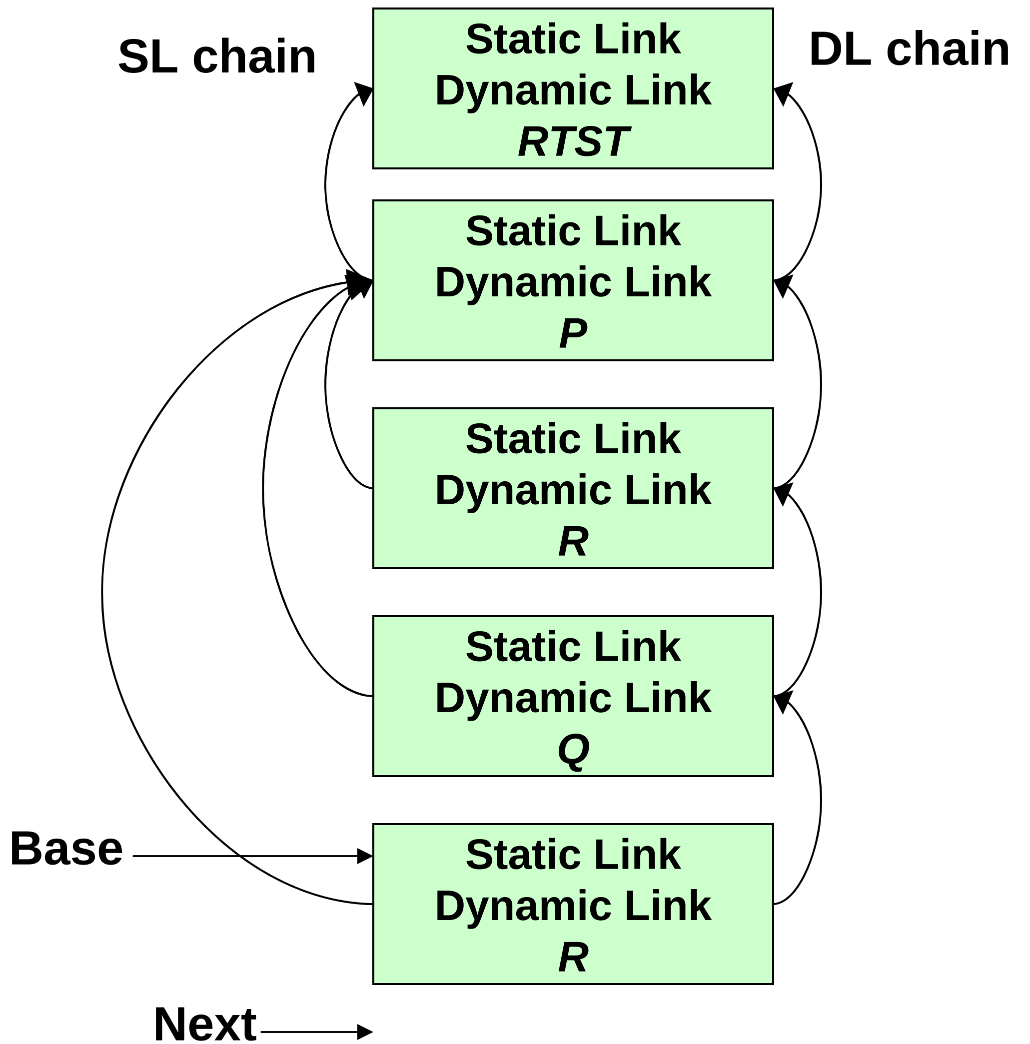


RTST -> **P** -> **R** -> **Q** -> R

Allocation of Activation Records

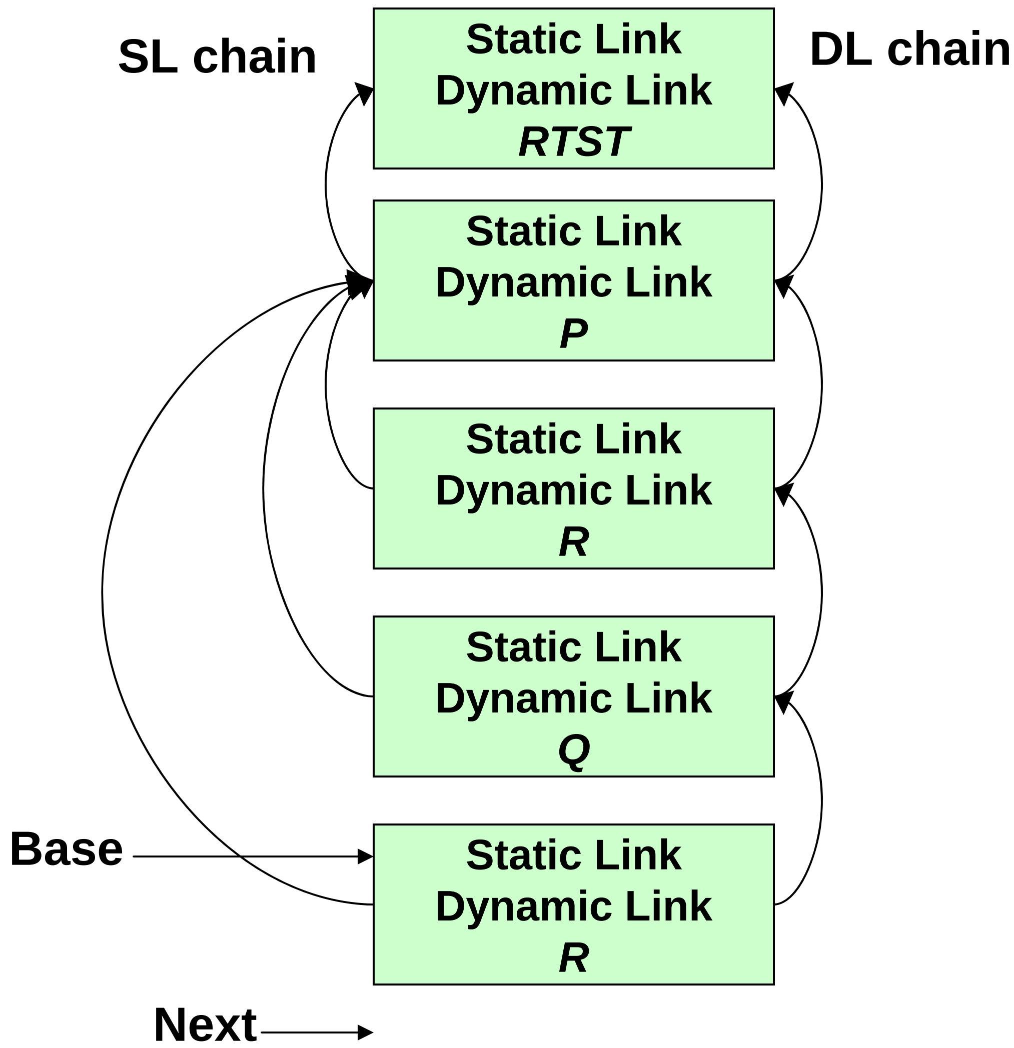
1 program *RTST*;
2 procedure *P*;
3 procedure *Q*;
begin *R*; end
3 procedure *R*;
begin *Q*; end
begin *R*; end
begin *P*; end

RTST[1] -> P[2] -> R[3] -> Q[3] -> R[3]



Allocation of Activation Records

- ***Skip L_1-L_2+1 records starting from the caller's AR and establish the static link to the AR reached***
- ***L_1 – caller, L_2 – Callee***
- **RTST^1 -> P^2 -> R^3 -> Q^3 -> R^3**
- **Ex: Consider P^2 -> R^3**
- **2-3+1=0; hence the SL of R points to P**
- **Consider R^3 -> Q^3**
- **3-3+1=1; hence skipping one link starting from R, we get P; SL of Q points to P**

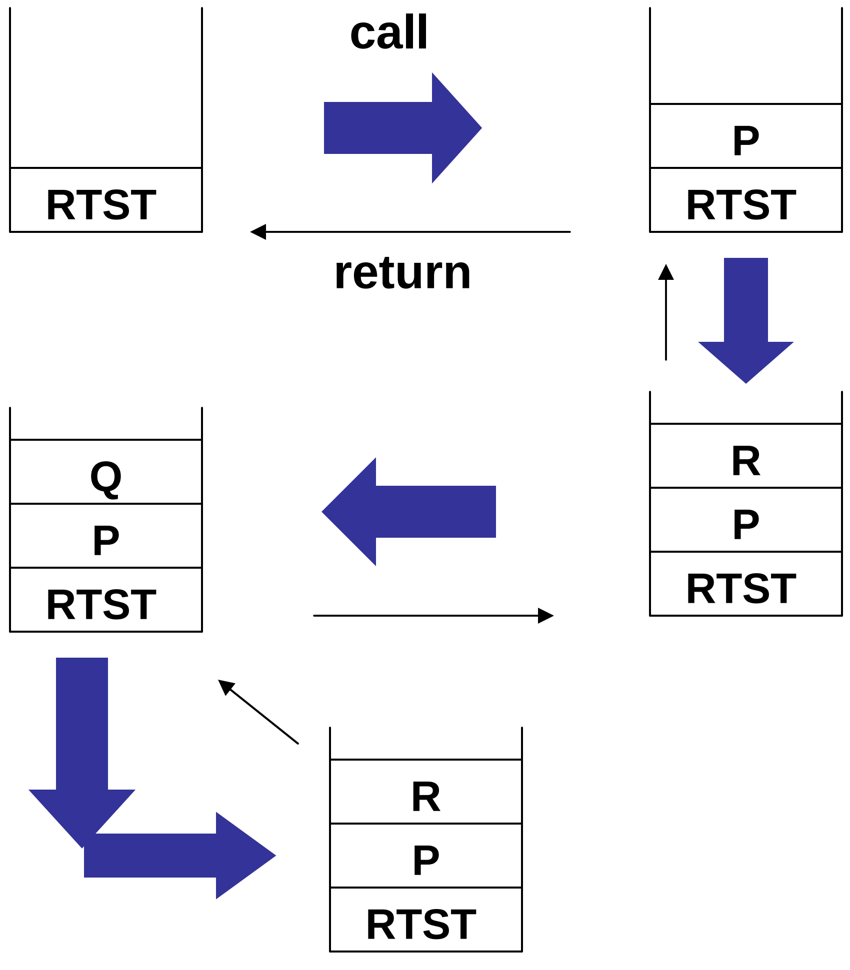


Display Stack of Activation Records

1 program *RTST*;
2 procedure *P*;
3 procedure *Q*;
begin *R*; end
3 procedure *R*;
begin *Q*; end
begin *R*; end
begin *P*; end

Pop L_1-L_2+1 records off the display of the caller and push the pointer to AR of callee (L_1 – caller, L_2 – Callee)

The popped pointers are stored in the AR of the caller and restored to the DISPLAY after the callee returns

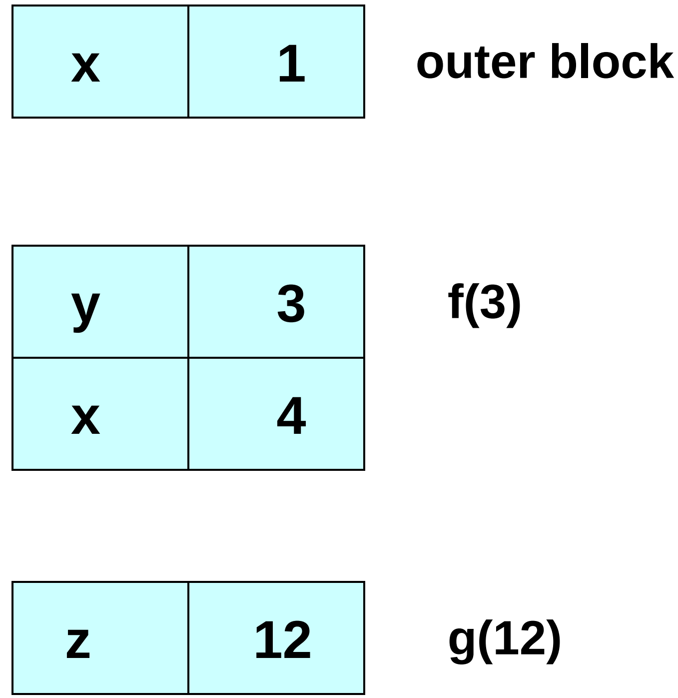


Static Scope and Dynamic Scope

- ***Static Scope***
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- ***Dynamic Scope***
  - A global identifier refers to the identifier associated with the most recent activation record
  - Uses the actual sequence of calls that are executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1;
function g(z) = x+z;
function f(y) = {
    int x = y+1;
    return g(y*x)
};
f(3);
```



Stack of activation records after the call to *g*

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf("%f",r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf("\n");
show(); small(); printf("\n");
}
```

- Under static scoping, the output is

  0.25 0.25

  0.25 0.25
- Under dynamic scoping, the output is

  0.25 0.125

  0.25 0.125

Implementing Dynamic Scope – Deep Access Method

- Use *dynamic link* as *static link*
- Search activation records on the stack to find the first AR containing the non-local name
- The depth of search depends on the input to the program and cannot be determined at compile time
- Needs some information on the identifiers to be maintained at runtime within the ARs
- Takes longer time to access globals, but no overhead when activations begin and end

Implementing Dynamic Scope – Shallow Access Method

- Allocate some static storage for ***each*** name
- When a new AR is created for a procedure ***p***, a local name ***n*** in ***p*** takes over the static storage allocated to name ***n***
- The previous value of ***n*** held in static storage is saved in the AR of ***p*** and is restored when the activation of ***p*** ends
- Direct and quick access to globals, but some overhead is incurred when activations begin and end

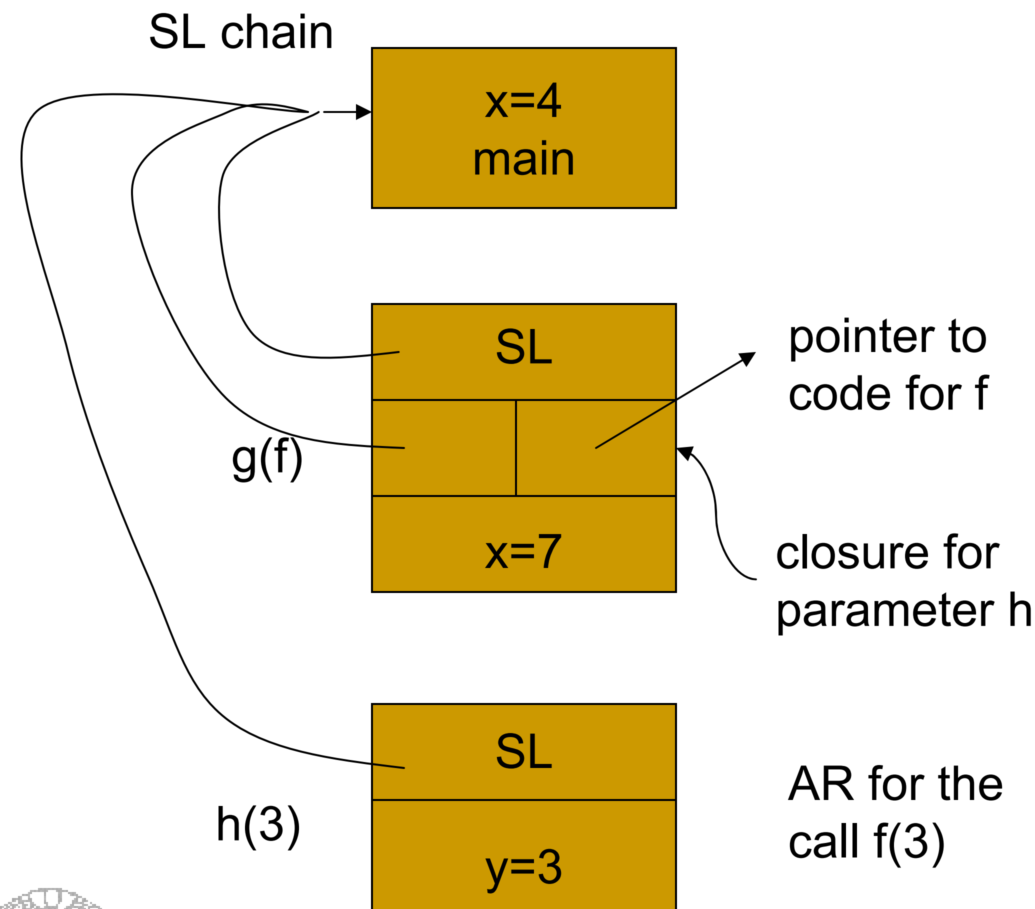
Passing Functions as Parameters

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);//returns 12
}
```

- A language has first-class functions if functions can be
  - declared within any scope
  - passed as arguments to other functions
  - returned as results of functions
- In a language with first-class functions and static scope, a function value is generally represented by a closure
  - a pair consisting of a pointer to function code
  - a pointer to an activation record
- Passing functions as arguments is very useful in structuring of systems using upcalls

Passing Functions as Parameters – Implementation with Static Scope



An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);// returns 12
}
```

Passing Functions as Parameters – Implementation with Static Scope

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

- In this example, when executing the call h(3), h is really f and 3 is the parameter y of f
- Without passing a closure, the AR of the main program cannot be accessed, and hence, the value of x within f will not be 4
- When f is passed as a parameter in the call g(f), a closure consisting of a pointer to the code for f and a pointer to the AR of the main program is passed
- When processing the call h(3), after setting up an AR for h (i.e., f), the SL for the AR is set up using the AR pointer in the closure for f that has been passed to the call g(f)

Heap Memory Management

- Heap is used for allocating space for objects created at run time
  - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
  - *malloc()* and *free()* in C programs
  - *new()* and *delete()* in C++ programs
  - *new()* and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)