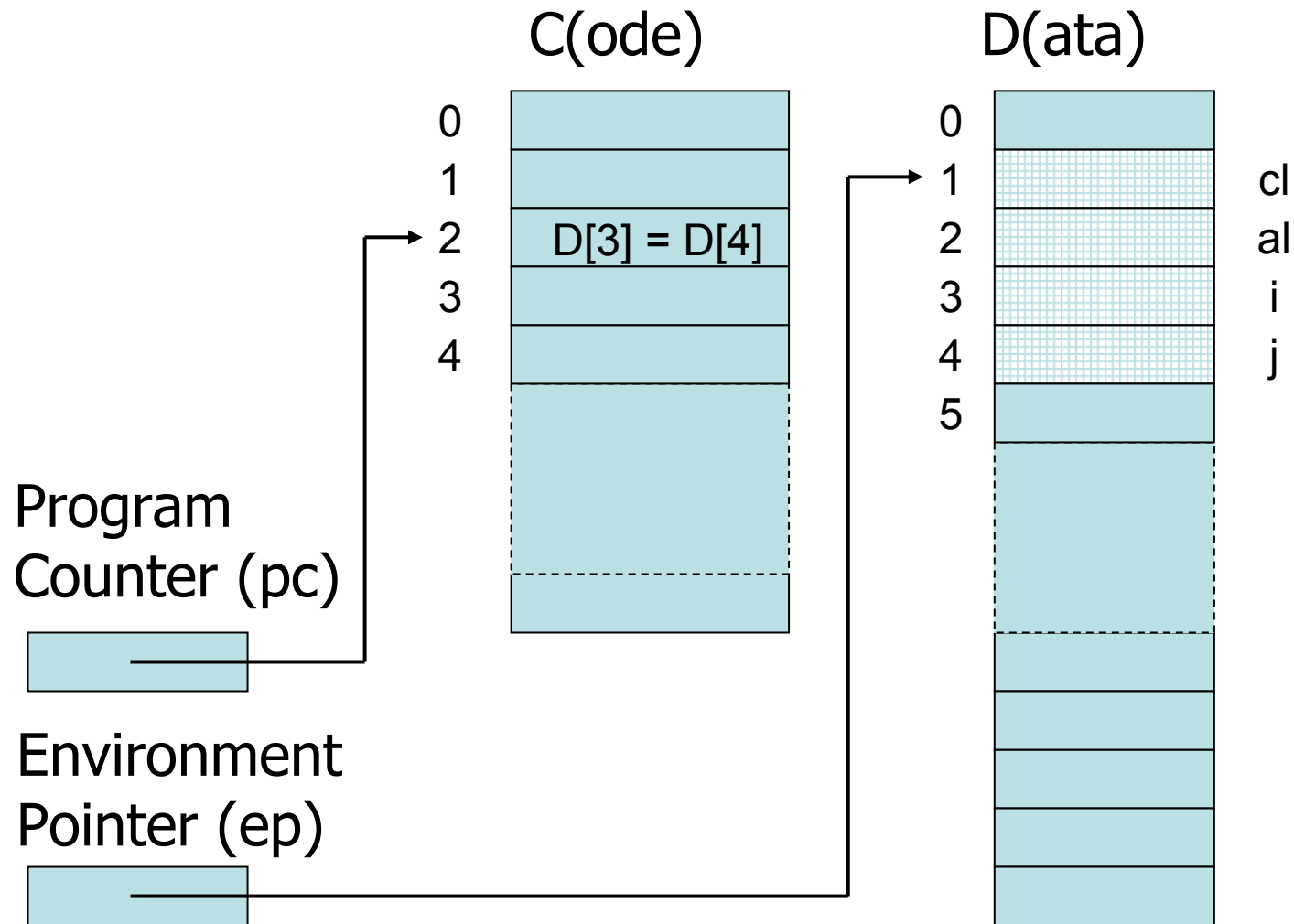
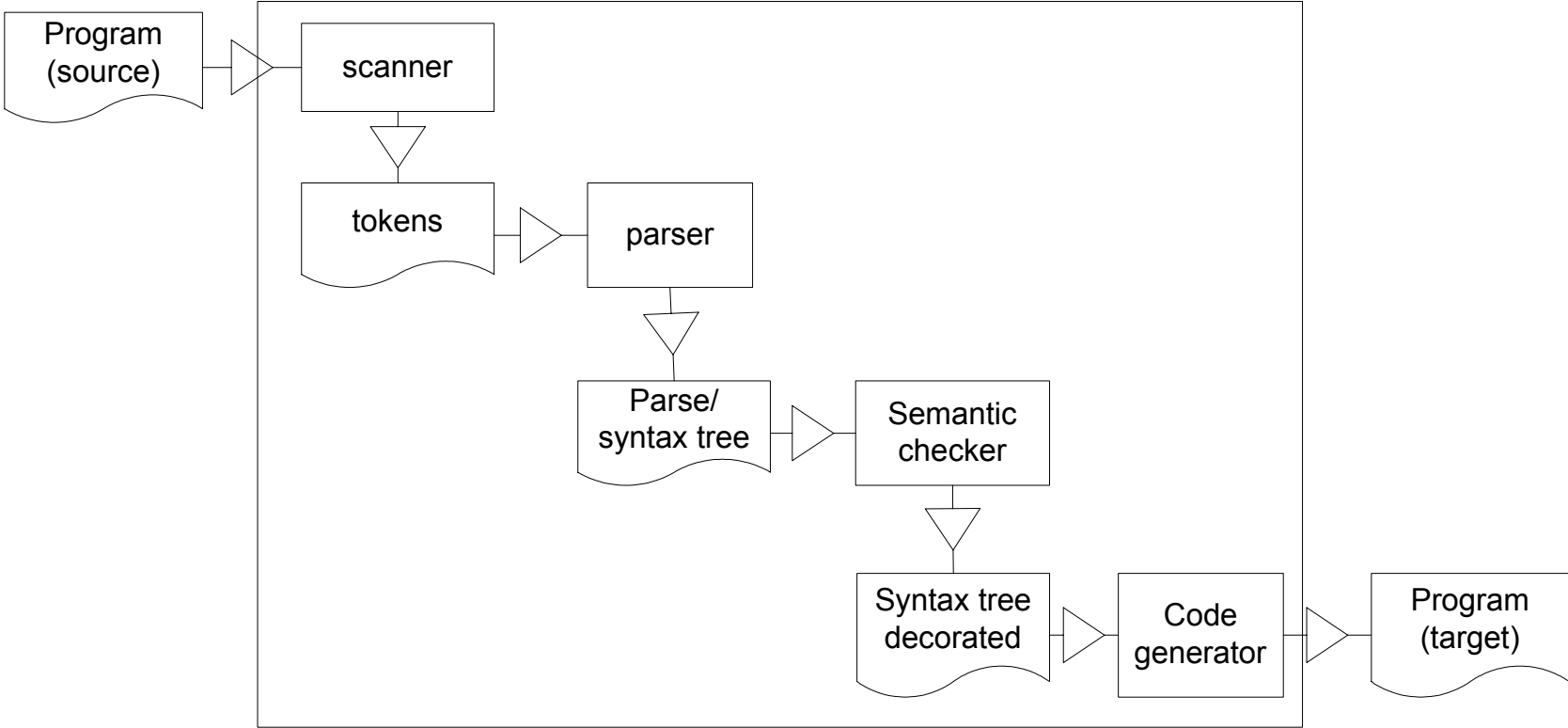


Details of virtual machine operation

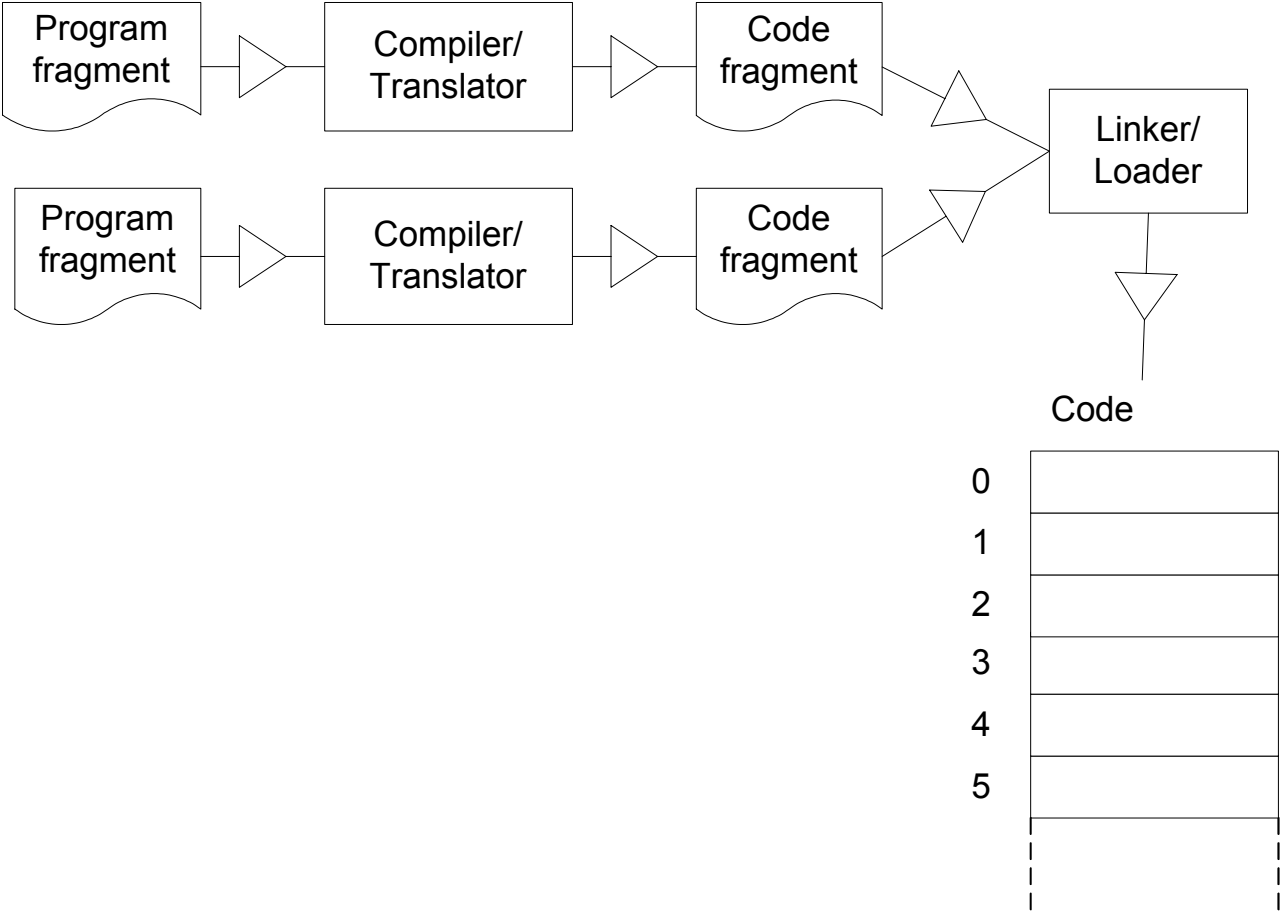
$i = j$



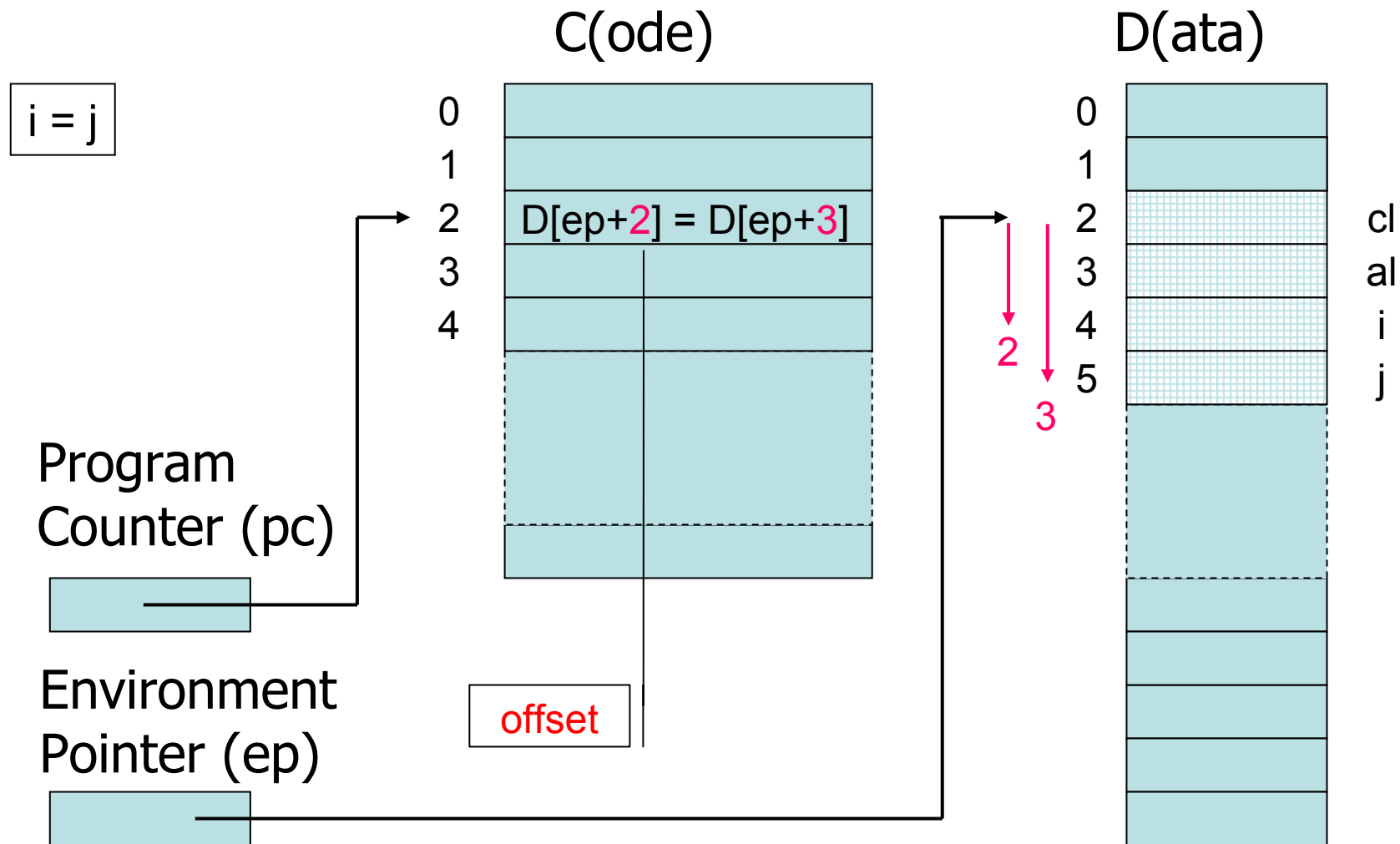
Compiler



Separate compilation



Relocation/offsets

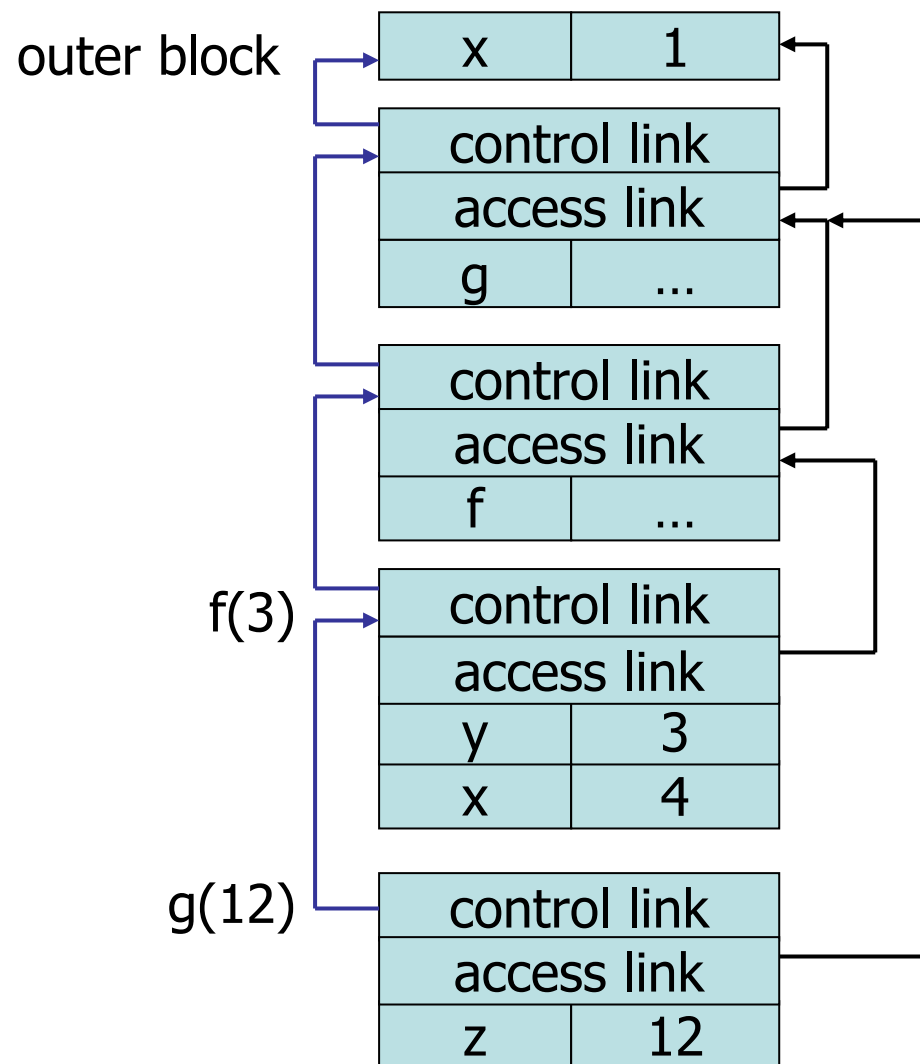


Static scope with access links

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

Use access link to find global variable:

- Access link is always set to frame of closest enclosing lexical block
- For function body, this is block that contains function declaration



Static Block Level (BL) and Context Vector (CV)

Blocks

```
{ int x=1 ...
  { ...
    { ...;
      x; ...
    }
  }
  { ...
    { ...;
      x;
    }
  }
}
```

BL

1

2

3

2

3

```
void makeContextVector() {
  int i;
  CV[ep.BL] = ep;
  for (i = ep.BL; i >= 2; i--) {
    CV[i-1] = CV[i].AL;
  }
}
```

where AL is the Access Link
(Static Link).

CV[0] always denotes the main
program block, and is thereby
constant.

Issues for first-order functions

- Access to global variables ✓
- Parameter passing
 - pass-by-value
 - pass-by-reference
 - pass-by-name
- Assignment

```
int a = 5;

void f(int x)
{
    x = x+1;
}

void main()
{
    f(a);
    print(a);
}
```

Parameter passing

- Pass-by-value
 - Caller places **R-value** (contents) of actual parameter in activation record
 - Function cannot change value of caller's variable
 - Reduces aliasing (alias: two names refer to same loc)
- Pass-by-reference
 - Caller places **L-value** (address) of actual parameter in activation record
 - Function can assign to variable that is passed
 - Aliasing
- Pass-by-name
 - Actual parameter expression is passed as such and evaluated whenever the formal parameter is used in the function

L-value and R-value

- Assignment `y := x+3`
 - Identifier on left refers to location, called its L-value
 - Identifier on right refers to contents, called R-value
- dereferencing

C

```
int x = 5;
```

```
int* px;
```

```
...
```

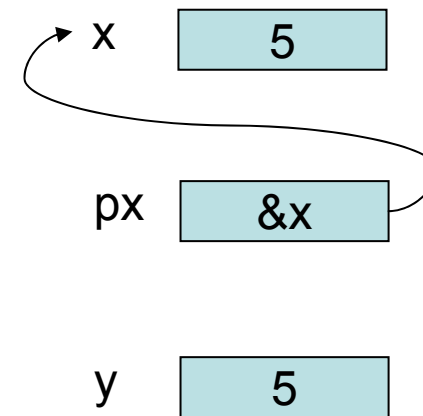
```
px = &px;
```

```
...
```

```
int y = *px
```

```
*px = 0
```

L R



Call-by-copy

- Call by value
 - Local variable assigned at call
 - `f(int in x){...}`
- Call by result
 - Local variable not assigned at call, but returned at exit
 - `f(int out x){...}`
- Call by value-result
 - Local variable assigned at call, and returned at exit
 - `f(int in-out x){...}`

`f.x = a`

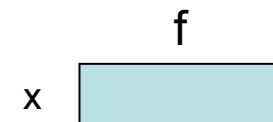
...

...
`a = f.x`

`f.x = a;`

...

`a = f.x`



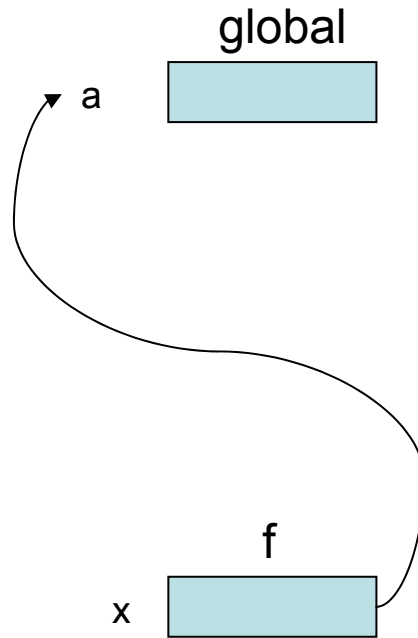
```
int a = 5;

void f(int x)
{
    x = x+1;
}

void main()
{
    f(a);
    print(a);
}
```

by-reference

```
int a = 5;
void f(int x)
{
  x = x+1;
}
void main()
{
  f(a);
  print(a);
}
```



- The 'x' (within f) is set to the address of 'a' (L-value).
- The assignment 'x+1' assigns the value of 'x+1', that is 6, to the variable whose L-value is kept by 'x', i.e. the global variable 'a'.
- 'a' is therefore changed to 6, and 6 is printed.

If 'a' has offset 3 in Data, then 'x' of 'f' is set to 3.
Access to 'x' in 'f' is translated to: (start of global) + 3

Example - aliasing

```
void pour(real& v1, real& v2, real& v) {  
    v1 = v1 - v;  
    v2 = v2 + v;  
};
```

```
real x, y, z;
```

```
x = 4.0;  
y = 6.0;  
z = 1.0;
```

```
pour (x, y, z);
```

```
x = 3.0;  
y = 7.0;
```

```
pour(x, y, x)
```

```
x = 0.0;  
y = 7.0;
```

```
pour (a[i], a[j], a[k]);
```

By name

```
begin integer i;  
  integer procedure sum(i,j);  
    integer i,j;  
  begin  
    integer sm; sm:= 0;  
    for i = 1 step 1 until 100 do  
      sm := sm + j;  
    sum:= sm  
  end;  
  print(sum(i,i*10))  
end;
```

x := x ???

integer procedure p; begin p := p end;

```
swap(int a, b) {  
  int temp;  
  temp = a;  
  a = b;  
  b = temp;  
};
```

```
i=3;  
a[3]=6;  
swap(i, a[i]);
```

-- i = 6

-- a[3] = 6

-- a[6] = 3

```
temp = i;  
i = a[i];  
a[i] = temp;
```

by name

“4.7.3.2. **Name replacement (call by name)**. Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved. “

What is the difference between this and macro expansion?

```
int i; int a[];
swap(int a, b) {
  int i;
  i = a;
  a = b;
  b = i;
};

swap(i, a[i]);
```

```
i=3;
a[3]=6;
swap(i, a[i]);
```

By name

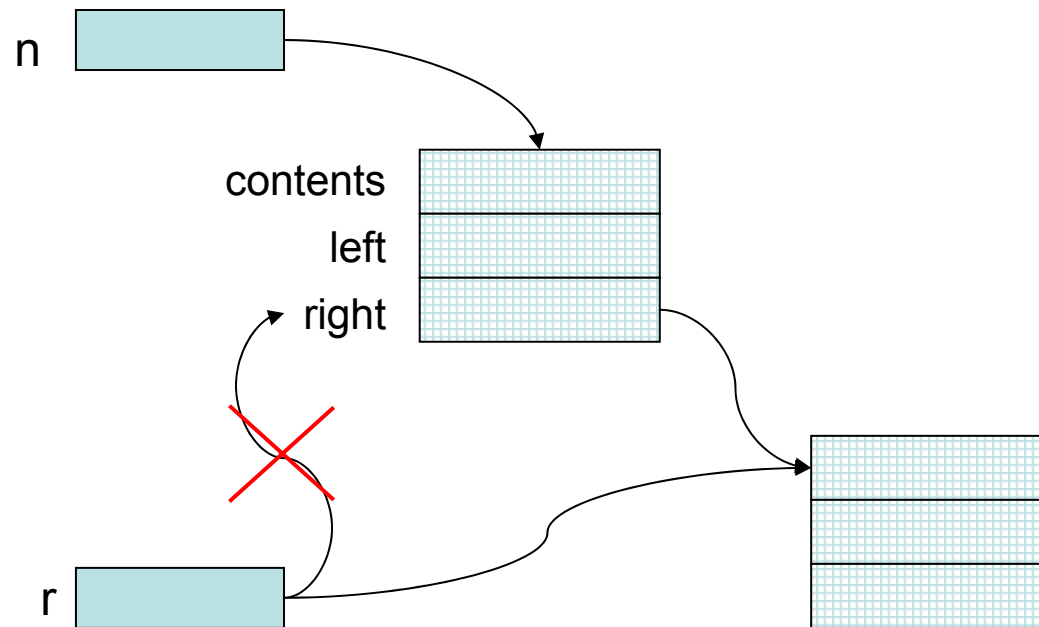
```
-- i = 6
-- a[3] = 6
-- a[6] = 3
```

By macro expansion

```
i = i;      -- local i=0
i = a[i];   -- local i=a[0]
a[i] = i;   -- a[0]=0
```

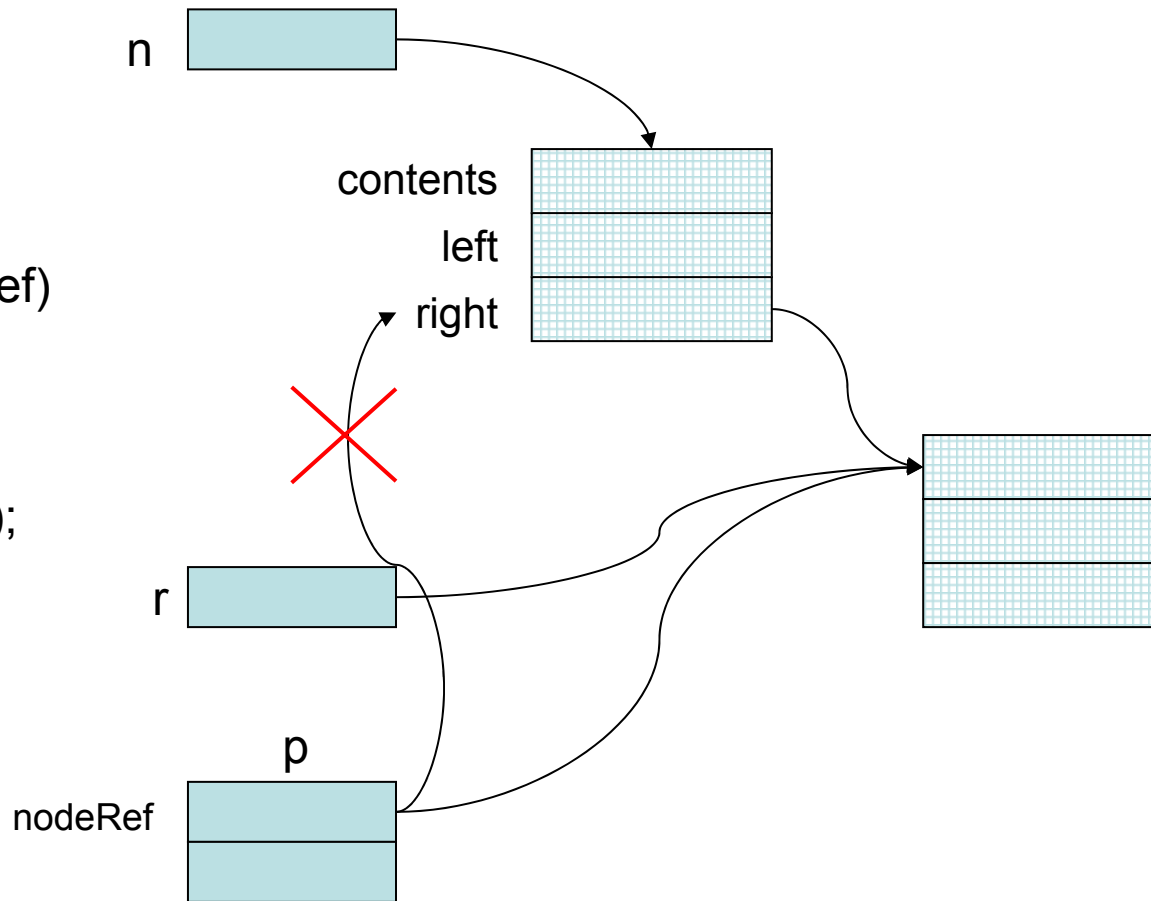
Java: References = L-values ?

```
class main {  
  class Node {  
    Object contents  
    Node left, right;  
  };  
  Node n, r;  
  ...  
  n= new Node();  
  n.right= new  
  Node();  
  ...  
  r= n.right  
  ...  
};
```



Java: by value or by reference

```
class main {  
  class Node {  
    Object contents  
    Node left, right;  
  };  
  Node n, r;  
  void p(Node nodeRef)  
  {... nodeRef ...}  
  ...  
  n= new Node();  
  n.right= new Node();  
  ...  
  r= n.right  
  ...  
  p(n.right)  
};
```



Assignment: Copy semantics vs reference semantics

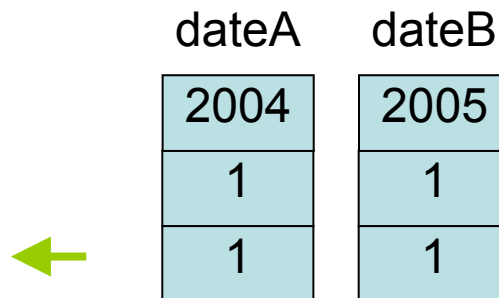
- What happens when a composite value is assigned to a variable of the same type?
 - Copy semantics: All components of the composite value are copied into the corresponding components of the composite variable.
 - Reference semantics: The composite variable is made to contain a pointer (or reference) to the composite value.
- C/C++ adopt copy semantics.
- Java adopts copy semantics for primitive values, but reference semantics for objects.

Example: copy semantics C, C++

```
class Date {  
    int y, m, d;  
};
```

```
Date dateA = {2004, 1, 1};  
Date dateB;
```

```
dateB = dateA;  
dateB.y = 2005;
```

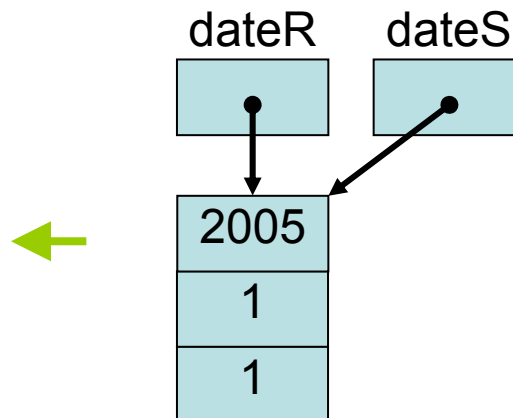


Example: Java reference semantics

```
class Date {  
    int y, m, d;  
    public Date (int y, int m, int d) { ... }  
}
```

```
Date dateR = new Date(2004, 1, 1);  
Date dateS = new Date(2004, 12, 25);
```

```
dateS = dateR;  
dateR.y = 2005;
```



...

- We can achieve the *effect* of reference semantics in C by using pointers

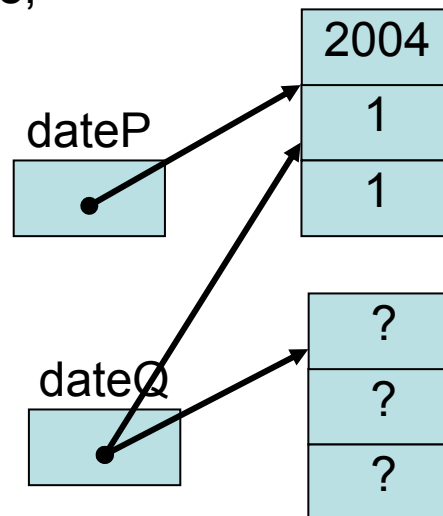
```
class Date {  
    int y, m, d;  
};
```

```
Date* dateP = new Date;  
Date* dateQ = new Date;
```

```
*dateP = dateA;
```

```
dateQ = dateP;
```

dateA	dateB
2004	2005
1	1
1	1



Copy semantics in Java

- We can achieve the *effect* of copy semantics in Java by cloning

```
Date dateR = new Date(2004, 4, 1);  
dateT = dateR.clone();
```

Mappings

$v = f(g(\text{exp1}), h(\text{exp2}, \text{exp3}))$

$\text{int } f(p1, p2) \{ \dots \}$

$(\text{exp1} \rightarrow g, (\text{exp2}, \text{exp3}) \rightarrow h) \rightarrow f \rightarrow v$

$\text{int } f(p1, p2) \{ \dots \}$

$(\text{exp1} \rightarrow g, (\text{exp2}, \text{exp3}) \rightarrow h)$

$\rightarrow f1 \rightarrow f2 \dots fn \rightarrow v$

$\text{int } fn(p1, p2) \{ \dots \}$

$f1(\text{in } p1, p2, \text{out } o1, o2)$

$f2(\text{in } p1, p2, \text{out } o1, o2)$

$\text{exp1} \rightarrow \text{exp2} \rightarrow \dots \rightarrow \text{expn}$

$v1 = v2 = \dots = vn$

$a[\text{exp1}] = \text{exp2}$

Higher-Order Functions

- Language features
 - Functions passed as arguments
 - Functions that return functions from nested blocks
 - Need to maintain environment of function
- Simpler case
 - Function passed as argument
 - Need pointer to activation record “higher up” in stack
- More complicated second case
 - Function returned as result of function call
 - Need to keep activation record of returning function

Why functions as parameters?

```
procedure fsum(f, a, l, u);  
  value l, u; integer array a;  
  integer procedure f;  
begin  
  integer sum:= 0;  
  for i:= l step 1 until l do sum:= sum + f(a[i]);  
  fsum:= sum  
end
```

```
integer array aa[5:100];  
integer procedure ip1(i); value i; integer i; begin ... end;  
integer procedure ip2(i); value i; integer i; begin ... end;
```

```
fsum(ip1, aa, 5, 100)  
fsum(ip2, aa, 5, 100)
```

Pass function as argument

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

There are two declarations of x

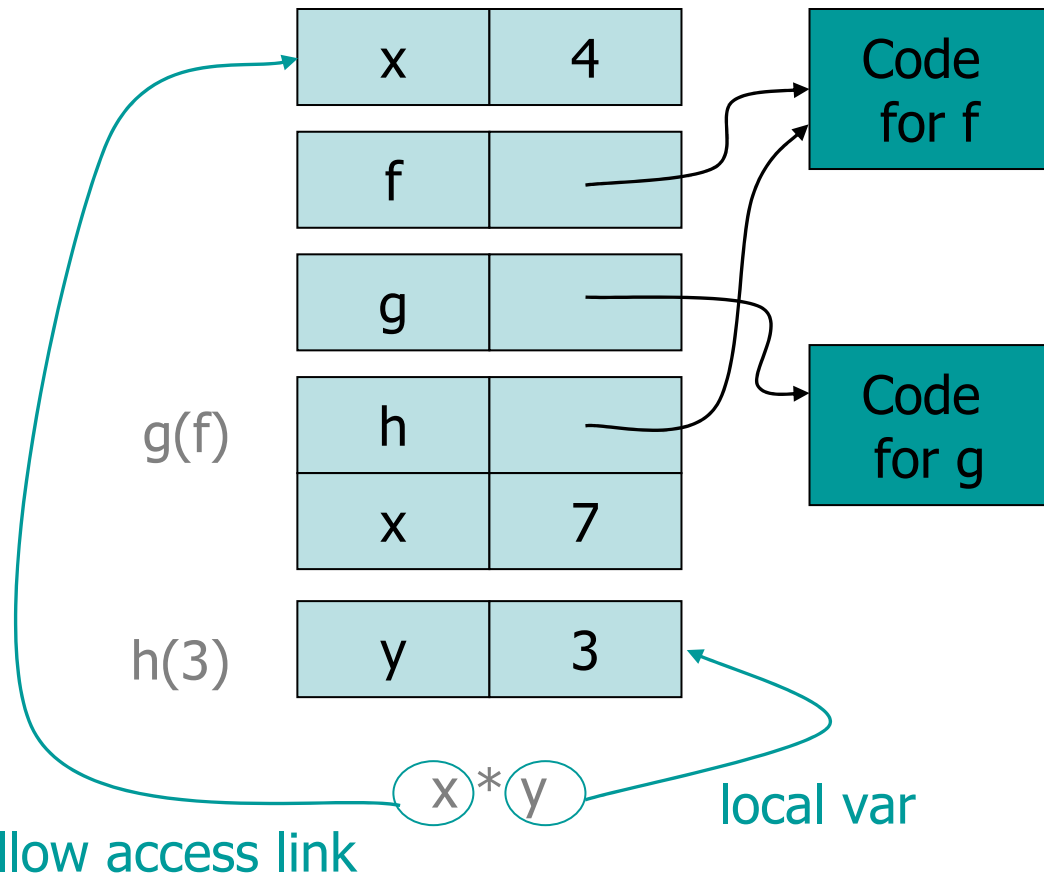
Which one is used for each occurrence of x?

Static Scope for Function Argument

```

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

```



How is access link for $h(3)$ set?

Closures

- Function value is pair *closure* = $\langle env, code \rangle$
- When a function represented by a closure is called,
 - Allocate activation record for call (as always)
 - Set the access link in the activation record using the environment pointer from the closure

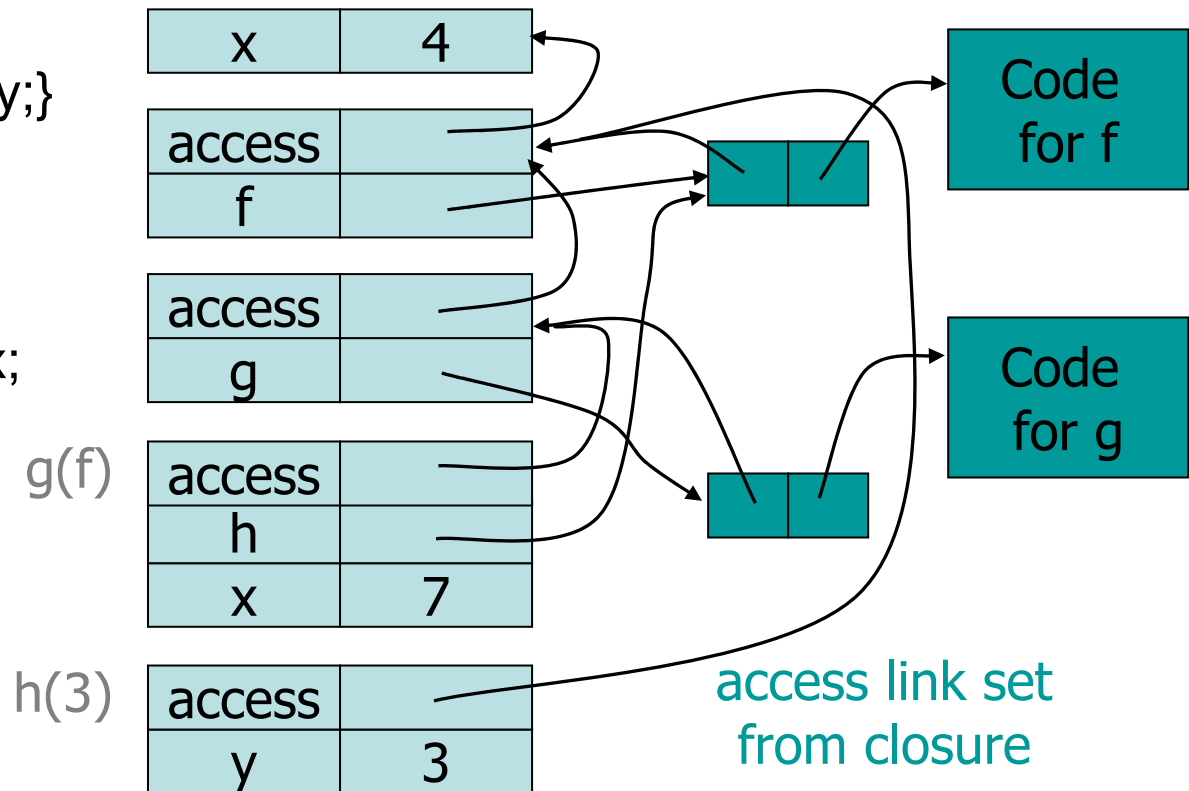
Function Argument and Closures

Run-time stack with access links

```

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

```



access link set from closure

Summary: Function Arguments

- Use closure to maintain a pointer to the static environment of a function body
- When called, set access link from closure
- All access links point “up” in stack
 - May jump past active records to find global variables
 - Still de-allocate active records using stack (LIFO) order

Return Function as Result

- Language feature
 - Functions that return “new” functions
 - Need to maintain environment of function

- Function “created” dynamically
 - expression with free variables
 - values are determined at run time
 - function value is closure = $\langle \text{env}, \text{code} \rangle$
 - code *not* compiled dynamically (in most languages)

Example: Return fctn with private state

```
{int→int mk_counter (int init) {  
    int count = init;  
    int counter(int inc)  
        { return count += inc;}  
    return counter}  
int→int c = mk_counter(1);  
print c(2) + c(2);  
}
```

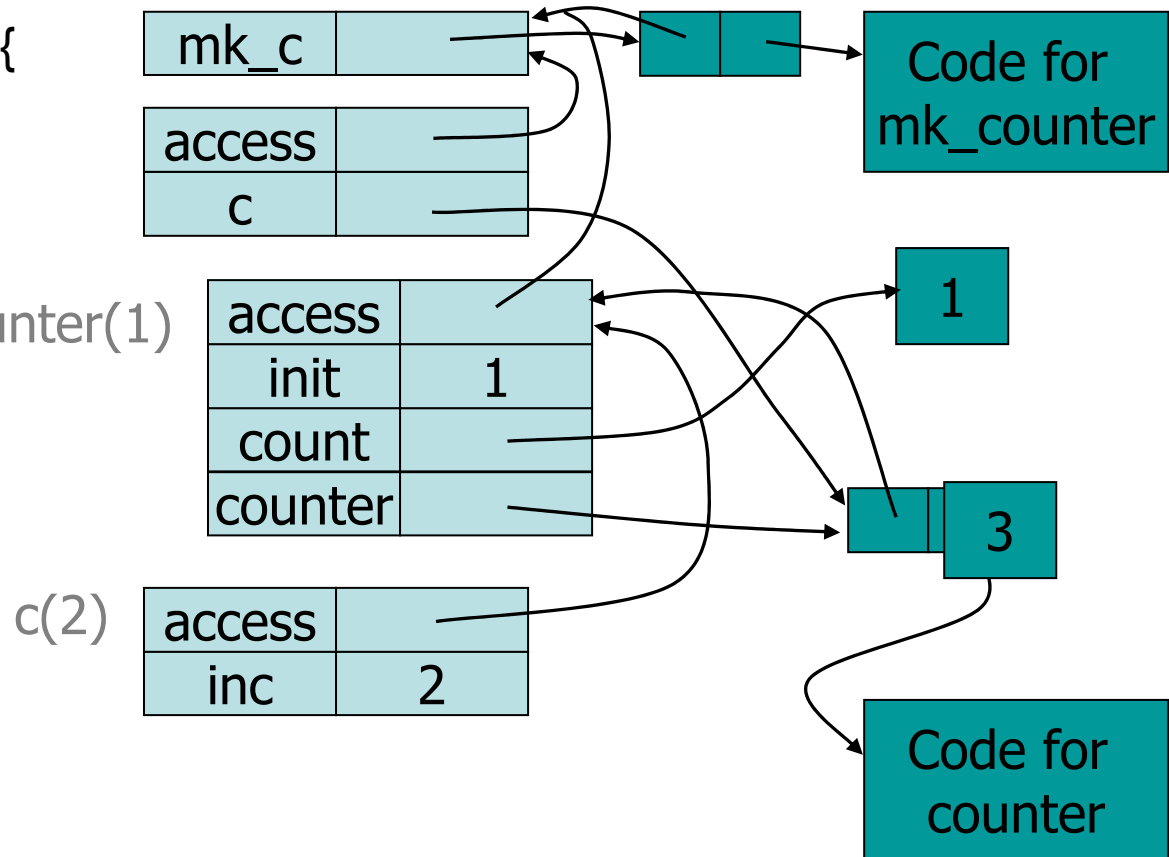
Function to “make counter” returns a closure

How is correct value of count determined in call `c(2)` ?

Function Results and Closures

```

{int→int mk_counter (int init) {
  int count = init;
  int counter(int inc)
    { return count+=inc;}
}
int→int c = mk_counter(1);
print c(2) + c(2);
}
    
```



Call changes cell value from 1 to 3

Summary: Return Function Results

- Use closure to maintain static environment
- May need to keep activation records after return
 - Stack (lifo) order fails!
- Possible “stack” implementation
 - Forget about explicit de-allocation
 - Put activation records on heap
 - Invoke garbage collector as needed
- How to achieve the effect of function parameters in e.g. Java??

```
class CwF {int f(int i) {...} };

fsum(CwF ref, int[] a) {
    int sum= 0;
    for (i= 1, i<aa.length, i++)
        sum= sum + ref.f(a[i]);
    return sum
}

int aa[] = new aa[95];
int ip1(int i); {...};
int ip2(int i); {...};
```

```
class CwFip1 {
    int f(int i) {return ip1(i)}
};
class CwFip2 {
    int f(int i) {return ip2(i)}
};

CwFip1 reflp1 = new CwFip1;
CwFip1 reflp2 = new CwFip2;

fsum(reflp1, aa)
fsum(reflp2, aa)
```