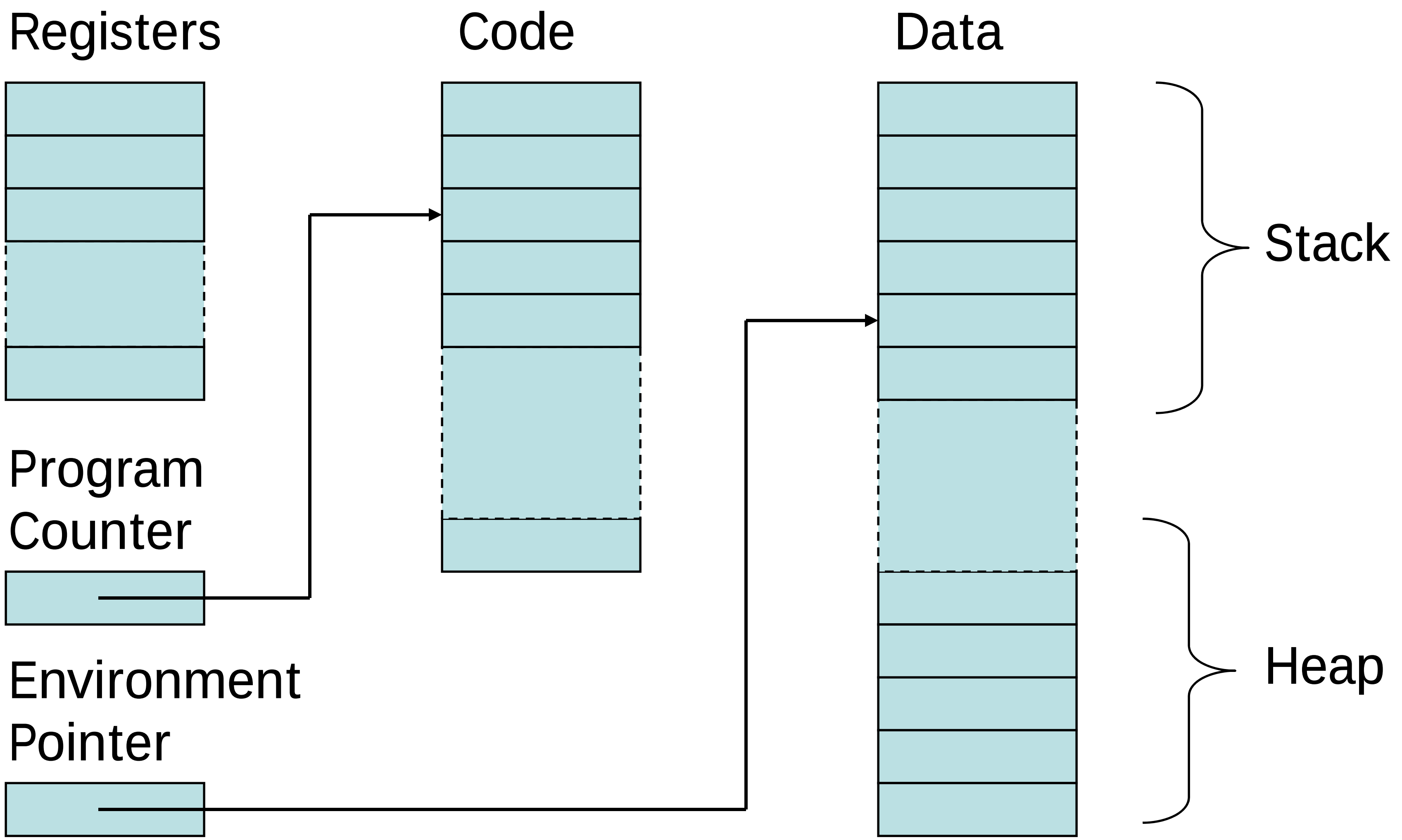
Runtime Organization I & II

- Block-structured languages and stack organization
- In-line Blocks
  - activation records
  - storage for local and global variables
- First-order functions
- Parameter passing

- Higher-order functions
- Heap Organization

Simplified Machine Model



INF 3110 - 2009

Block-Structured Languages

- Nested blocks

```
...;
{ int x = 2;
    { int y = 3;
      x = y+2;
    }
};
...
```

new variables declared in nested blocks

outer block

inner block

x: global variable
y: local variable

  – Storage management
    - Enter block: allocate space for variables
    - Exits block: space may be de-allocated

Examples

- Blocks in common languages
  - C/C++/Java { … }
  - Algol/Simula begin … end
  - ML let … in … end
- Two forms of blocks to start with
  - In-line blocks
  - Blocks associated with functions or procedures
  - To come: blocks associated with classes

```
class Node
{   Object contents
    Node left, right;
    insert(Node n)
    { ...
    }
};
```

Some basic concepts

- Scope
  – Region of program text where declaration is visible
- Lifetime
  – Period of time when location is allocated
- Declaration - application

```
{ int x = … ;
    {  int y = … ;
        {  int x = … ;
              …; x; …
         };
    };
};
```

– Inner declaration of x hides outer one.

– Called “hole in scope”

– Lifetime of outer x includes time when inner block is executed

– Lifetime ≠ scope

In-line blocks

- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables

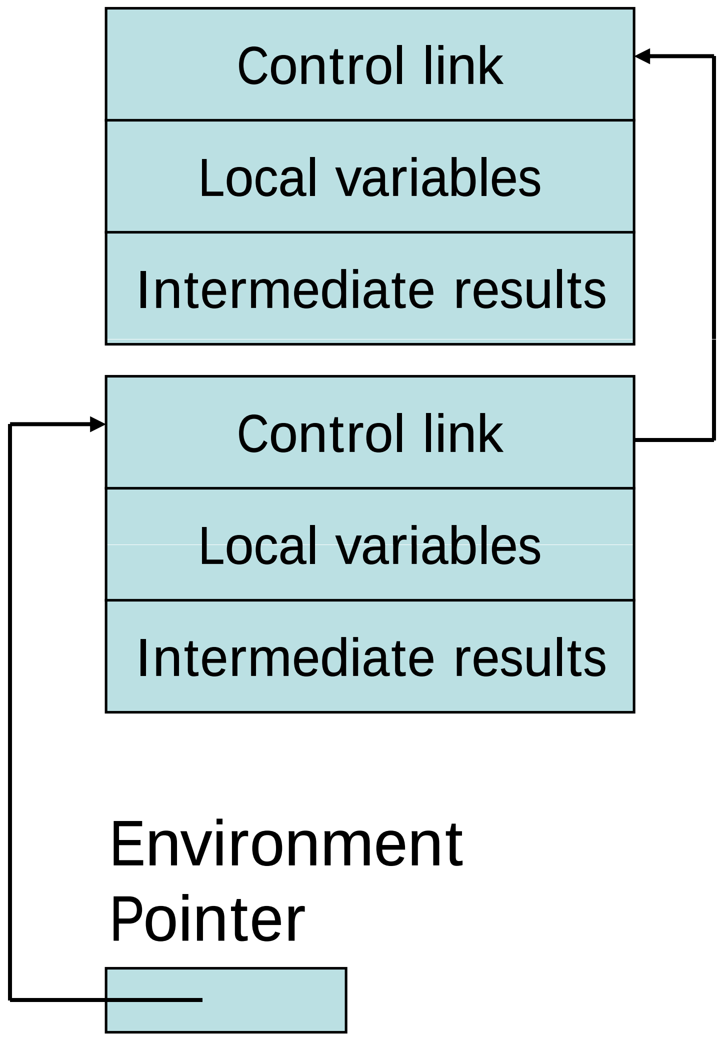
```
{  int x = 0;
   int y = x+1;
     {  int z = (x+y)*(x-y);

     };
};
```

Push record with space for x, y
Set values of x, y
  Push record for inner block
  Set value of z
  Pop record for inner block
Pop record for outer block

May need space for variables and intermediate results like (x+y), (x-y)

INF 3110 - 2009

Activation record for in-line block



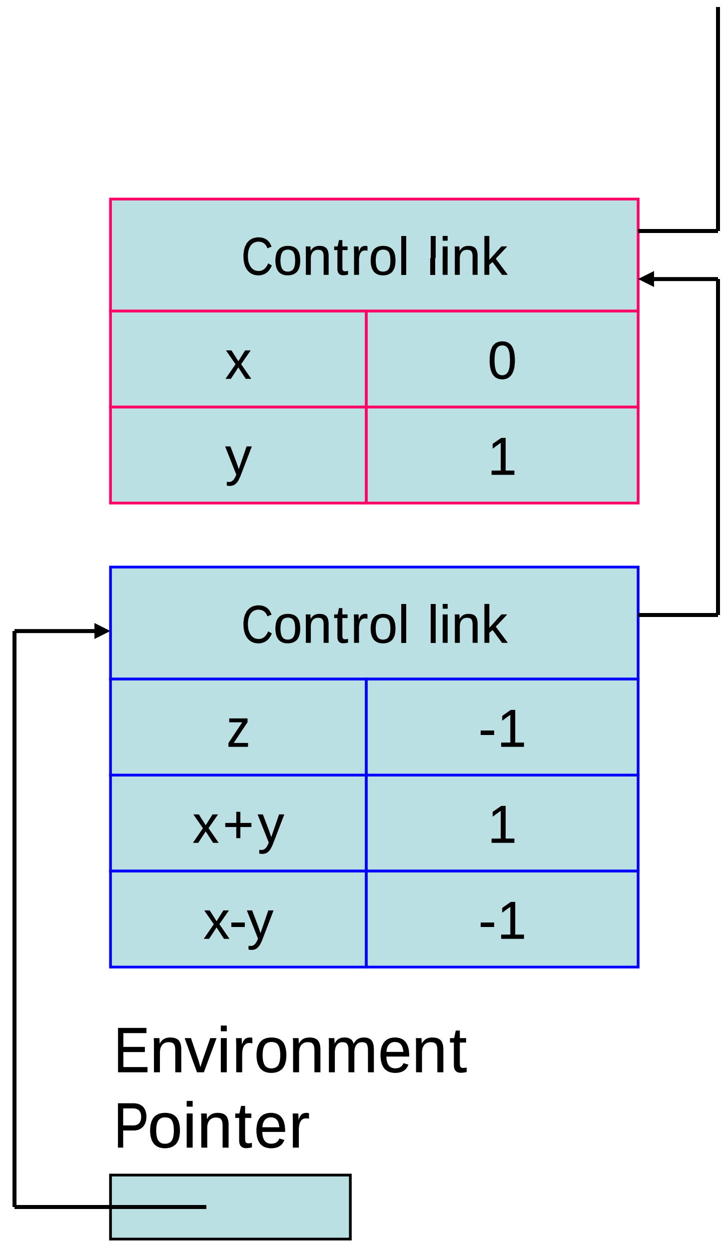
- Control link (dynamic link)
  - pointer to previous record on stack
- Push record on stack:
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

INF 3110 - 2009

Example

```
{ int x = 0;
   int y = x+1;
      {  int z = (x+y)*(x-y);


      };
};
```

- Push record with space for x, y
- Set values of x, y
  - Push record for inner block
  - Set value of z
  - Pop record for inner block
- Pop record for outer block



Initial values

- Specified initial value
- Default initial value

- Languages differ
  - C/C++: no default initial value
    - Undefined?
    - Arbitrary value?
  - Algol/Simula/Java: default initial value

```
{ int x = 0;
  int y;
       {  int z = (x+y)*(x-y);
        };
};
```

INF 3110 - 2009

Scoping rules

- Global and local variables
  - x, y are local to outer block
  - z is local to inner bock
  - x, y are global to inner block

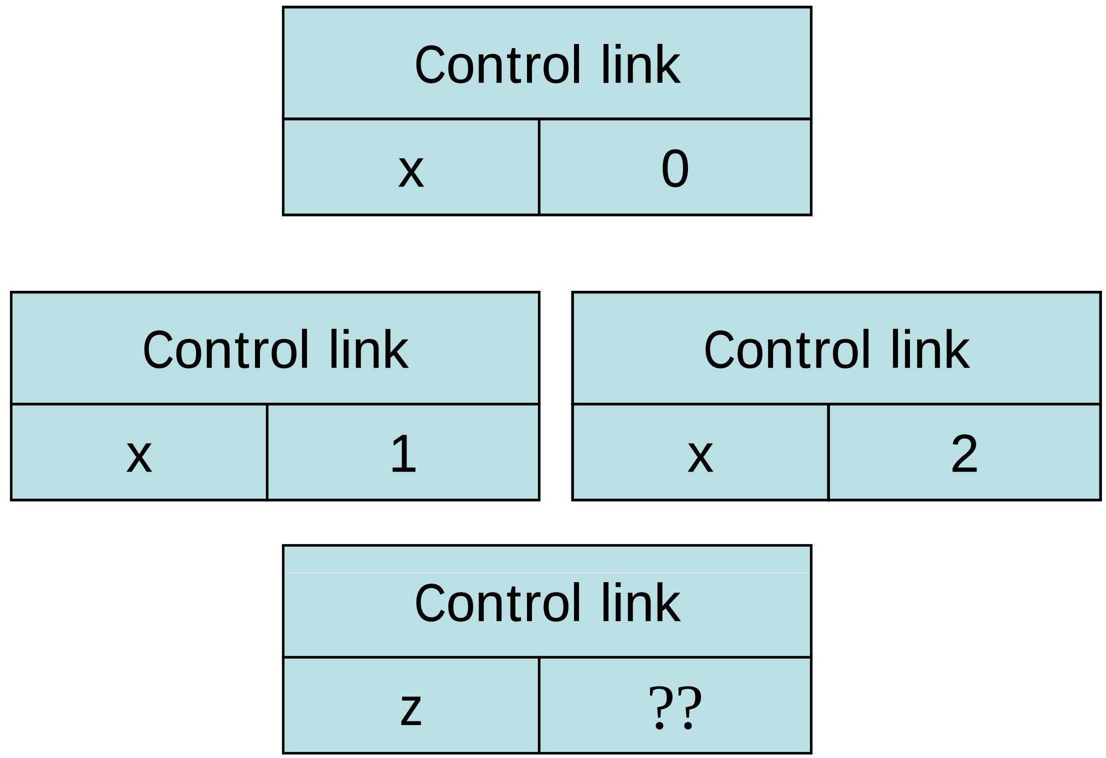
```
{ int x = 0;
  int y = x+1;
     { int z = (x+y)*(x-y);

     };
};
```

- Static scope
  - global refers to declaration in closest enclosing block
- Dynamic scope
  - global refers to most recent activation record

These are the same until we consider function calls.

Example – static/dynamic scoping

```
{
   int x = 0;
   {               -- block 1
     int x = 1;
   };
   {               -- block 2
     int x = 2;
   };
   {               -- block 3
     int z = x;
   }
}
```



As is
z = 0

block 1 executes block 3
Static scope: z = 0
Dynamic scope: z = 1

block 2 executes block 3
Static scope: z = 0
Dynamic scope: z = 2

Functions and procedures

- Syntax of procedures (Algol) and functions (C)

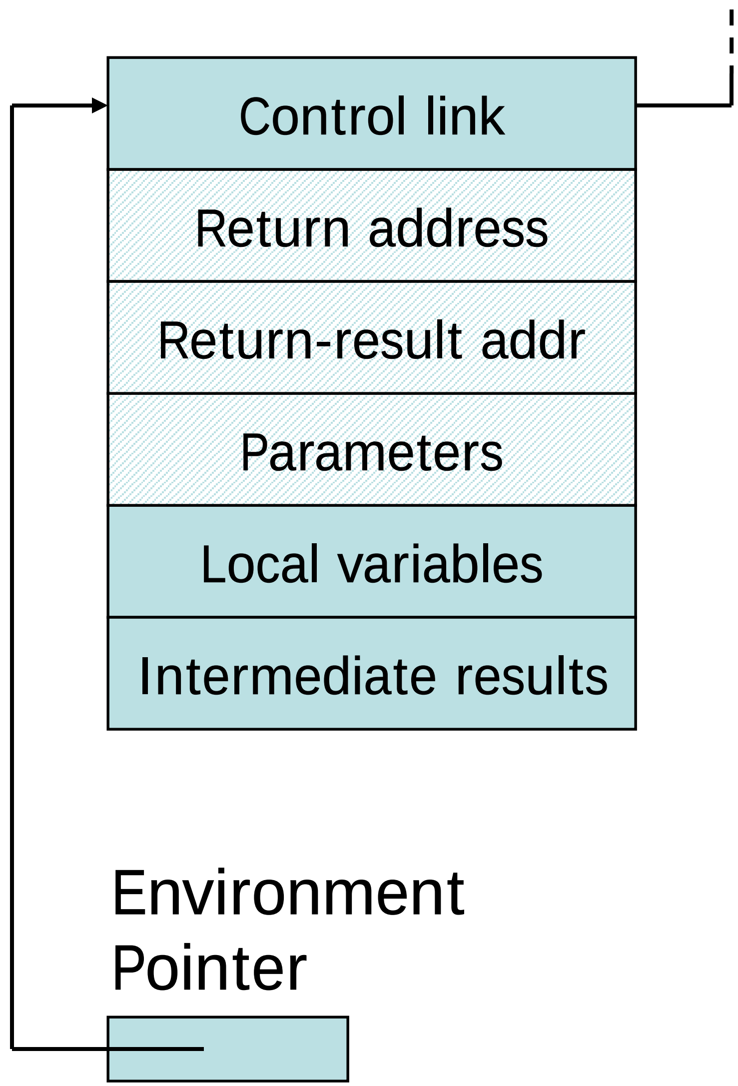
```
procedure P (<params>)          <type> function f(<params>)
begin                           {
    <local variables>               <local variables>
    <proc body>                     <function body>
end;                            };
```

- Activation record must include space for
  - parameters
  - return address
  - return value
    (an intermediate result)
  - location to put return value on function exit

```
int fact(int n) { … }
int i, j;
...
i = 7;
j = fact(i);
print(j)
```

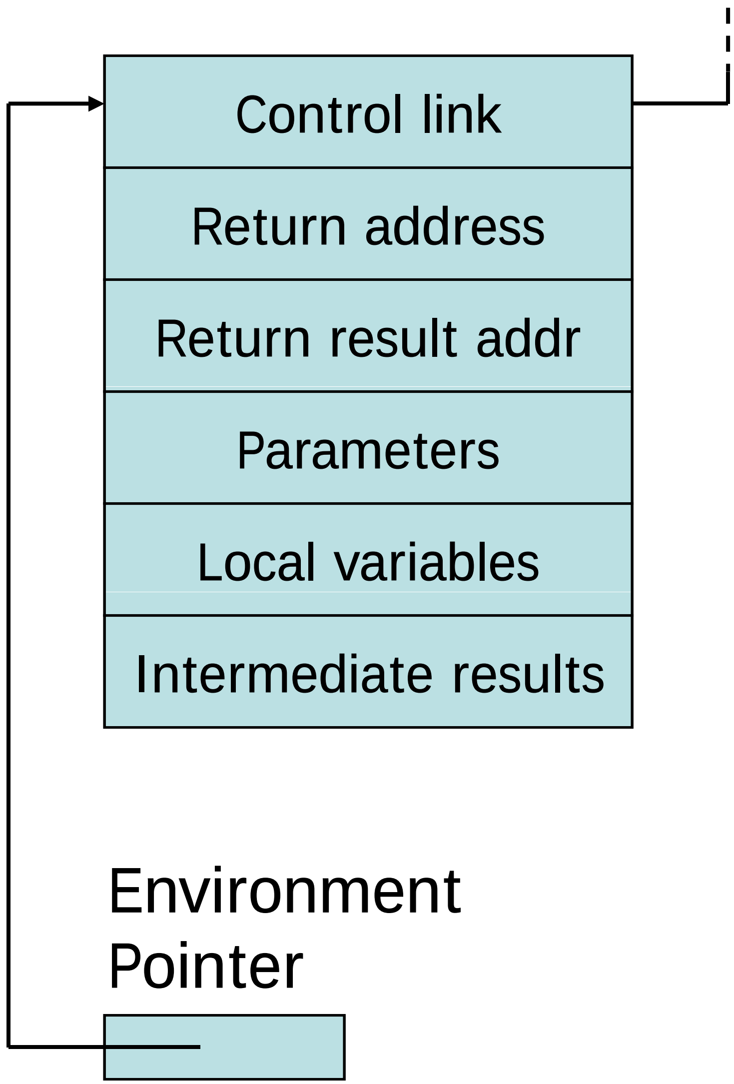
INF 3110 - 2009

Activation record for function



- Return address
  - Location of code to execute on function return
- Return-result address
  - Address in activation record of calling block to receive return value
- Parameters
  - Locations to contain data from calling block

Example



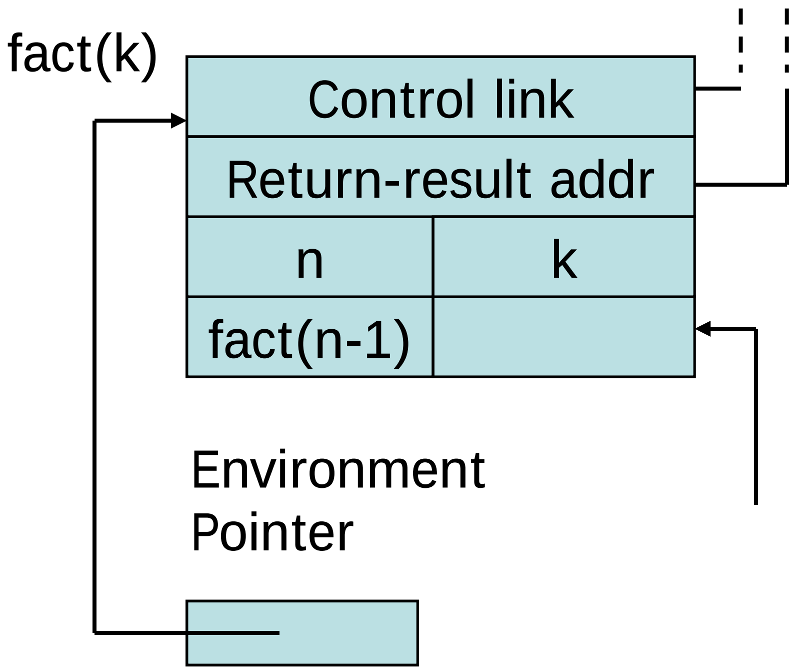
- Function

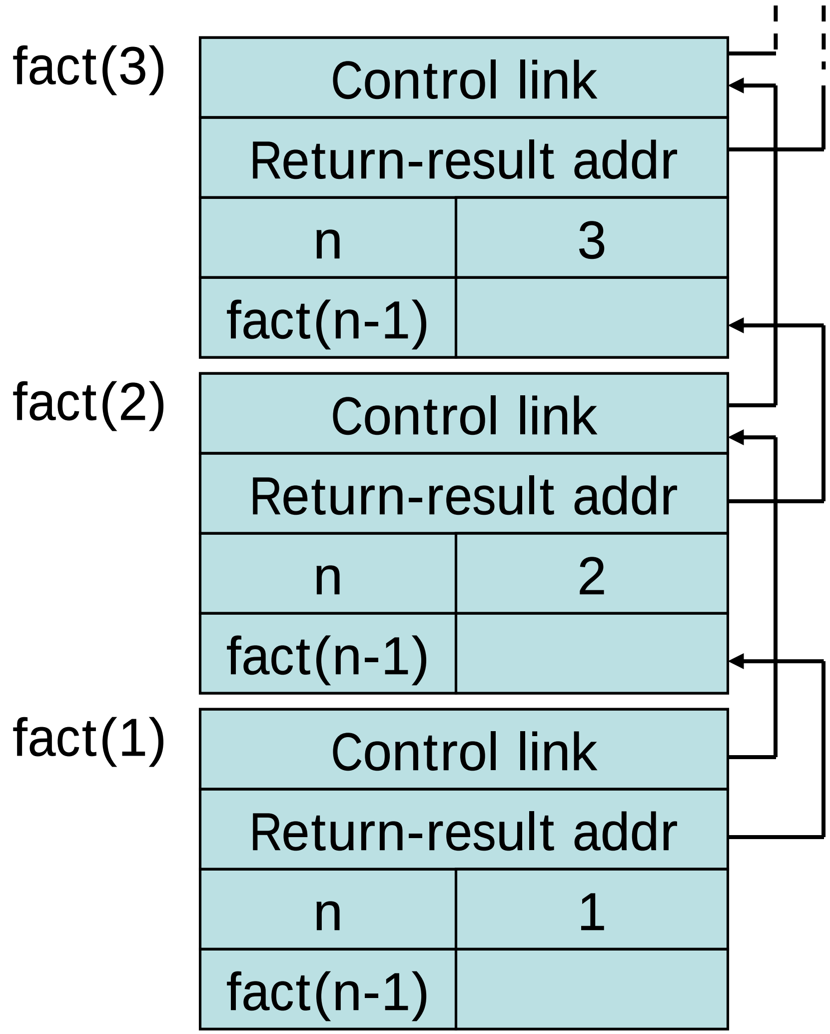
  fact(n) = if n <= 1 then 1

  else n * fact(n-1)

- Return result address
  - location to put fact(n)
- Parameter
  - set to value of n by calling sequence
- Intermediate result
  - locations to contain value of fact(n-1)

Function call



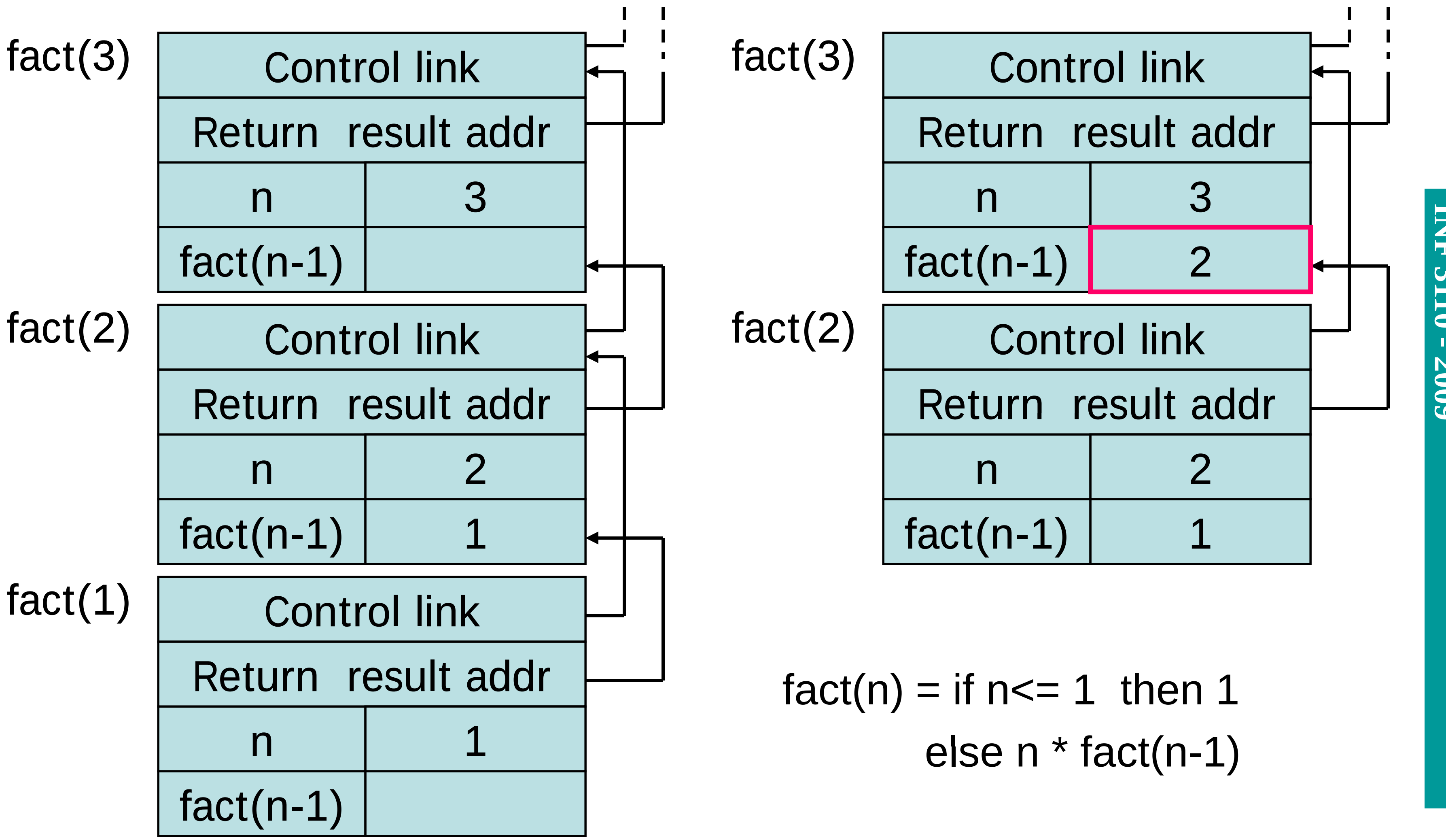


fact(n) = if n<= 1 then 1
else n * fact(n-1)

Return address omitted; would be ptr into code segment

Function return next slide →

Function return



fact(n) = if n<= 1 then 1
else n * fact(n-1)

Access to global variables

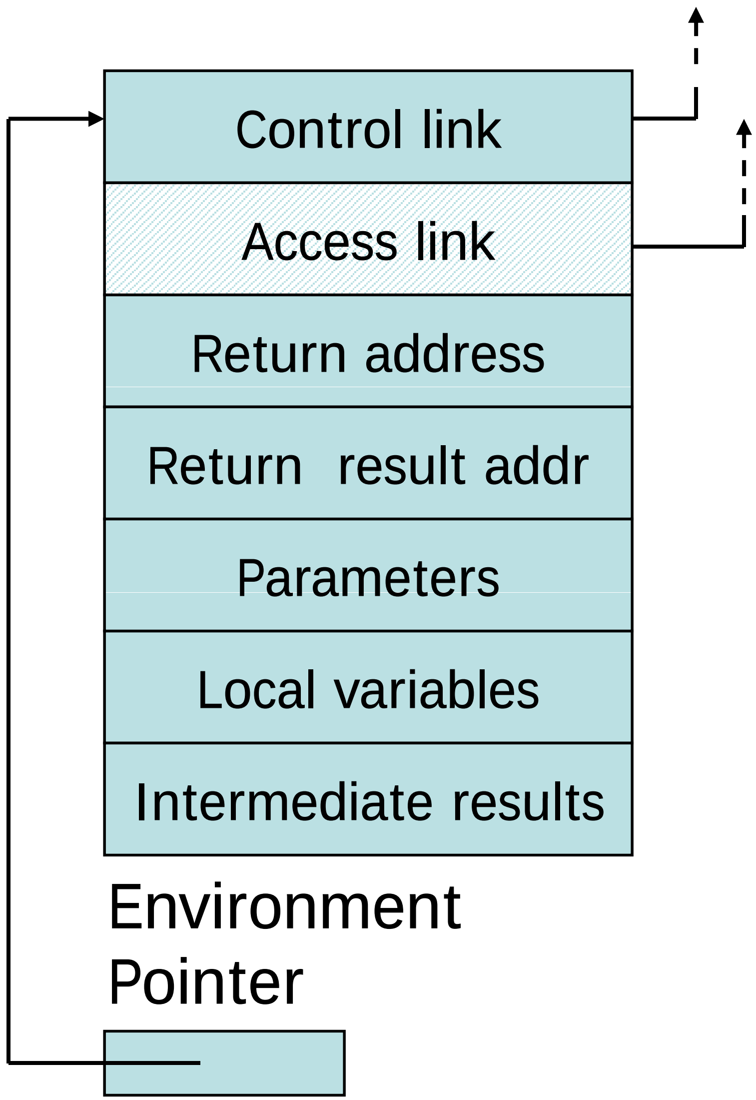
- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example

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

outer block	x	1
f(3)	y	3
	x	4
g(12)	z	12

Which x is used for expression x+z ?

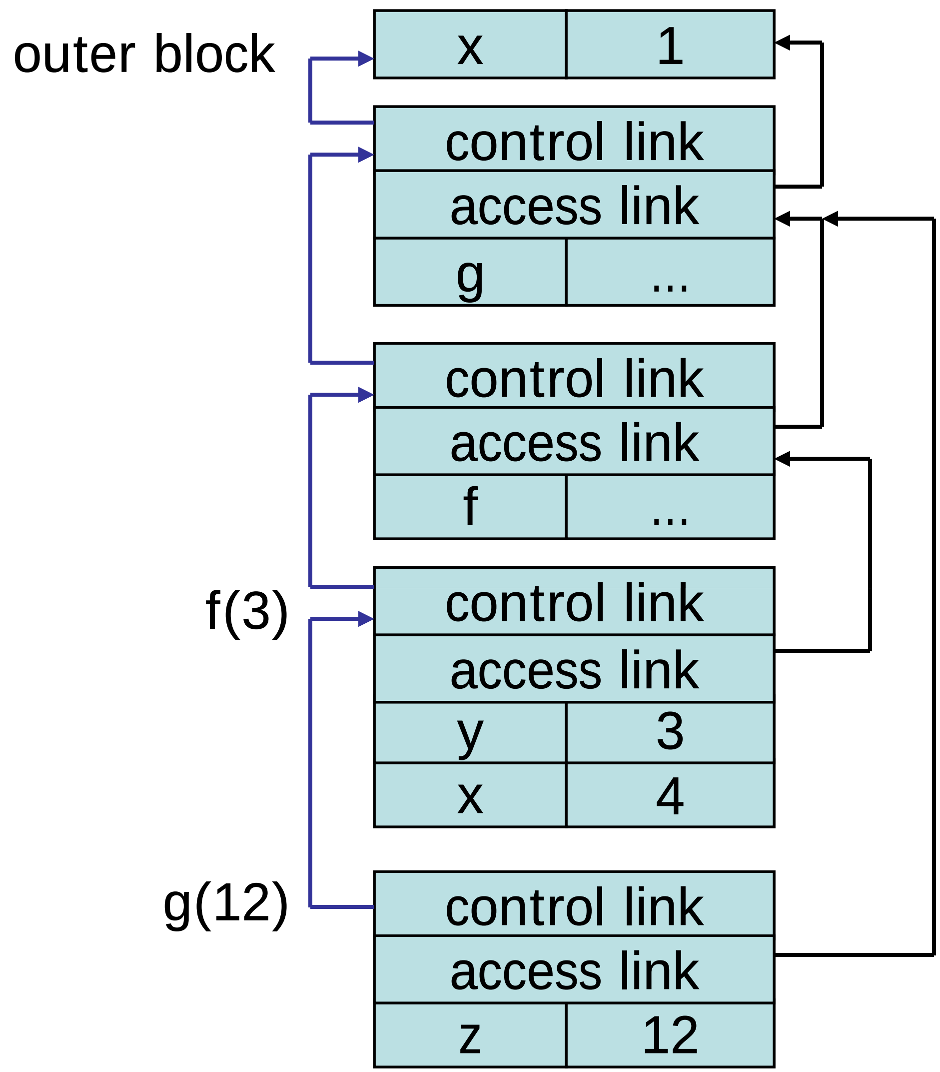
Activation record for static scope



- Control link
  - Link to activation record of previous (calling) block
- Access link (static link)
  - Link to activation record of closest enclosing block in program text
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text

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);
```



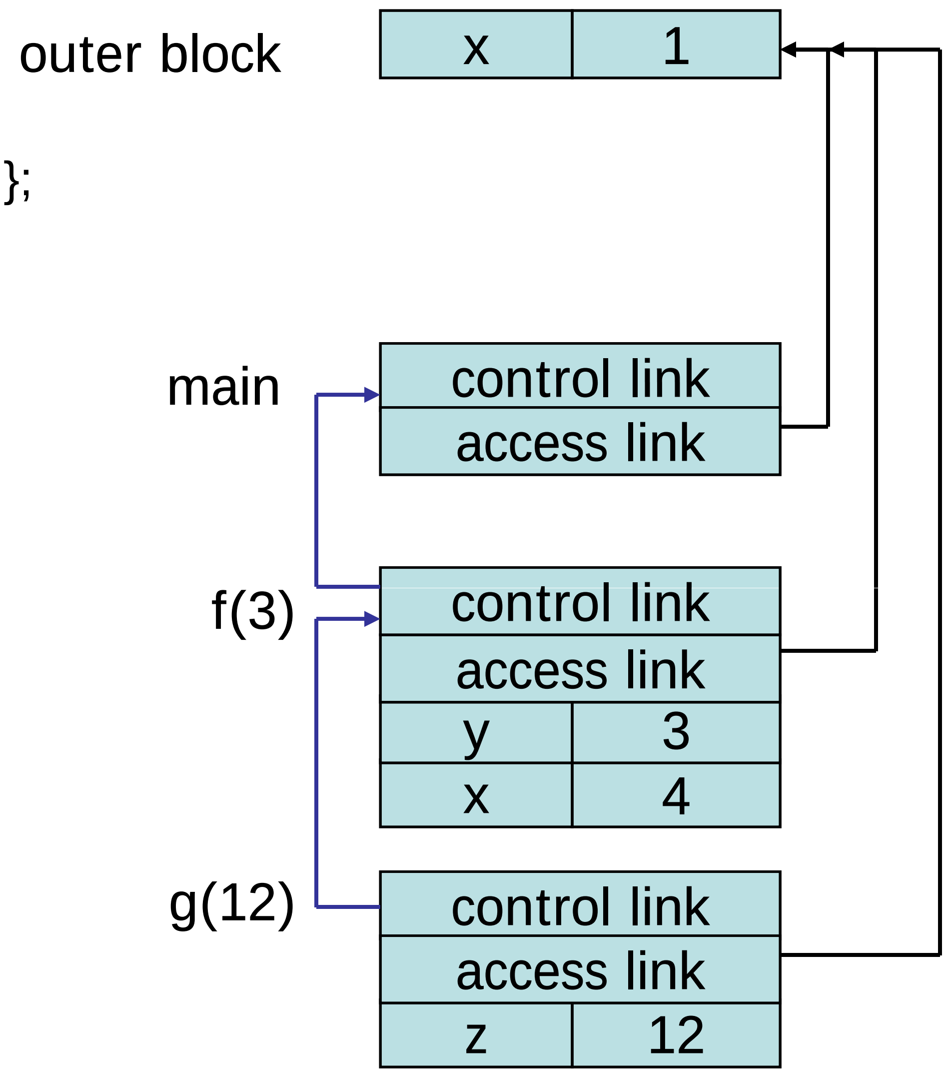
Use access link to find global variable:

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

INF 3110 - 2009

Same in not-ML-like notation

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



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);
}
```

INF 3110 - 2009

Parameter passing

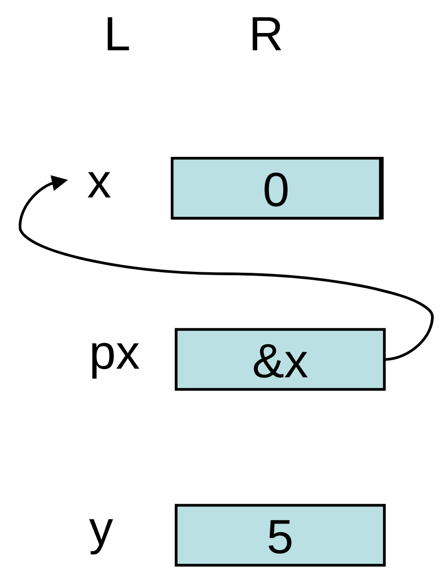
- Pass-by-value
  - Caller places **R-value** (contents) of actual parameter in activation record
  - Function cannot change value of caller's variable
  - Reduces aliasing (alias: two names refer to same location)
- Pass-by-reference
  - Caller places **L-value** (address) of actual parameter in activation 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 = &x;
...

int y = *px

*px = 0
```



Call-by-copy

- Call by value
  - Local variable assigned at call
  - f(int **in** x){...}

f.x = a
...

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

...
a = f.x

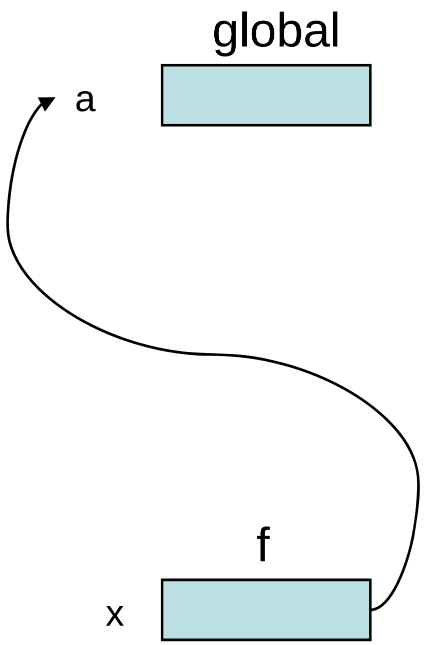
- Call by value-result
  - Local variable assigned at call, and returned at exit
  - f(int **in-out** x){...}

f.x = a;
...
a = f.x


```
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.

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;
```

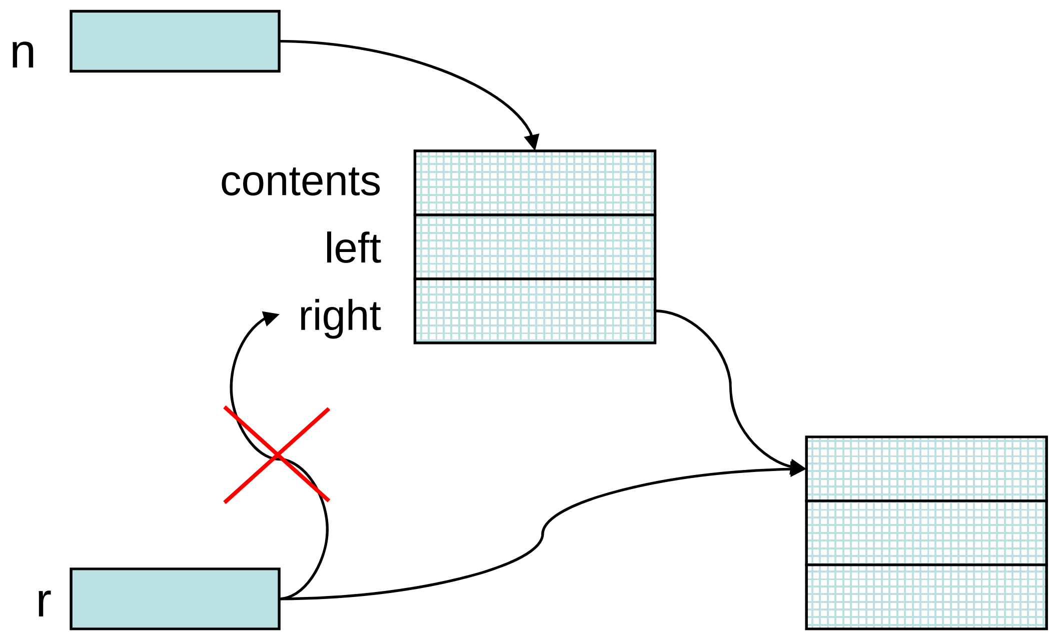
```
pour(x, y, x)
```

```
x = 0.0;
y = 6.0;
```

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

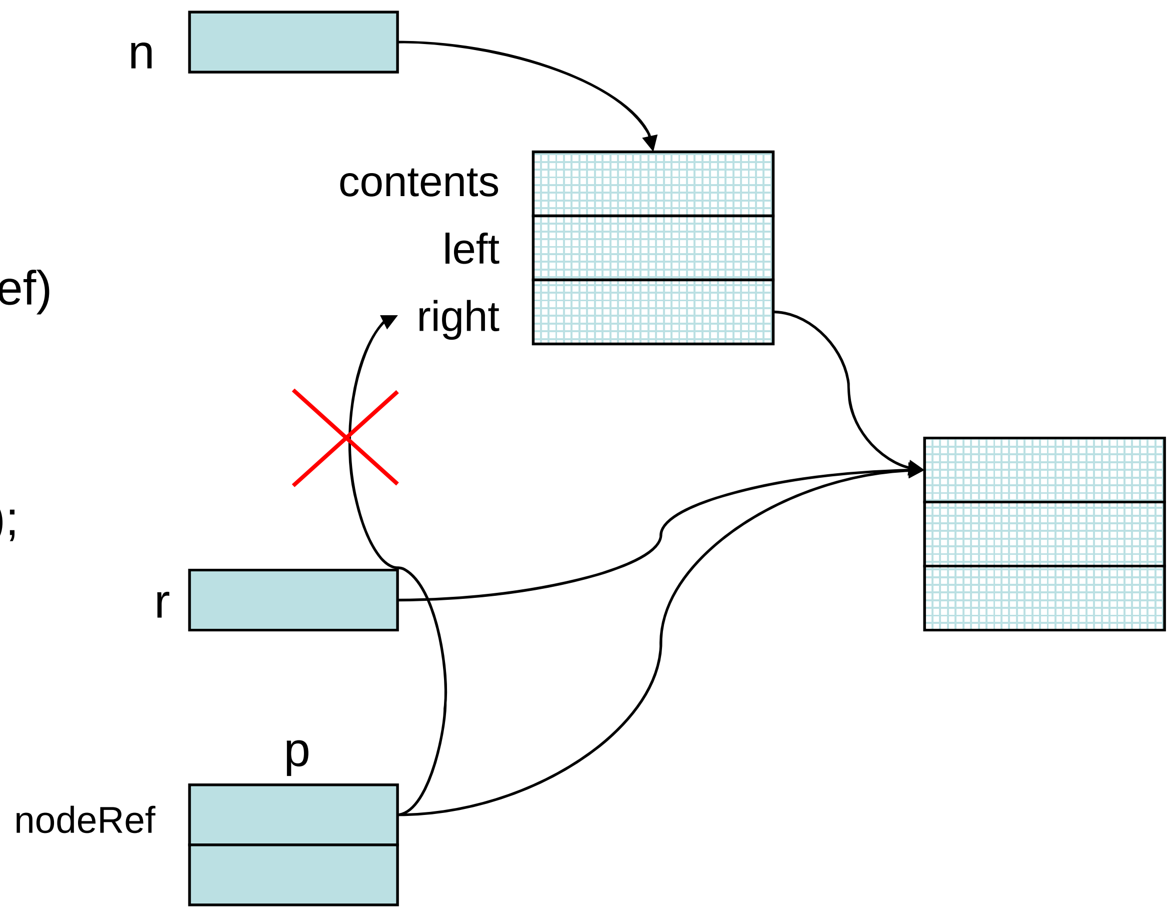
Java: References = L-values ?

```
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

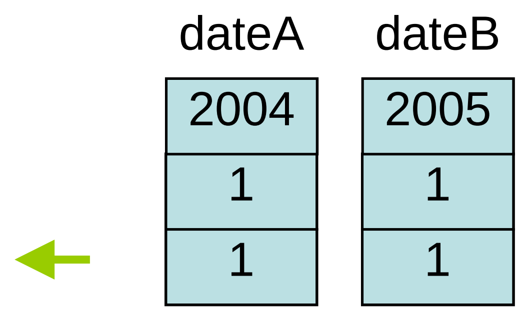
- What happens when a composite value is assigned to a variable of the same type?
  - Copy semantics: All components of the composite value are copied into the corresponding components of the composite variable.
  - Reference semantics: The composite variable is made to contain a pointer (or reference) 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;
```