# INF3110 – Programming Languages
# Scope and Runtime Organization


Eyvind W. Axelsen

eyvinda@ifi.uio.no | @eyvindwa

http://eyvinda.at.ifi.uio.no


Slides adapted from previous years' slides made by Birger Møller-Pedersen

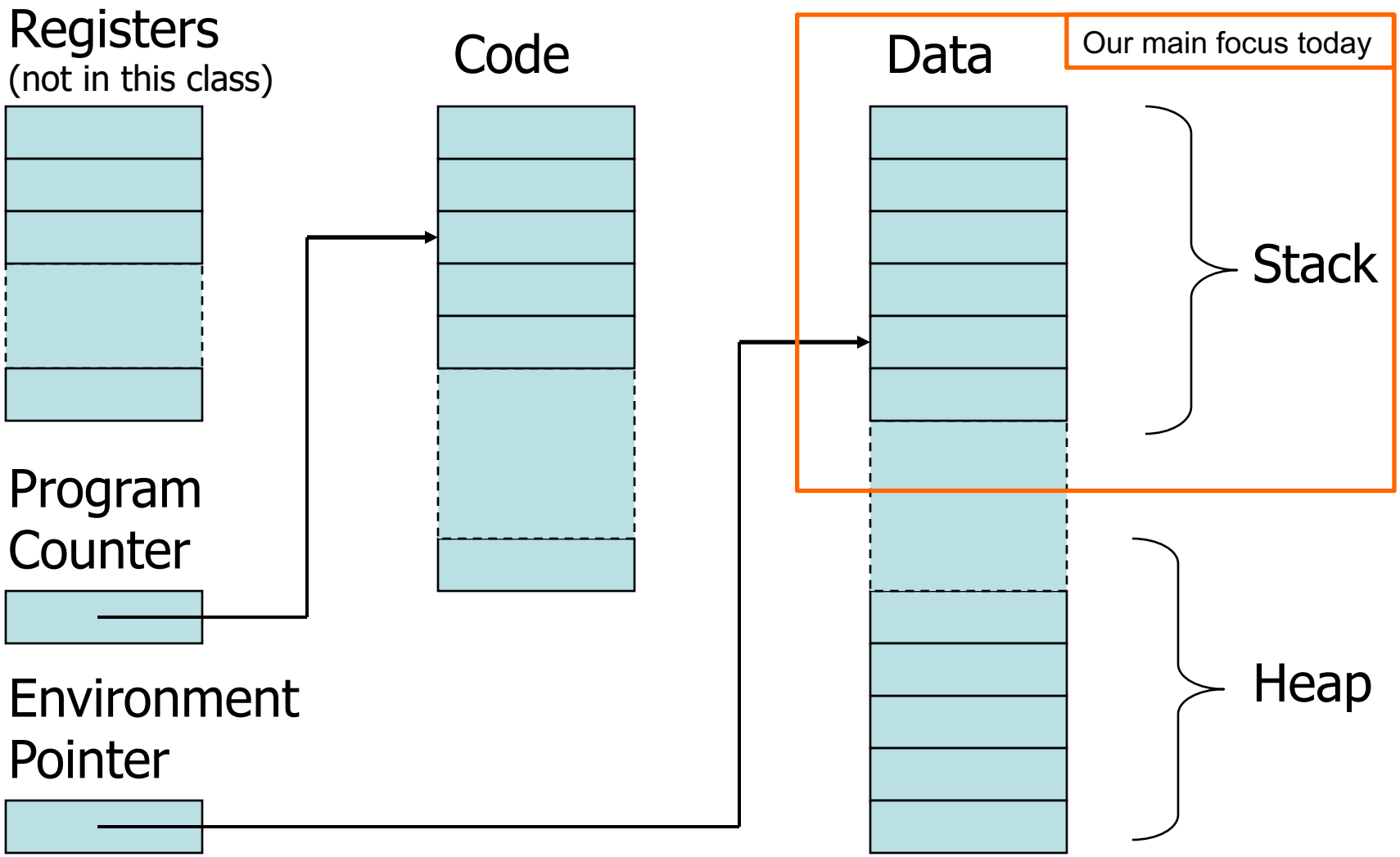

birger@ifi.uio.no


INF 3110 – 2018

# Outline: Runtime Organization I & II

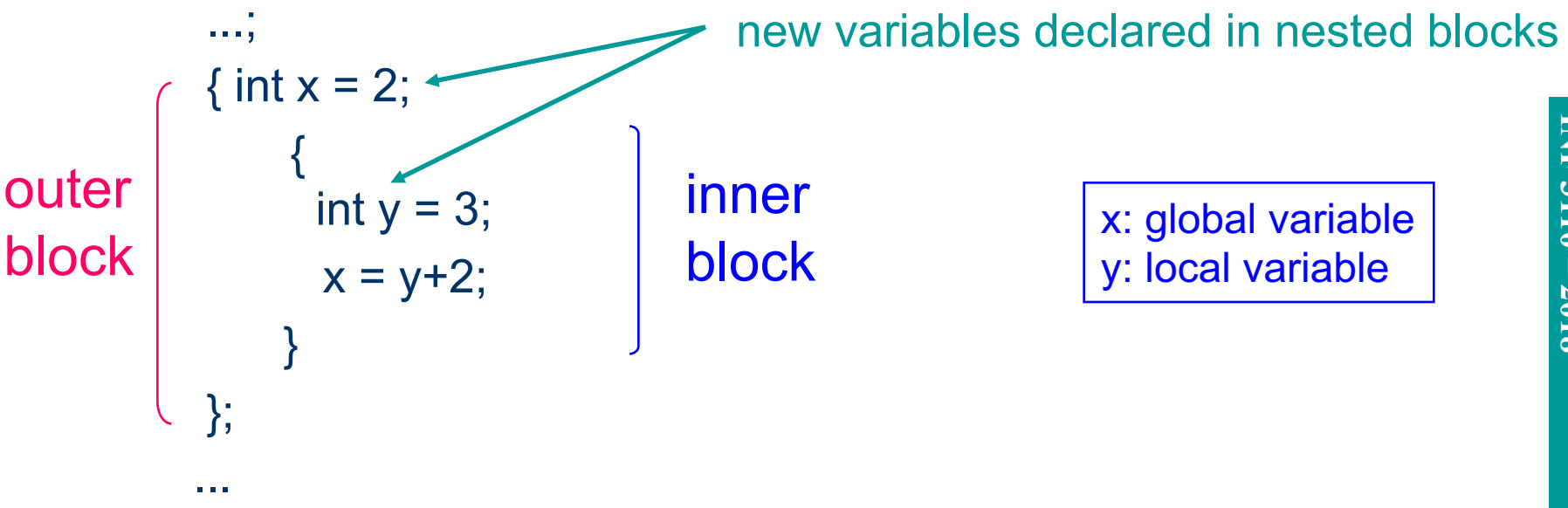
- Block-structured languages and stack organization
- In-line Blocks
  - activation records
  - storage for local and global variables
- First-order functions
- Parameter passing
- Higher-order functions (not today)

# Simplified Reference Model of a Machine - used to understand memory management



# Block-Structured Languages

- Blocks are *syntactical* structures
- Can be nested within each other




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

# Examples

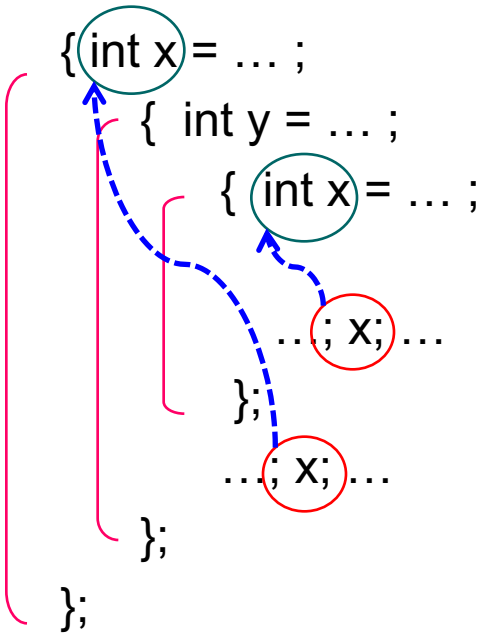
- Blocks in common languages
  - C/C++/Java/C# { … }
  - Algol/Simula/Basic begin … end
  - ML let … in … end
  - Python (whitespace!)

- Two forms of blocks to start with
  - In-line blocks
  - Blocks associated with functions or procedures
  - To come: blocks associated with classes:

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

# Some basic concepts

- Declaration
  - Specifies properties (name, type, kind) of an identifier
- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated




- 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 in a block

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

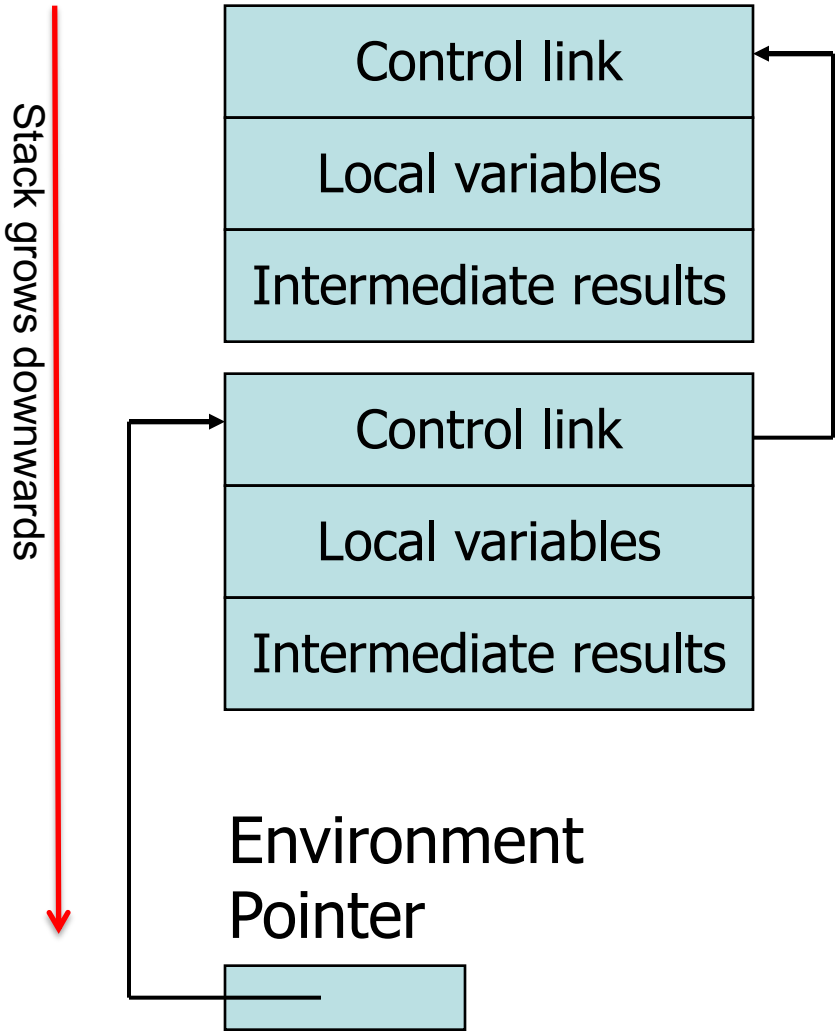
Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block

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

Do we *need* activation records for blocks?

Yes, for loops (and funcs etc)

# Activation record for in-line block

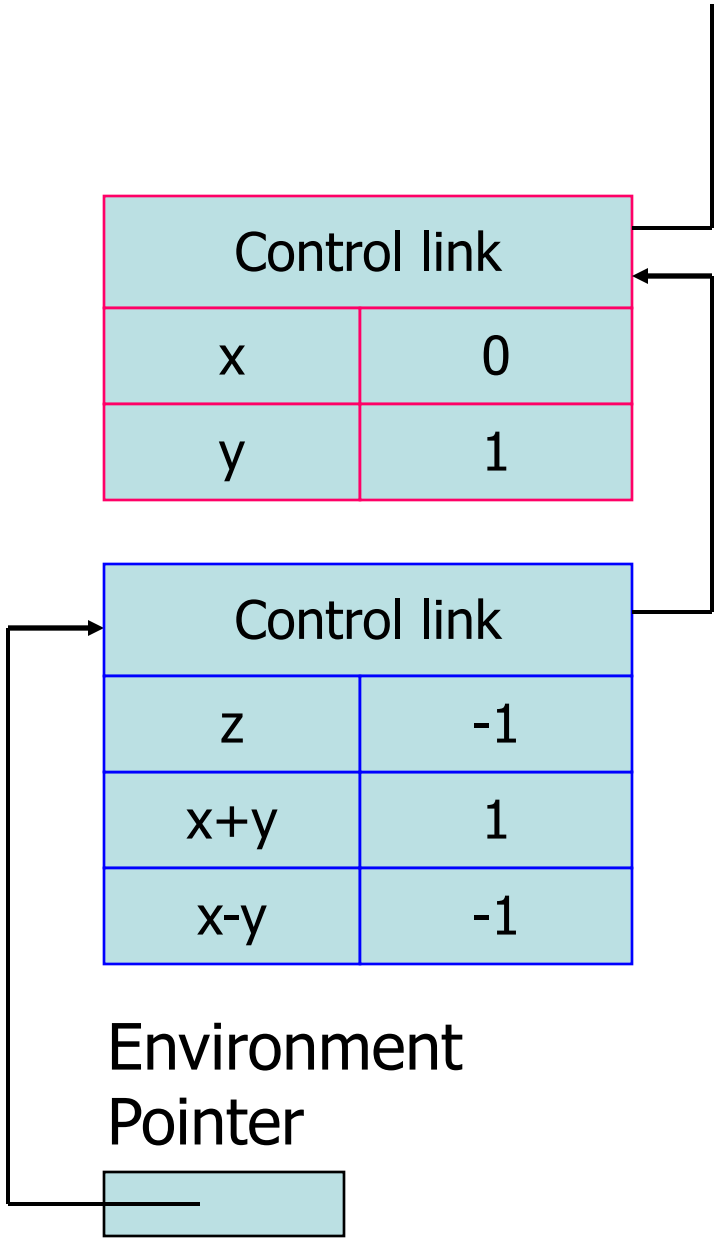



- Environment pointer
  - Pointer to current record on stack
- Control link (dynamic link)
  - Pointer to previous record on stack
  - Why is it called *dynamic*?
- Push record on stack
  - Set new control link (in new record) to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer
  - (No need to actively blank memory)

# Example

```
{ int x = 0;
    int y = x+1;
        {  int z = (x+y)*(x-y);
              // figure shows state here!
        };
};
```

- 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 – what is in the activation block?

- Specified initial value
- Default initial value

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

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

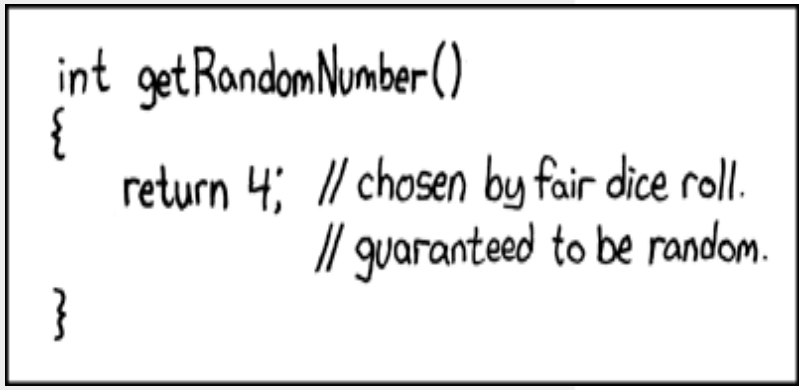
# Is uninitialized local variable the fastest random number generator?

262

48

I know the uninitialized local variable is undefined behaviour(*UB*), and also the value may have trap representations which may affect further operation, but sometimes I want to use the random number only for visual representation and will not further use them in other part of program, for example, set something with random color in a visual effect, for example:

```
void updateEffect(){
    for(int i=0;i<1000;i++){
        int r;
        int g;
        int b;
        star[i].setColor(r%255,g%255,b%255);
        bool isVisible;
        star[i].setVisible(isVisible);
    }
}
```




is it that faster than

```
void updateEffect(){
    for(int i=0;i<1000;i++){
        star[i].setColor(rand()%255,rand()%255,rand()%255);
        star[i].setVisible(rand()%2==0?true:false);
    }
}
```

and also faster than other random number generator? http://stackoverflow.com/questions/31739792/is-uninitialized-local-variable-the-fastest-random-number-generator

# Scope 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

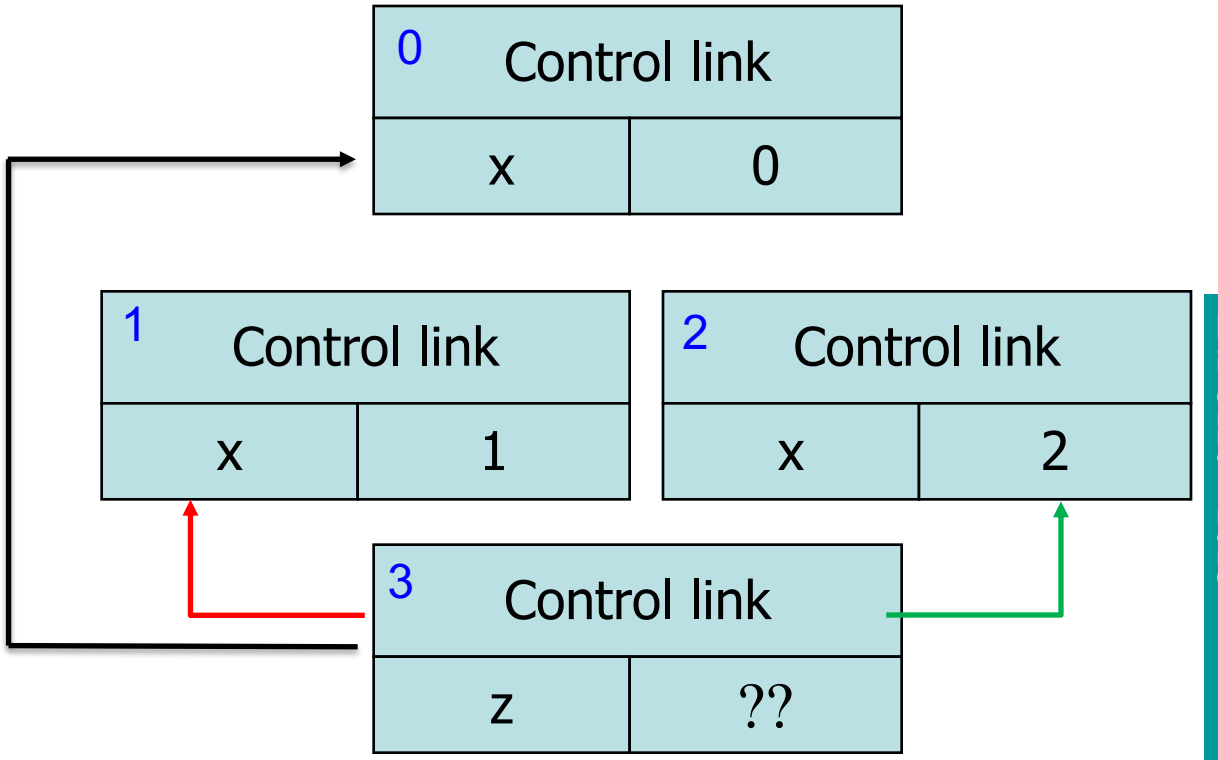
```
{ int x = 0;
  int y = x+1;
      {  int z = (x+y)*(x-y);
      };
};
```

- 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

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




If block 0 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

If block 1 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

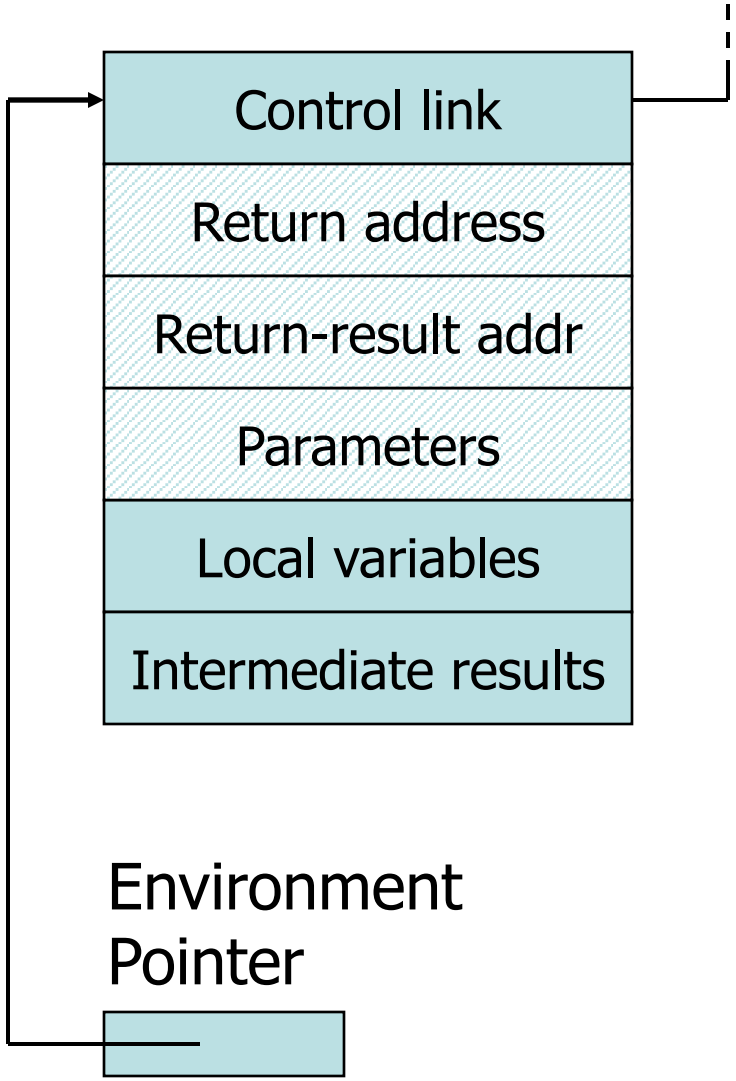
If block 2 executes block 3
Static scope: z = ?
Dynamic scope: z = ?

# Functions and procedures

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

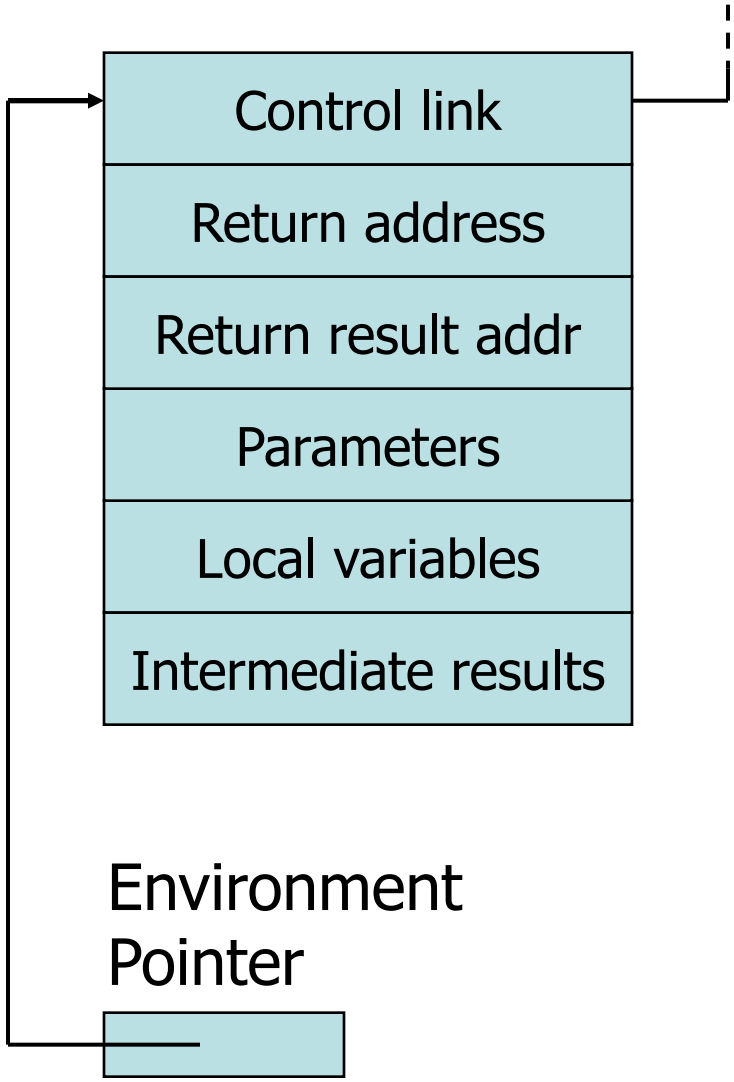
```
int fact(int n) { … }
int i, j;
...
i = 3;
j = fact(i);
print(j)
```

# Activation record for function call




- 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

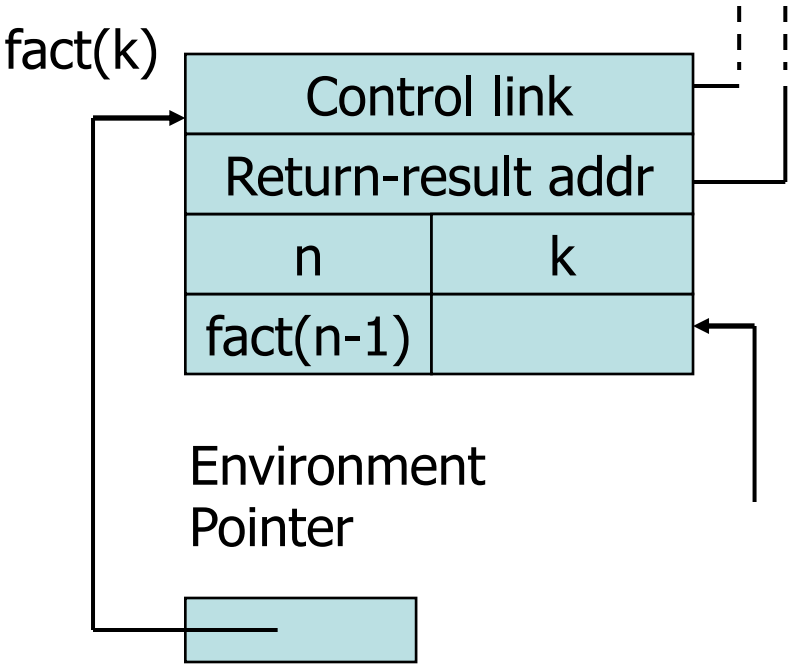



- Function

  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
  - location to contain value of fact(n-1)

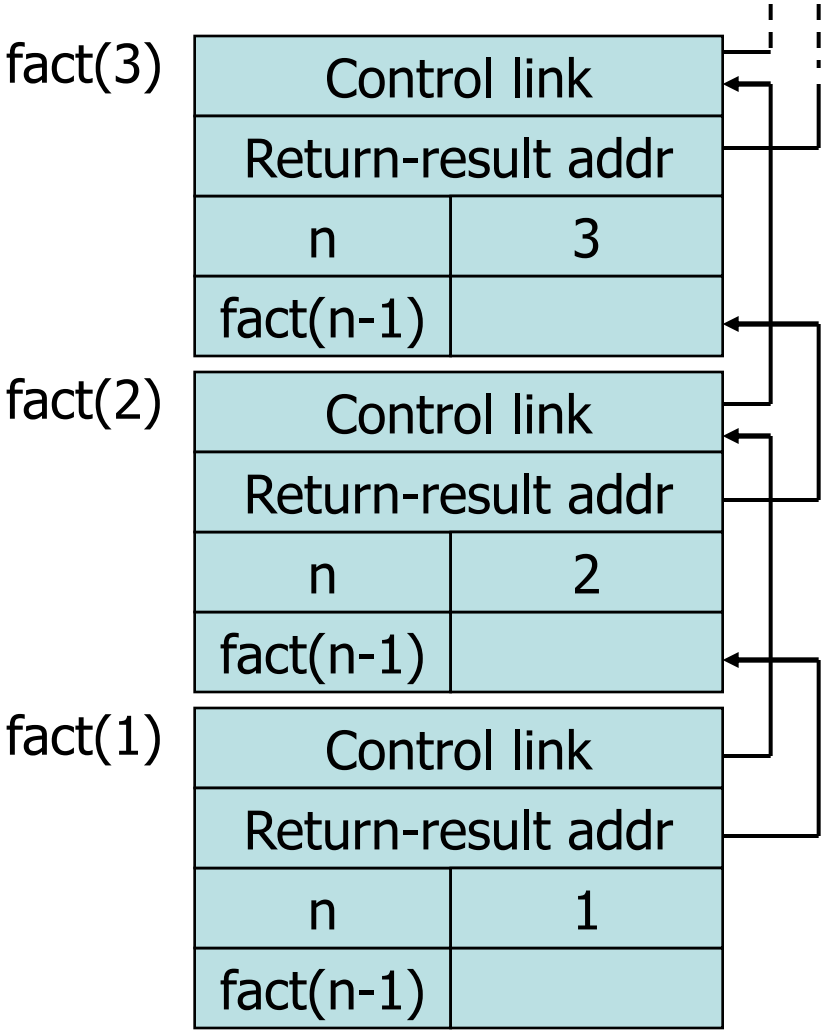
# Function call




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

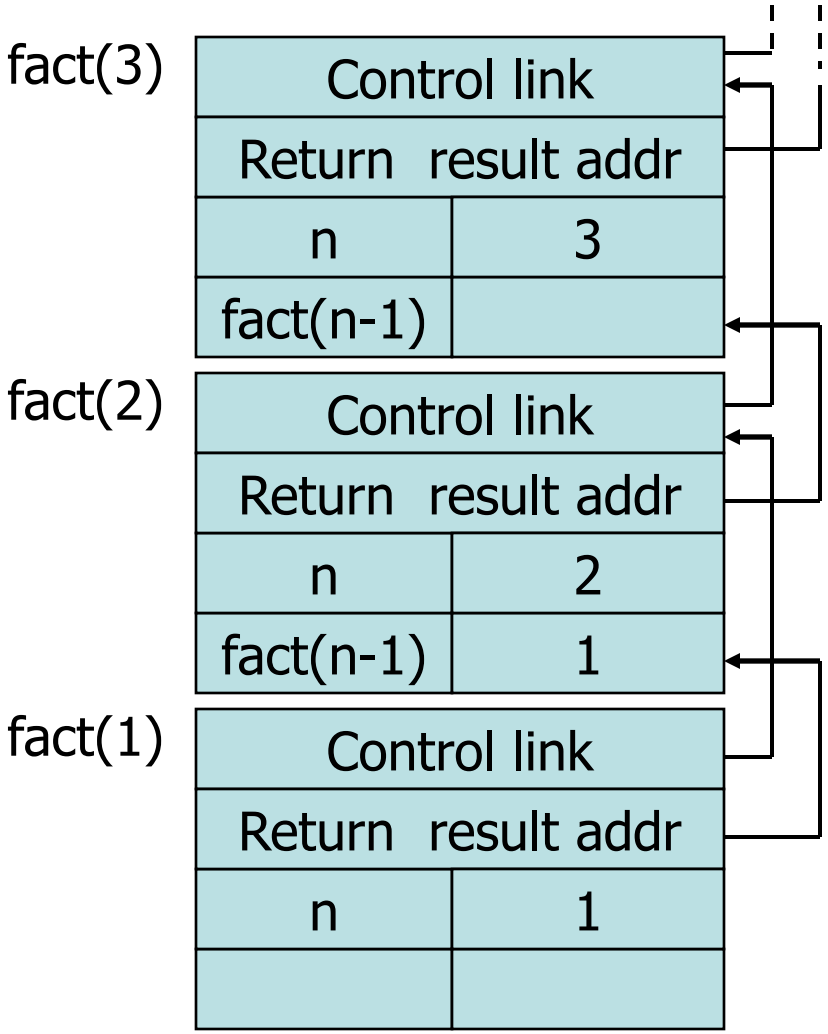
else n * fact(n-1)

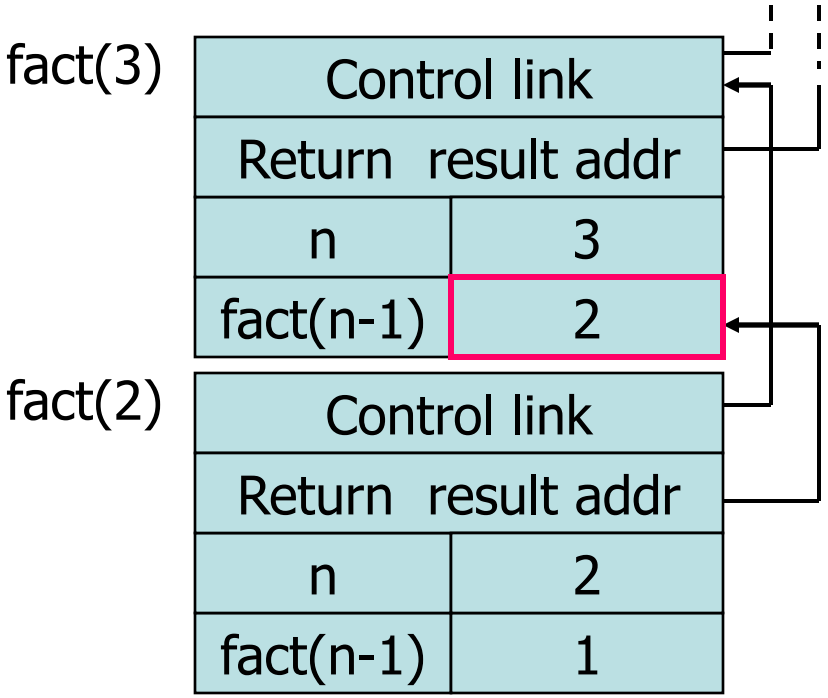
Return address omitted; would be ptr into code segment




Function return next slide →

# Function return – store result, pop record from the stack



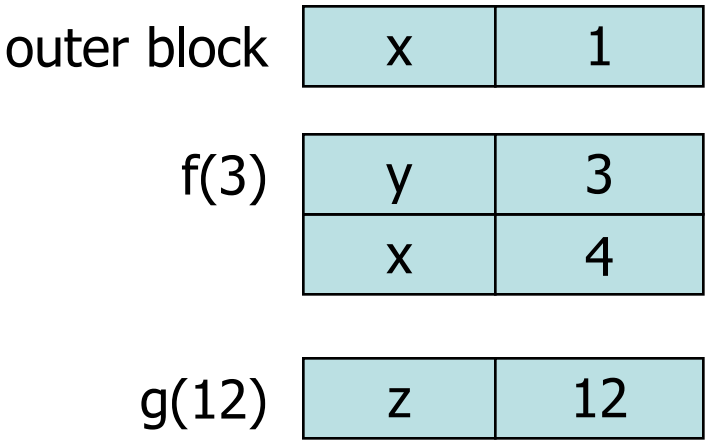




fact(n) = if n<= 1 then 1
else n * fact(n-1)

# Access to global variables

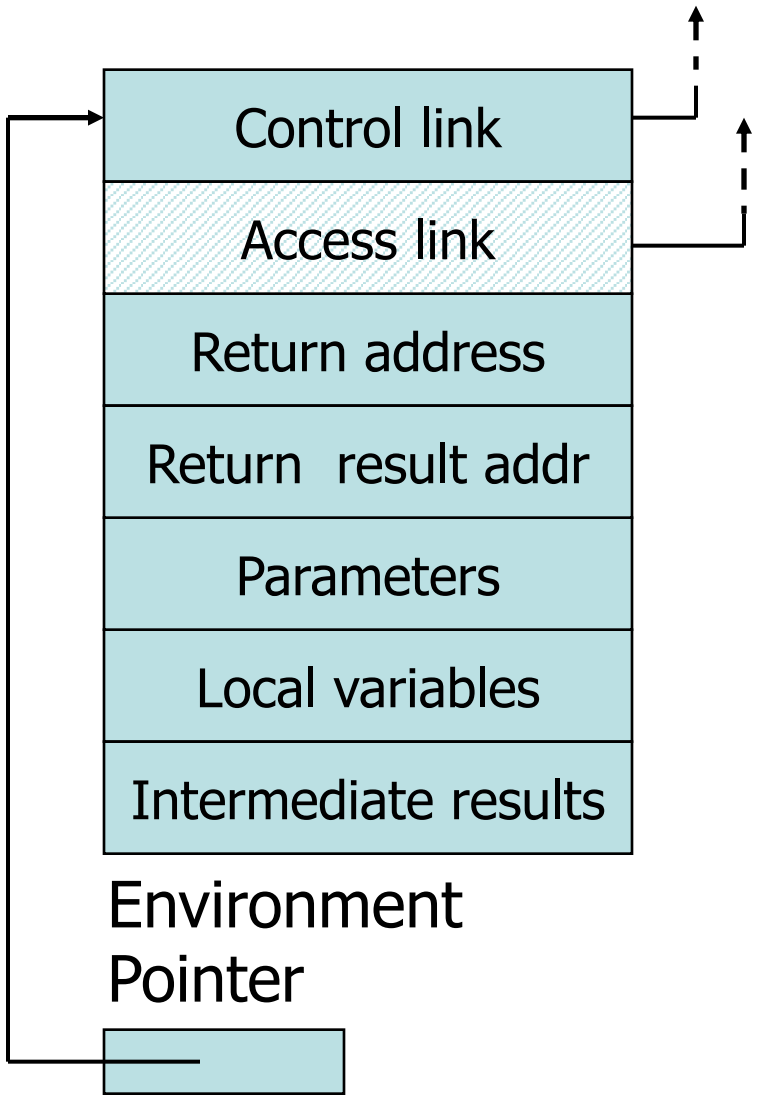
- Two possible scoping conventions
  - Static scope: refer to closest enclosing block (syntactically)
  - Dynamic scope: most recent activation record on stack
- Example

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




*Which x is used for expression x+z?*

# Activation record for static scope



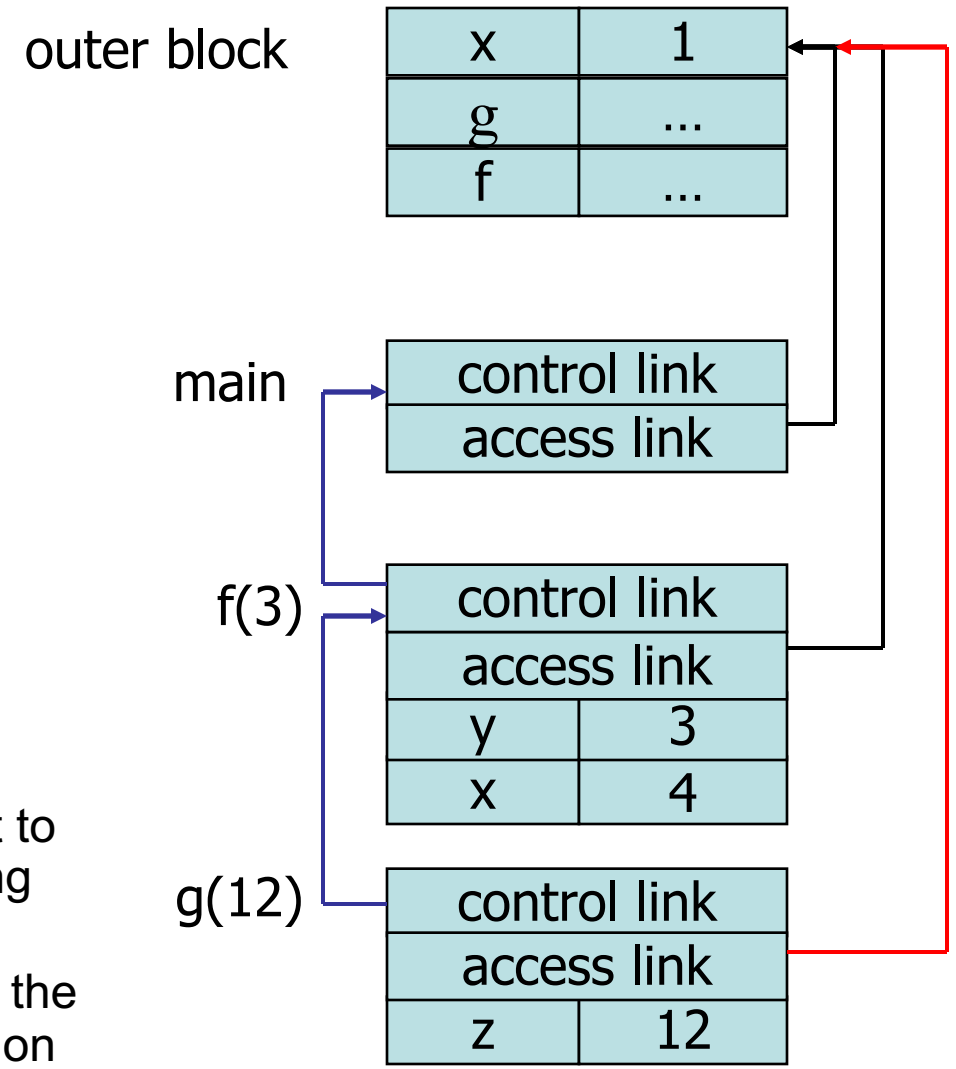

- Control link (dynamic link)
  - Link to activation record of previous (calling) block
- Access link (static link)
  - Link to activation record corresponding to the closest enclosing block in program text
  - Why is it called *static*?
- Difference
  - Control link depends on dynamic behavior of program
  - Access link depends on static form of program text

# Static scope with access links (C-like notation)

```
{
  int x = 1;

  int function g(z) { return x+z };

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

  main() {
    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 the block that contains function declaration

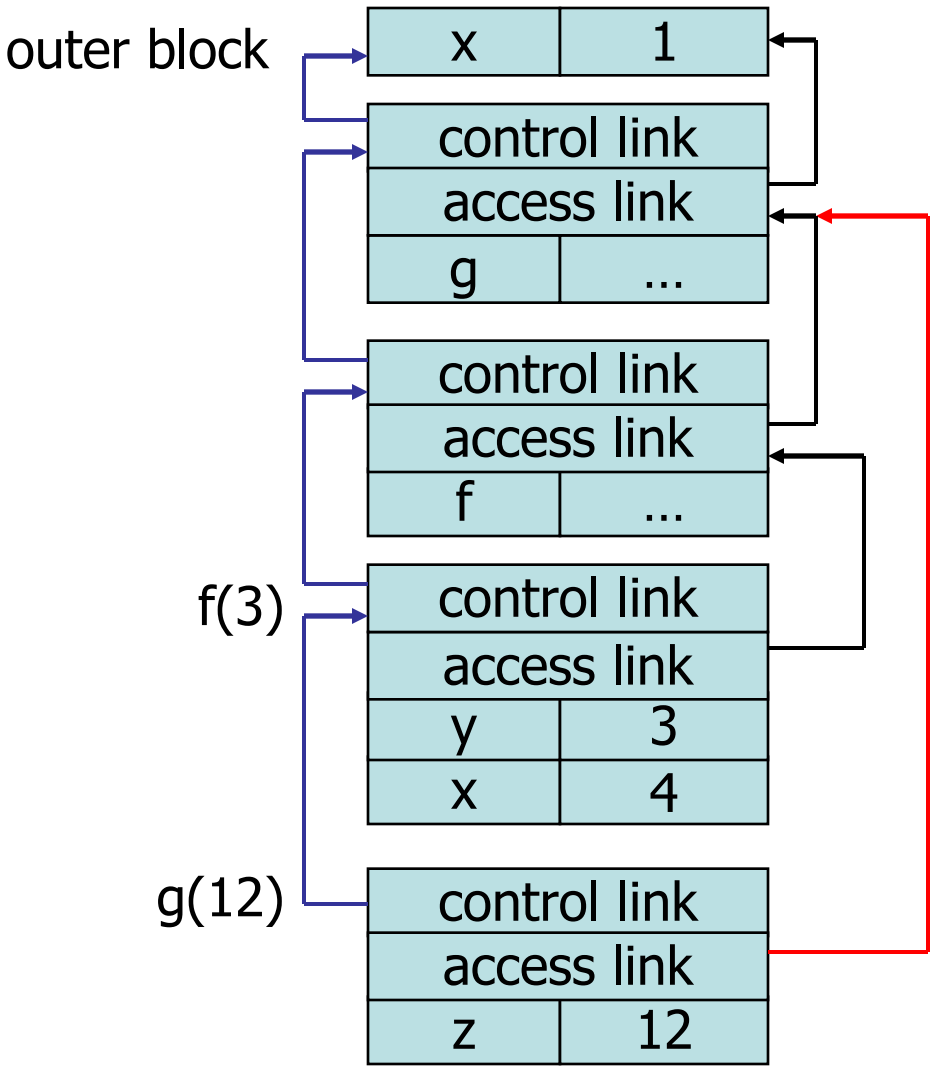


# Static scope with access links (ML-like notation)

```
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 the block that contains function declaration



# Issues for first-order functions

- Access to global variables ✓

- Parameter passing
  - pass-by-value
  - pass-by-reference
  - pass-by-name

```
int a = 5;

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

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

# Parameter passing

- Call-by-value (or "*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)
- Call-by-reference
  - Caller places **L-value** (address) of actual parameter in activation record
  - Function can assign to variable that is passed (must have an L-value!)
  - Aliasing
- Call-by-name
  - Actual parameter expression is passed as such and evaluated whenever the formal parameter is used in the function
  - (Not common in most languages)

# Different variants of copying the value

- 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);
}
```

# What is the parameter passing semantics of YOUR favorite language?

- Java?
- C#?
- Python?
- JavaScript?
- SML?

```
class Person {
   public String name = “”;
}


public static void main(String[] args) {
    Person p = new Person();
    p.name = “Wonderwoman!”;
    changeName(p);
    println(p.name);  // what happens here?
}

public static void changeName(Person p) {
    p.name = “Batman!”;
}
```

# What is the parameter passing semantics of YOUR favorite language?

- Java?
- C#?
- Python?
- JavaScript?
- SML?

```
class Person {
   public String name = “”;
}


public static void main(String[] args) {
    Person p = new Person();
    p.name = “Wonderwoman!”;
    changeName(p);
    println(p.name);  // ?
}

public static void changeName(Person p) {
    p = new Person();
    p.name = “Batman!”;
}
```

# What is the parameter passing semantics of YOUR favorite language?

- Java?
- C#?
- Python?
- JavaScript?
- SML?

```
class Person {
   public String name = “”;
}


public static void main(String[] args) {
    Person p = new Person();
    p.name = “Wonderwoman!”;
    changeName(p);
    println(p.name);  // ?
}

public static void changeName(ref Person p) {
    p.name = “Batman!”;
}
```

# What is the parameter passing semantics of YOUR favorite language?

- Java?
- C#?
- Python?
- JavaScript?
- SML?

```
class Person {
   public String name = “”;
}


public static void main(String[] args) {
    Person p = new Person();
    p.name = “Wonderwoman!”;
    changeName(p);
    println(p.name);  // ?
}

public static void changeName(ref Person p) {
    p = new Person();
    p.name = “Batman!”;
}
```

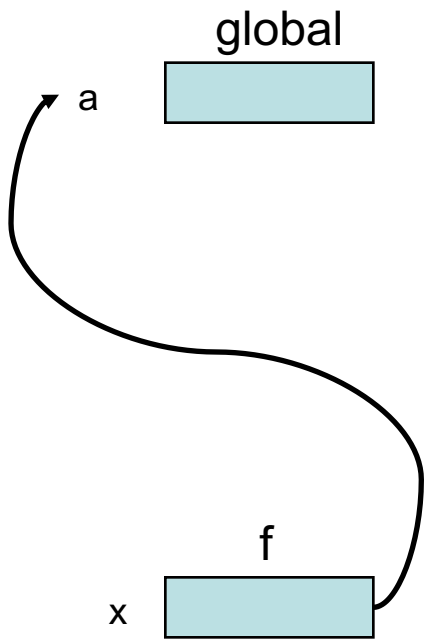
# What is the parameter passing semantics of YOUR favorite language?

- Java?
- C#?
- Python?
- JavaScript?
- SML?

```
class Person {
   public String name = “”;
}


public static void main(String[] args) {
    Person p = new Person();
    p.name = “Wonderwoman!”;
    changeName(p);
    println(p.name);  // ?
}

public static void changeName(ref Person p) {
    p = null;
}
```

# Call 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 f(ref real v1, ref real v2, ref real v3) {
  v1 = v1 - v3;
  v2 = v2 + v3;
};

real x, y, z;

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

f(x, y, z);    // ?
```

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

```
f(x, y, x)     // ?
```

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

Shouldn't the compiler warn me about this stuff???

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

# Going forward

- Higher-order functions
- Closures
- Classes and objects

Next time:

- Part II of this lecture