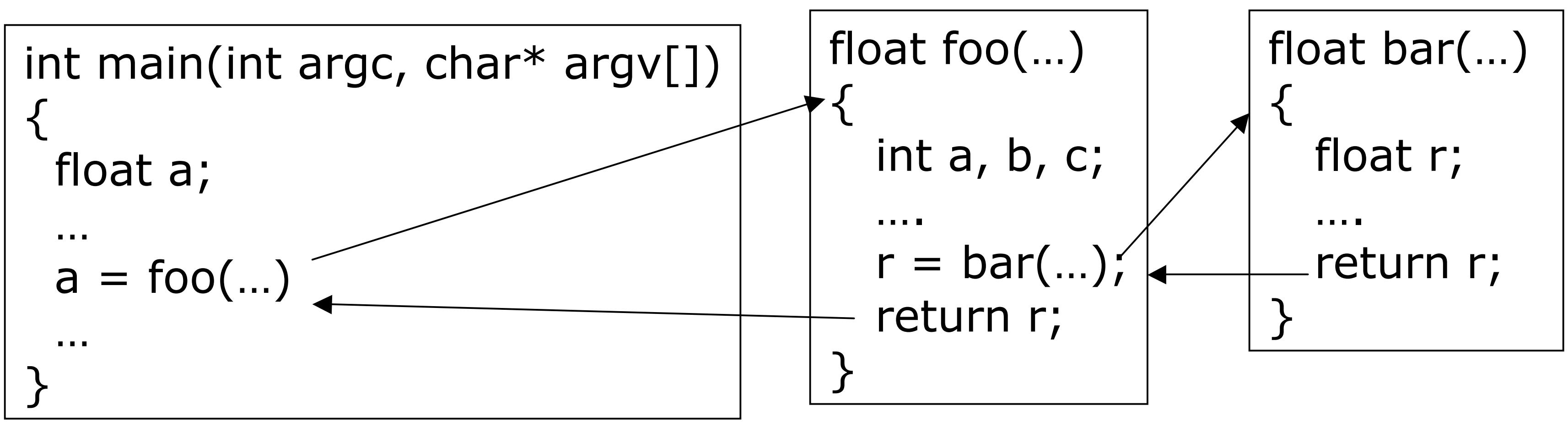
Procedure and Object-Oriented Abstraction

Scope and storage management

Procedure abstractions

- Procedures are fundamental programming abstractions
  - They are used to support dynamically nested blocks
    - Paired function call and return jumps
  - They have standalone semantics defined by an abstraction interface
    - input parameters, return values, global side effects
- Procedures are units of separate compilation
  - They represent parameterized blocks of computation



Scoping rules

- Global and local variables

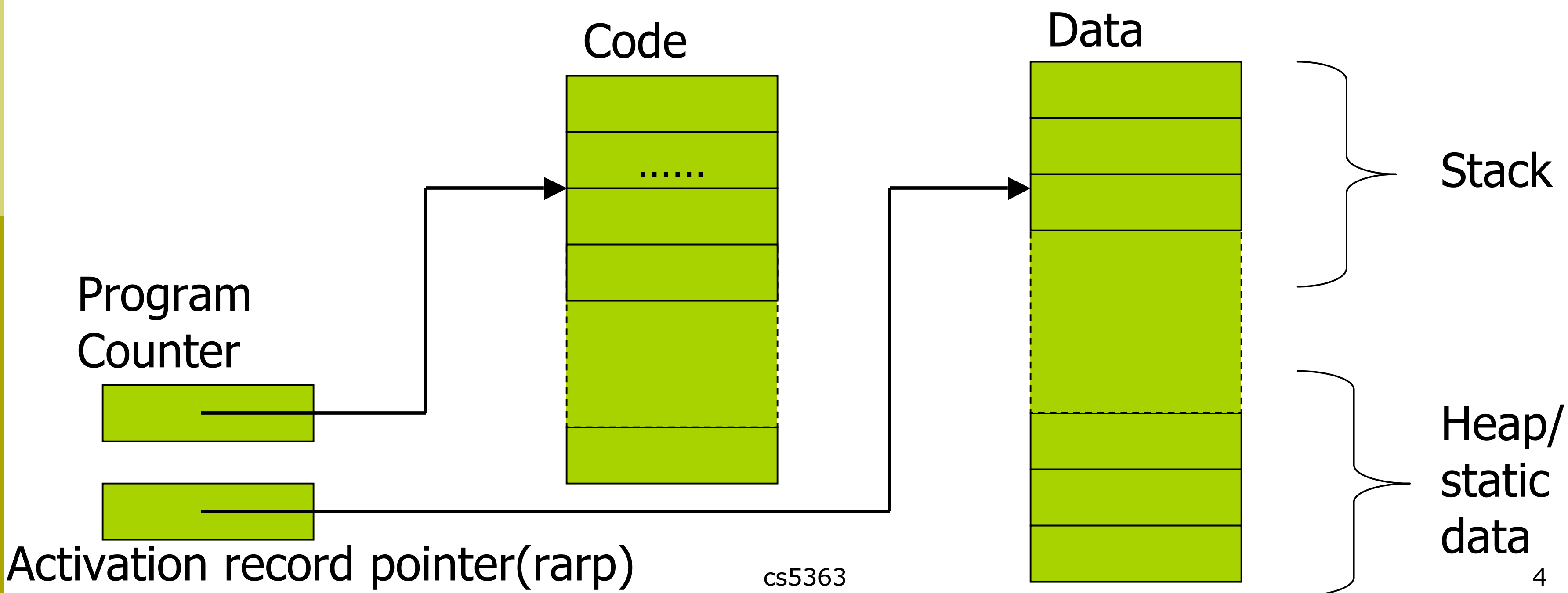
outer block	x	0
h(3)	z	3
	x	1
g(12)	z	3

```
program main(input,output);
  var x : integer;
  function g(z: integer) :integer;
    begin g := x+z   end;
  function h(z: integer) :integer;
    var x : integer;
    begin x := 1; h:=g(z) end;
 begin x := 0; print(h(3)) end
```

- Static scoping
  - Find global variables in enclosing blocks in program text
- Dynamic scoping
  - Find global variables in the most recently evaluated blocks
  - Easier to implement in interpreted languages
- What is the scoping rule for C/C++, Java?

Simplified memory model

- Runtime stack: activation records of blocks/functions
  - Block entry: add new data to stack
  - Block exit: remove outdated data
- Heap: data of varying lifetime
  - Variables that last throughout the program
  - Address may be contained by variables on the runtime stack



Managing Data Storage

- Local variables --- activation records on stack
  - Declared inside a block (e.g. function body)
    - Enter block: allocate space
    - Exit block: de-allocate space
  - Local variables in an enclosing block
    - Already allocated before entering current Block
    - Remain allocated after exiting current block
  - Function parameters and return value
    - Allocated and initialized before entering function body
    - Formal parameters dallocated after exiting function body
- Global/static variables --- static data areas
  - Allocated when program is loaded to memory
  - Storage remain until program exits
- Dynamically allocated variables --- heap
  - Storage dynamically allocated at runtime (e.g., malloc in C)
  - Storage remain until explicitly de-allocated or garbage collected

Activation Record

- Allocate storage for each block dynamically
  - Allocate an activation record before evaluating each block
    - Storage for each local variable determined as compile time
    - Values of local variables evaluated at runtime
  - Delete the activation record after block exits

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

Allocate AR with space for x, y

Set values of x, y

    Allocate AR for inner block

    Set value of z

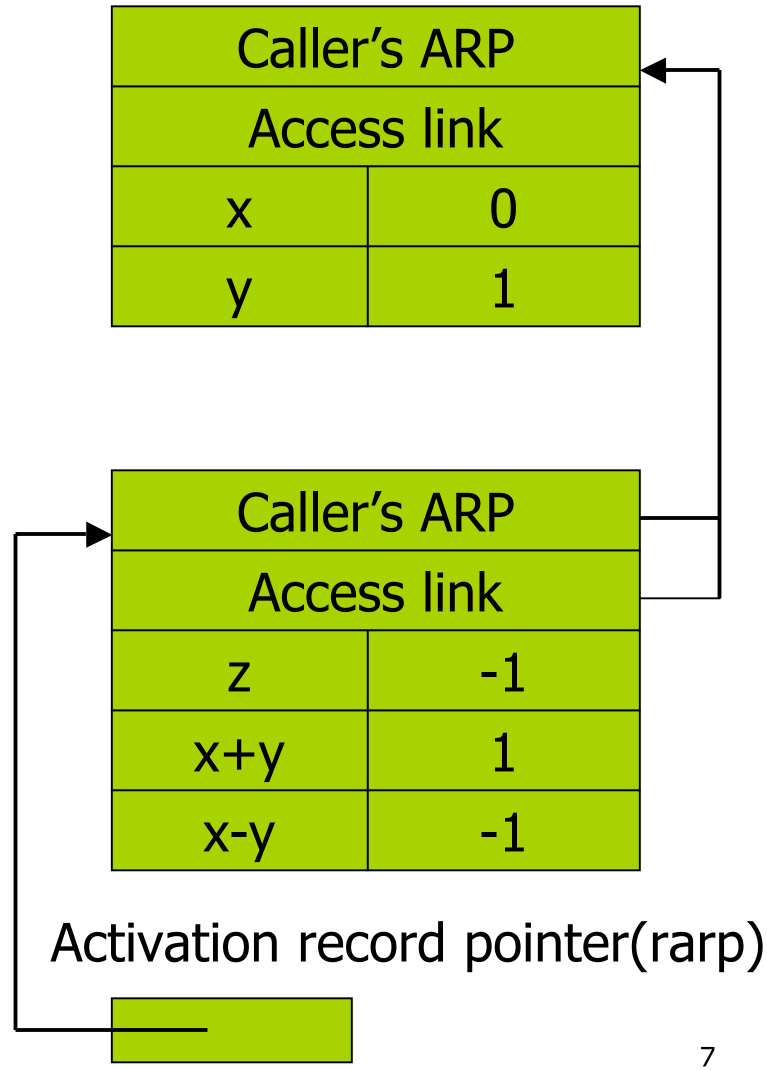
    Delete AR for inner block

Delete AR for outer block

May need space for intermediate results such as (x+y), (x-y)

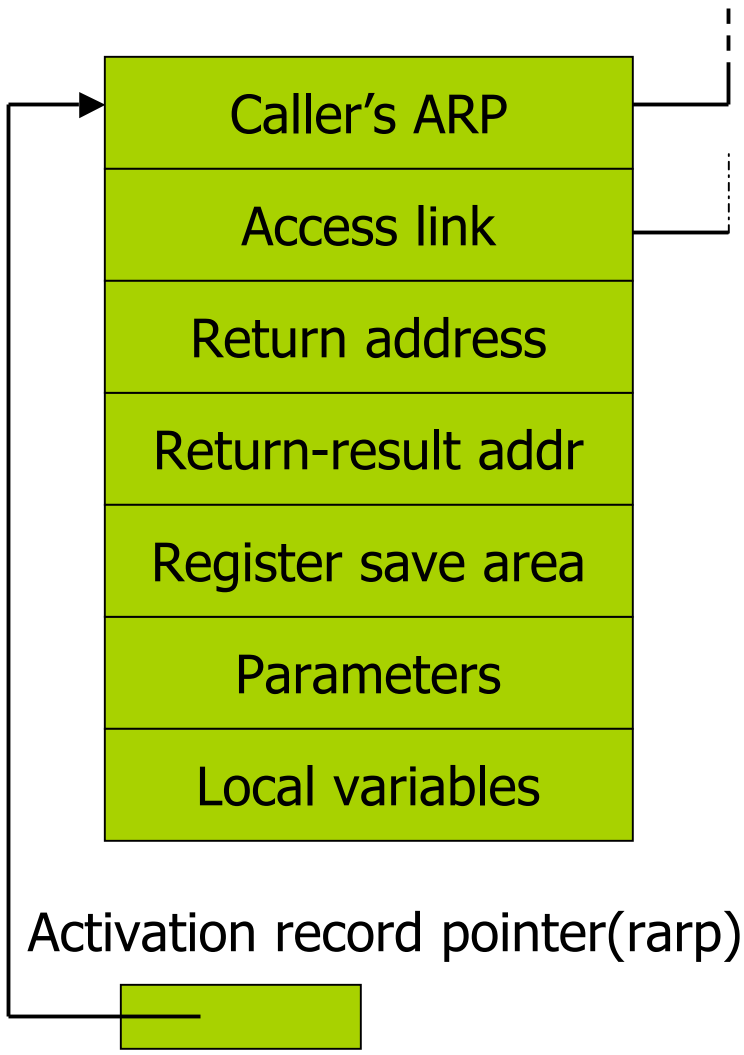
Activation Records For Inline Blocks

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



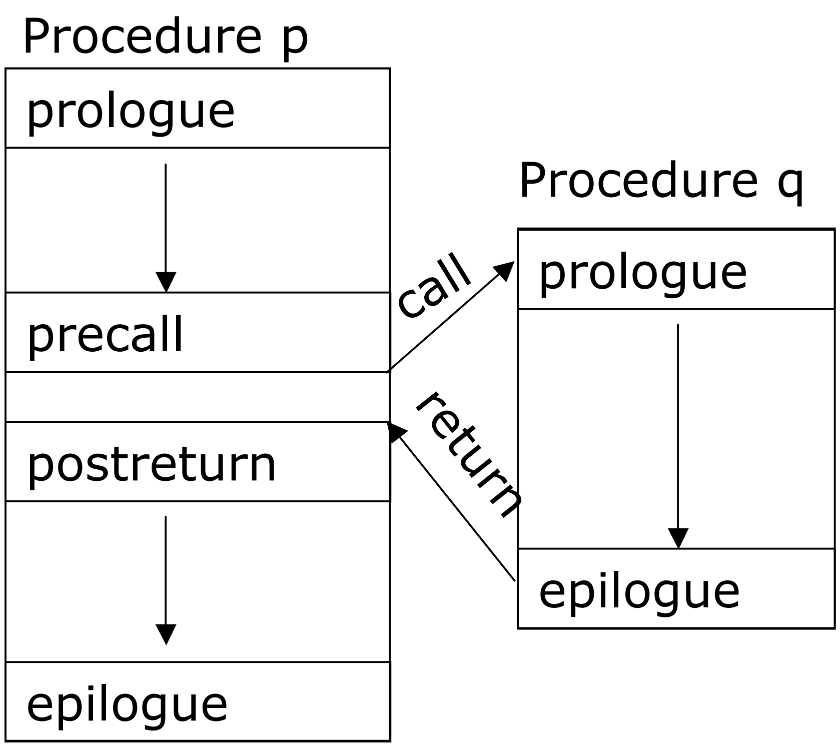
- Push activation record on stack
  - Set caller ARP to rarp
  - Set rarp to new AR
- Pop activation record off stack
  - Reset rarp to caller's ARP
- When making function calls
  - Caller must also set return address, return value addr, saved registers, and parameters

Activation Records For Procedures



- Access link
  - Pointer to activation record of the enclosing block
- Return address
  - Pointer to the instruction immediately following function call
- Return-result address
  - Address of the storage to put the result to be returned
- Register save area
  - Save register values before function call
  - Restore register values before return
- Parameters
  - Storage for function parameters
  - Values initialized by caller

Linkage Convention: Implementing Function Calls



Linkage convention:
programs in different files must follow a single contract of function call implementation

- Precall
  - Push callee's AR (increment rarp)
  - Set caller's ARP
  - Set return address
  - Set return result addr
  - Save live register values
  - initialize formal parameters
- Postreturn
  - Restore live register values
  - Pop callee's AR(decrement rarp)
- Prologue
  - Initialize local variables
  - Load local environment (access link)
- Epilogue
  - Deallocate local variables
  - Goto return address

Parameter Passing

- Formal and actual parameters
  - Parameter declarations and initializations
- Pass-by-value
  - Formal parameters contain values of actual parameters
  - Callee cannot change values of actual parameters
- Pass-by-reference
  - Formal parameters contain locations of actual parameters
  - Callee can change values of actual parameters
  - Formal parameters in activation record may be aliased
    - Aliasing: two names refer to same location
- What about pass-by-pointer (in C)?

Formal parameter

```
int f (int x)
   {
    x := x+1; return x;
   };
main() {
  int y = 0;
  print f(y)+y;
}
```

Actual parameter

Example: What is the final result?

pseudo-code

```
int f (int x)
    {
     x := x+1; return x;
    };
main() {
  int y = 0;
  print f(y)+y;
}
```

pass-by-ref =>2

pass-by-value =>1

- Draw the activation records for the evaluation
- What parameter passing is supported by the languages you know?

Exercise: Managing Function Calls

```
program main(input,output)
  var x : integar;
  function f(y : integer)
    begin
      f = (x + y) - 2
    end
  function g(function h(b:integer):integer)
    begin
      var x : integer;
      x := 7;
      g = h(x);
    end
 begin
   x := 5;
   g(f);
 end
```

Accessing Variables In Memory

- Each memory store has an address
  - Base address: the starting address of a data area
    - Local variables of current block
      activation record pointer (rarp)
  - Offset: the number of bytes after the base address
    - Local variables of current block
      predetermined at compile time
- Address of variable
  - base address + offset

Accessing local variable a:

```
LoadAI rarp, @a => r1
```

```
loadI @a => r1
loadA0  rarp, r1 => r2
```

```
loadI @a => r1
Add rarp, r1 => r2
load r2 => r3
```

Accessing Global/Static Variables

- Allocated separately in static data area
  - Base address unknown until program is loaded into memory
    - Use symbolic labels to substitute at compile time
    - Symbolic labels replaced with runtime value by assembler and loader
  - Offset calculated at compile time
    - Individual variables: offset=0
    - Group of data
      - layout pre-determined

Accessing global variable fee:

```
LoadI &fee => r1
Load r1 => r2
```

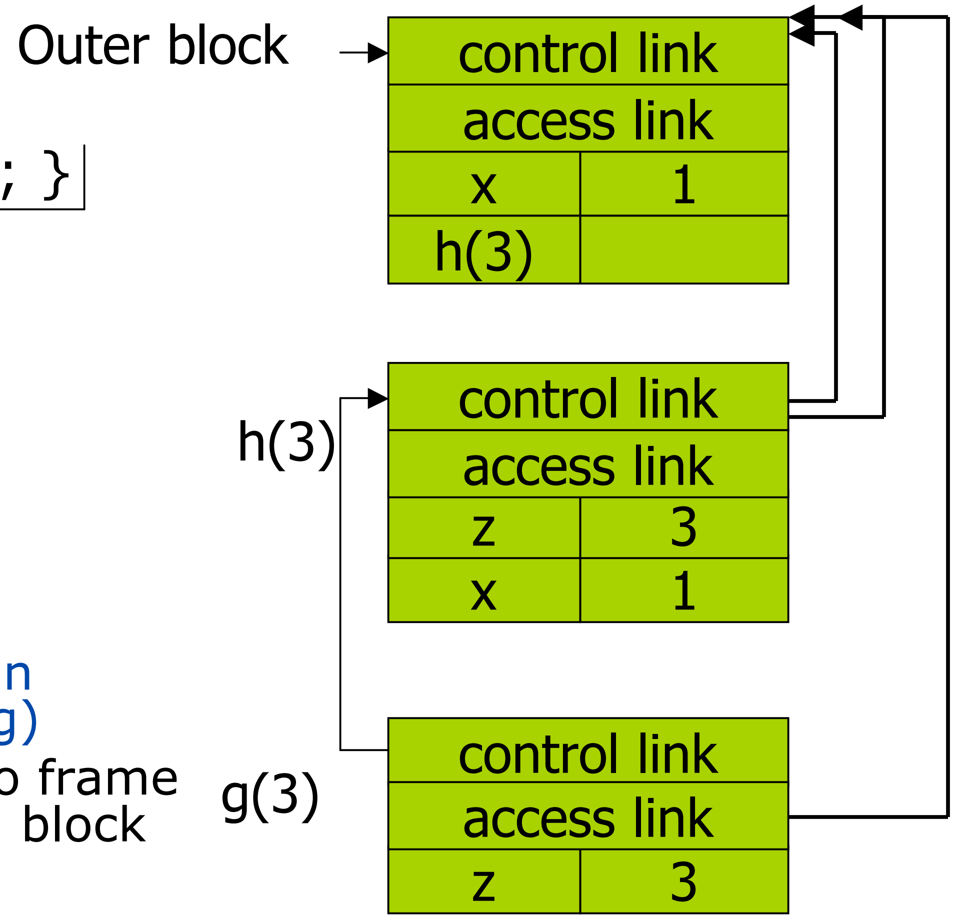
Accessing foo.a:

```
LoadI &foo => r1
LoadAI r1, @foo_a => r3
```

```
LoadI &foo => r1
Add r1, @foo_a => r2
Load r2 => r3
```

Variables of Enclosing Blocks

```
int x=1;
int g(int z) { return x+z; }
int h(int z) {
            int x = 1;
            return g(z); }
print h(3);
```



Use access link to find AR of an enclosing block (static scoping)

- Access link is always set to frame of closest enclosing lexical block

Coordinates of Variables

- Accessing local variables
  - Offset calculated at compile time
  - Need to find the base address
    - The AR that contains the variable
- Lexical level of a block
  - Number of enclosing scopes
    - g: 1; h: 1; outer-block: 0
- For each variable x
  - Coordinate of x is <n, o>, where
    - n: lexical level of block that defines x
    - O: offset of x in it's AR
  - If a block at lexical level m references x
    - Follow access link m-n times to find the base address for x

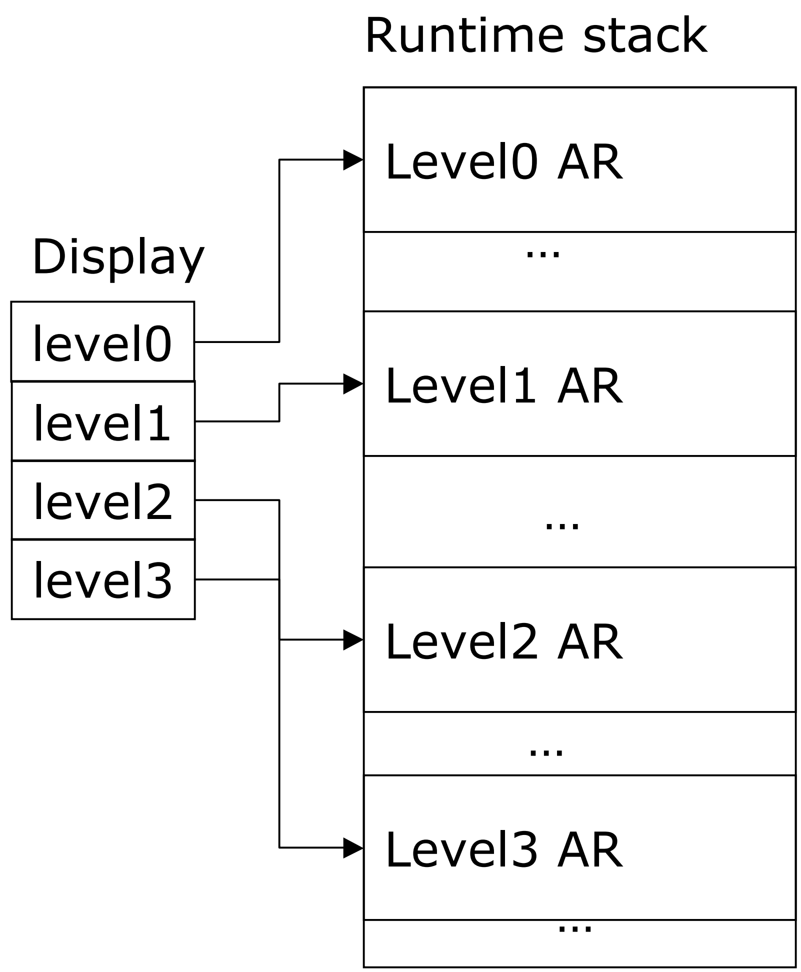
```
int x=1;
int g(int z) { return x+z; }
int h(int z) {
        int x = 1;
        return g(z); }
print h(3);
```

Coordinate for x: <0,8>
Lexical level of g: 1
Load instructions:

```
loadAI rarp, 4 => r1
loadAI r1, 8  => r2
```

Global Display

- Allocate a global array (global display)
  - hold the address of most recent ARs at each lexical level
  - When pushing a new AR, save the previous AR at the same lexical level, modify global display
  - When popping an AR, restore the global display with saved AR at the current lexical level
- To access variable <n,o>
  - use the ARP at element n of the global display



Global Display vs. Access Links

- Maintenance
  - Constant cost for global display
    - When entering every block at lexical level n, save the level-n ARP from global display, replace it with new ARP
    - When exiting the block, restore the old level-n ARP into display
  - Varying cost for access links
    - If a level-m block invokes a level-n block

      m==n–1 ➔ callee's access link points to caller's AR

      m==n ➔callee's access link = caller's access link

      m > n ➔callee's access link = caller's level (n-1) access link
- Referencing variables in enclosing scope
  - Constant cost through global display
  - Varying cost through access links
- The tradeoff depends on the ratio of non-local references
- If ARs can outlive their invocation, access link must be used
  - The chosen approach by functional programming languages

Managing memory

- Registers
  - Data need to be loaded to registers before being operated on
  - If a variable can be kept in register throughout its lifetime, it does not need a storage
  - Register-to-register model
    - Try to keep as many variables in registers as possible
    - Allocate memory storage later if not enough register
- Alignment and padding
  - Target machines may restrain where data can be stored
    - Needs to be at 32/64 bit boundaries, etc.
- Cache and variable layout
  - Data in memory can be loaded into cache and reused
- Managing the heap: dynamically allocate/free storage

Object-Orientation

- Abstraction: information hiding
  - Separate interface and implementation details
  - Function and data abstractions
- Object-oriented programming
  - Organize concepts into objects and classes
  - Build extensible systems
- Language concepts
  - Encapsulation (access control): members can be private
    - only a few functions can access private data
  - Dynamic lookup definitions of functions (function pointers)
    - Object behavior can change dynamically
  - Subtyping polymorphism (relations between types)
    - Operations can be applied to multiple types of values
  - Inheritance (reuse of implementation)
    - Subclasses can modify and inherit behavior of base classes

Static vs. dynamic lookup

- What about operator overloading (ad hoc polymorphism)?

```
int add(int x, int y)  { return x + y; }
float add(float x, float y) { return x + y; }
```

  - Static lookup: overloading is resolved at compile time
  - Examples: C++ non-virtual functions, Java static functions
- Dynamic lookup: resolved at run time
  - C++ virtual functions, Java non-static functions
  - Difference: flexibility vs. efficiency

```
class vehicle {
   protected: double speed, fuel;
   public: virtual void run() = 0;
};
class car : public vehicle {
   public: virtual void run() { if (fuel > 0) fuel = fuel – 1;}
};
vehicle* a = new car; a->run();
```

Static Binding of Methods

- C++ class: non-virtual member functions
  - Essentially global functions with an extra object pointer parameter

```
class vehicle {
  protected: double speed, fuel;
  public: vehicle() : speed(0),fuel(0) {}
          void start(double x) {speed = x;}
};
vehicle* a = new vehicle; a->start(5);
```

- Java/C++: Static Methods/Variables
  - Essentially global functions/variables in a name space
  - A single instance of member for all class objects

```
class vehicle {
  static protected double speed, fuel;
  public static void start(double x) {speed = x;}
};
Vehicle::start(3.0);
```

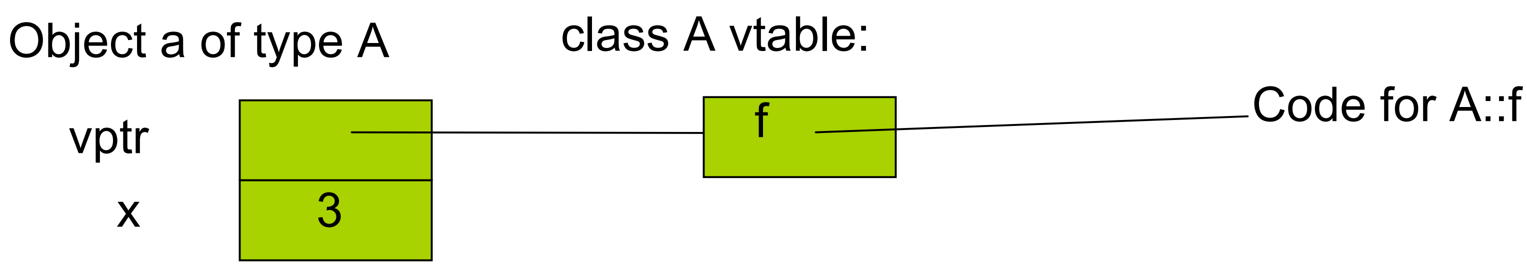
Implementing Dynamic Objects

- An object consists of
  - Hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - Public operations
    - methods or member functions
    - can also have public variables in some languages
  - Dynamic binding
    - Put all the name-value bindings into a table
    - Table can be changed, just like the activation record of a function
  - Example: the vehecle/car objects
- Object-oriented program:
  - Send messages to objects

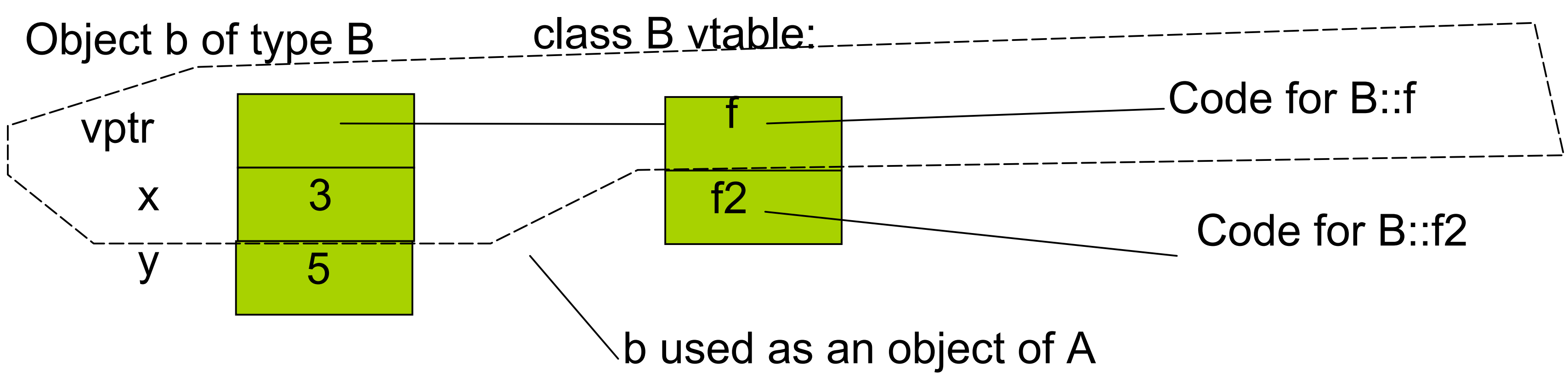
hidden data	
msg1	method1
. . .	. . .
msgn	methodn

C++: Object Layout and Single Inheritance

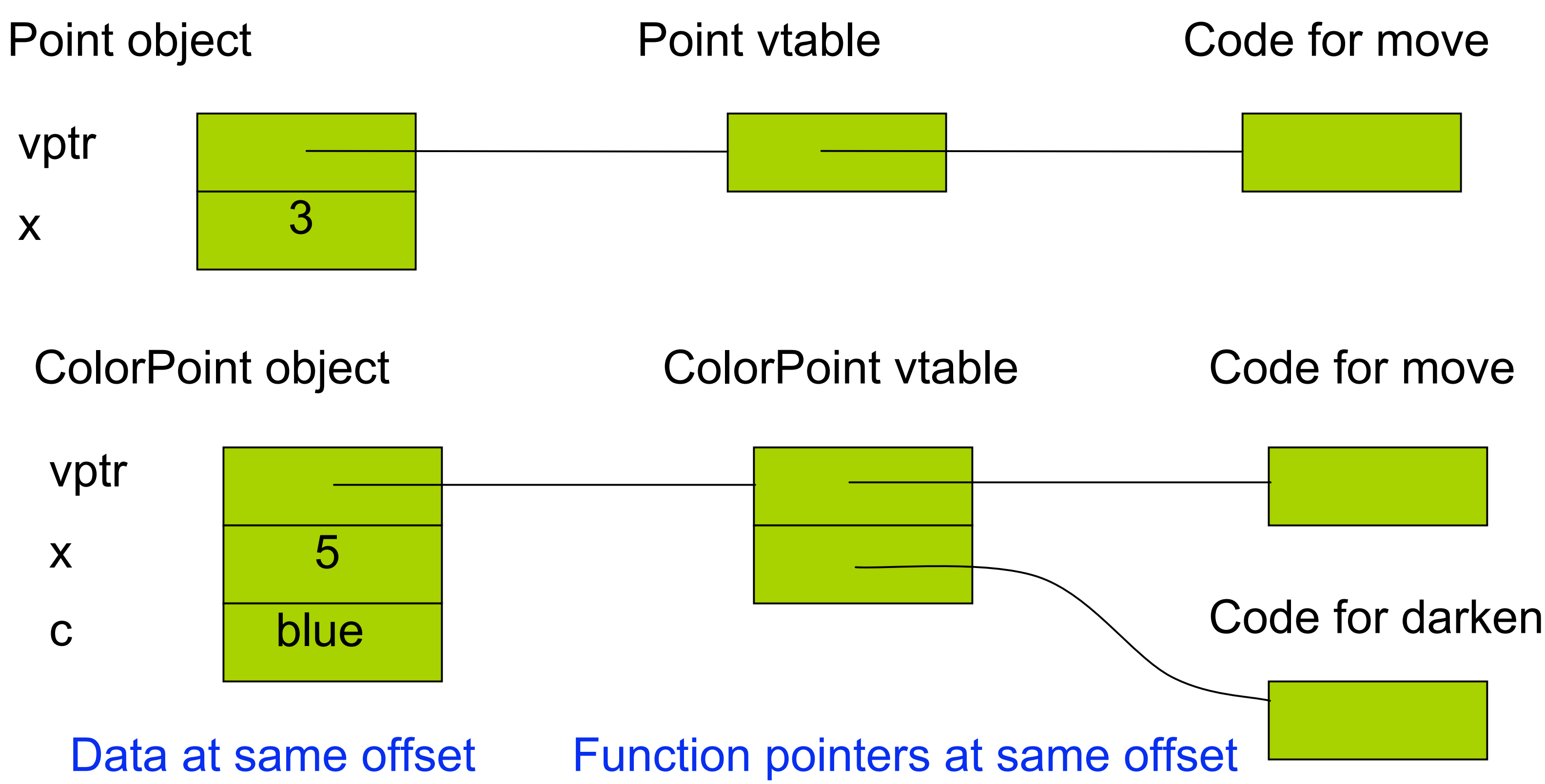
class A { int x;   public: virtual int f() { return x;}   };



class B : public A { int y;   public: virtual int f() { return y; }
virtual void f2() { … }   };



Looking up methods



Point p = new Pt(3);
p->move(2);   // (*(p->vptr[0]))(p,2)

C++ method lookup

- C++ compiler knows all the base classes
  - Offset of data and function pointer are same in subclass and base class
  - Offset of data and function pointer known at compile time
  - Code p->move(x) compiles to equivalent of (*(p->vptr[move_offset]))(p,x)

Exercise: OO Memory Layout

- Draw the memory layout for the following C++ code immediately before the main function returns.

```
class A { int x; public: virtual void f(); };
class B: public A { int y; public: virtual void f(); };
class C: public B { int z; public: virtual void g();  };
int main() { C *pc = new C; B *pb = pc; A *pa = pc; }
```