40-414 Compiler Design

Run-time Environments

Lecture 10

Outline

- Management of run-time resources

- Correspondence between
  - static (compile-time) and
  - dynamic (run-time) structures

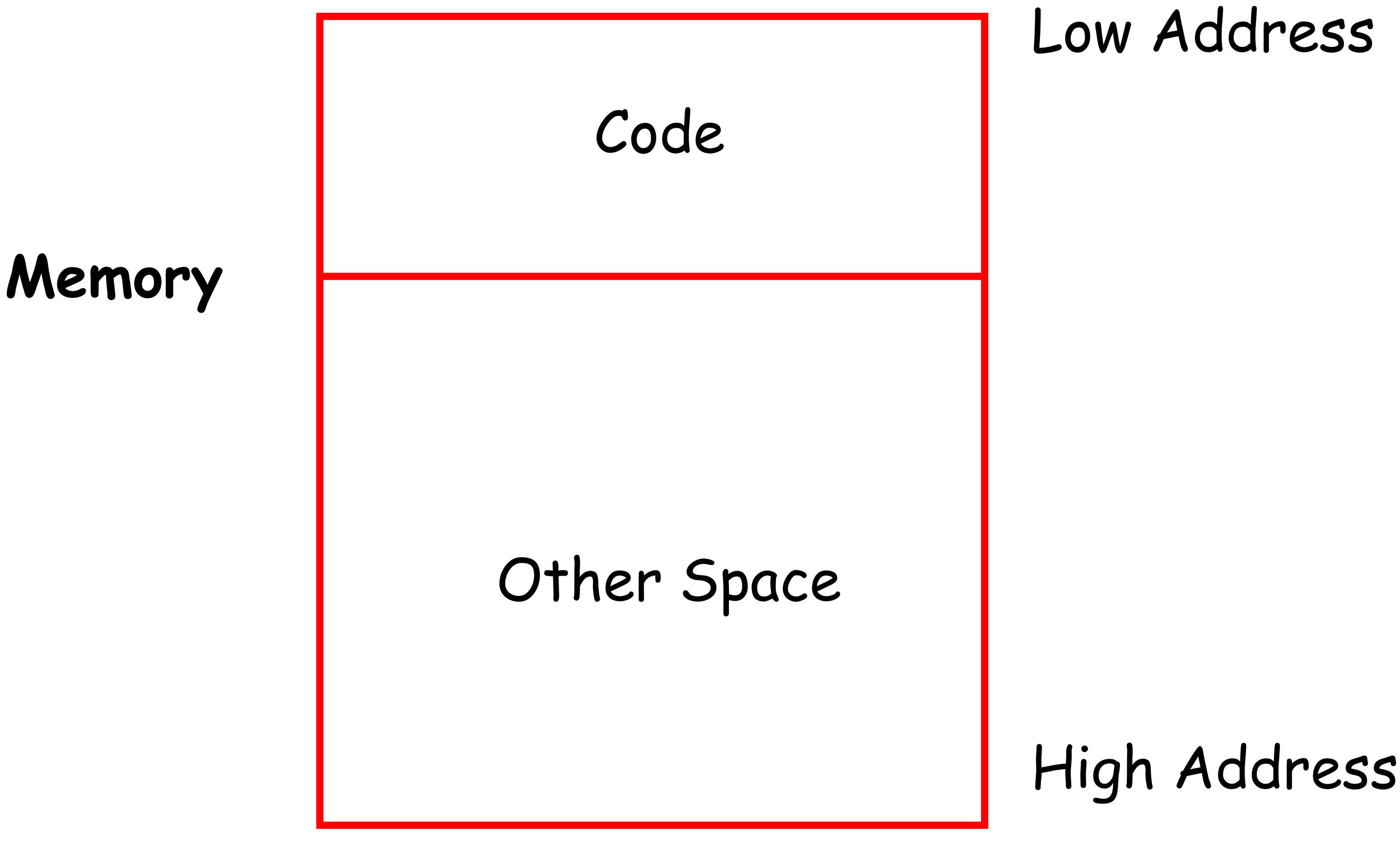
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system

- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



Notes

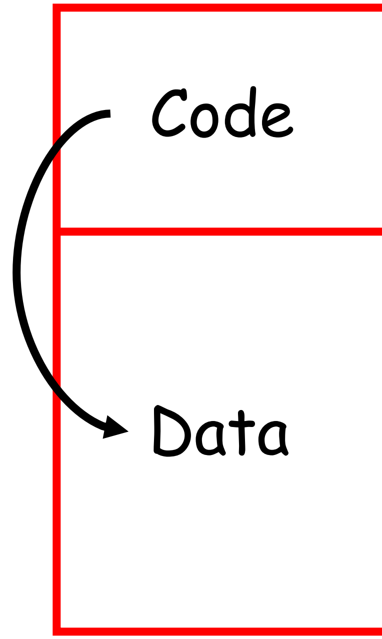
- By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

- These pictures are simplifications
  - E.g., not all memory need be contiguous

What is Other Space?

- Holds all data for the program
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area



Code Generation Goals

- Two goals:
  - Correctness
  - Speed

- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
   - No concurrency
2. When a procedure is called, control eventually returns to the point immediately after the call
   - No exceptions

Activations

- An invocation of procedure P is an *activation* of P

- The *lifetime* of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures P calls

Lifetimes of Variables

- The *lifetime* of a variable x is the portion of execution in which x is defined

- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope (i.e., the portion of the program text in which a variable is visible) is a static concept

Activation Trees

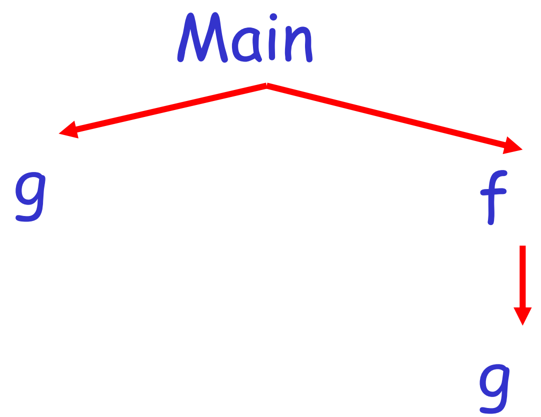
- Assumption (2) requires that when P calls Q, then Q returns before P does

- Lifetimes of procedure activations are properly nested

- Activation lifetimes can be depicted as a tree

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



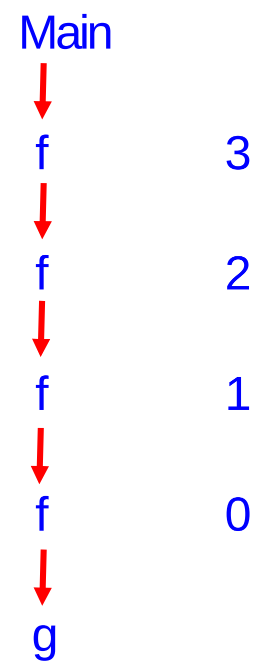
Example 2

```
Class Main {
  g() : Int { 1 };
  f(x:Int):  Int { if x = 0 then g() else f(x - 1) fi};
  main(): Int {{f(3); }};
}
```

What is the activation tree for this example?

Example 2

```
Class Main {
   g() : Int { 1 };
   f(x:Int):  Int { if x = 0 then g() else f(x - 1) fi};
   main(): Int {{f(3); }};
}
```



Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```

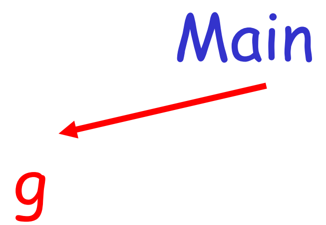
Main

Stack

Main

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



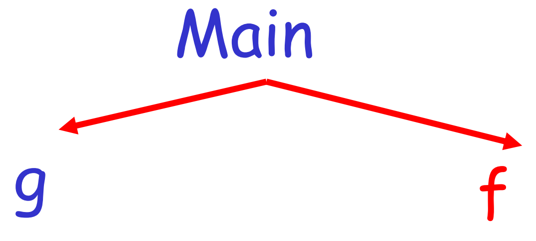
Stack

Main

g

Example

```
Class Main {
   g() : Int { 1 };
   f(): Int { g() };
   main(): Int {{ g(); f(); }};
}
```



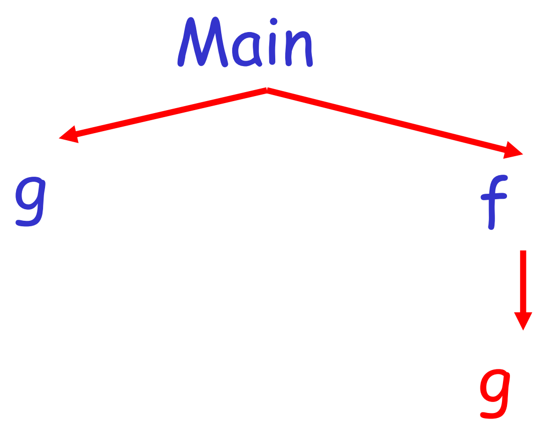
Stack

Main

f

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



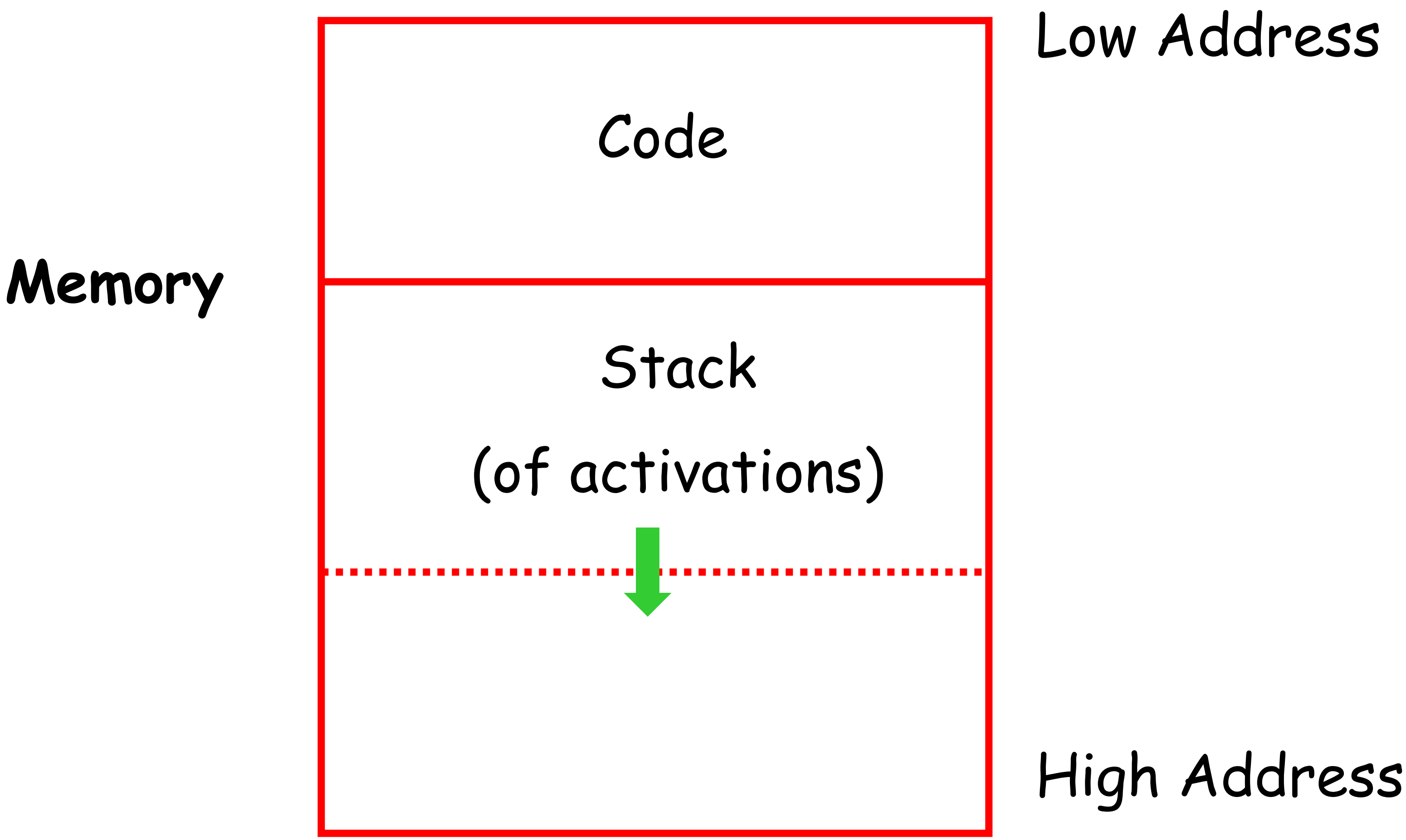
Stack

Main

f

g

Revised Memory Layout



Activation Records

- The information needed to manage one procedure activation is called an *activation record (AR)* or *frame*

- If procedure F calls G, then G's activation record contains a mix of info about F and G.

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes. G's AR contains information needed to resume execution of F.

- G's AR may also contain:
  - G's return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G's local variables

The Contents of a Typical AR for *G*

- Space for *G*' s return value
- Actual parameters
- Pointer to the previous activation record
  - The *control link;* points to AR of caller of *G*
- Machine status prior to calling *G*
  - Contents of registers & program counter
  - Local variables
- Other temporary values

Example 2

```
Class Main {
   g() : Int { 1 };
   f(x:Int):Int {if x=0 then g() else f(x - 1)(**)fi};
   main(): Int {{f(3); (*)
 }};}
```

AR for f:

result
argument
control link
return address

Notes

- Main has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes

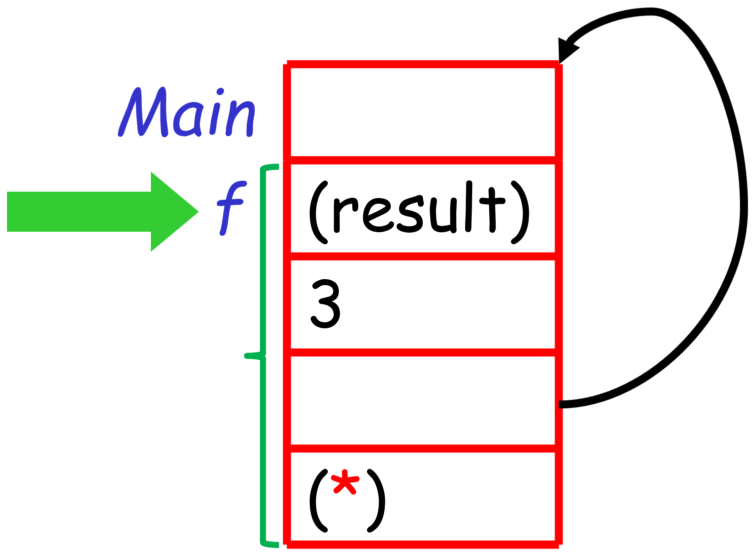
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

The Main Point

The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record

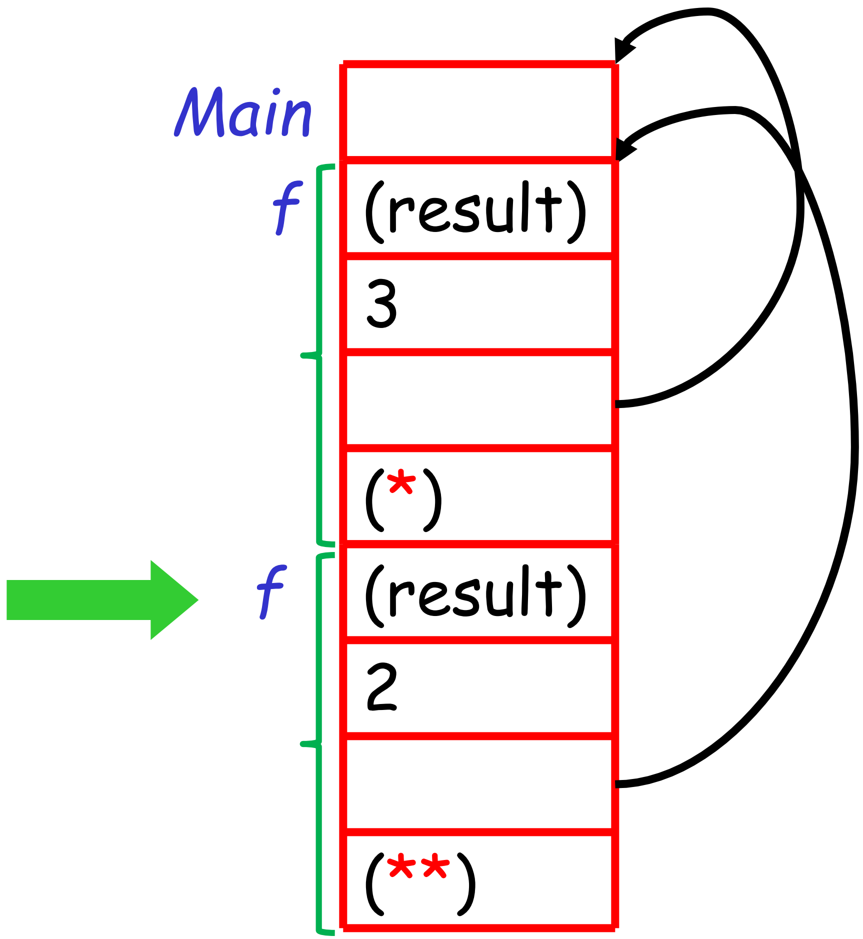
Thus, the AR layout and the code generator must be designed together!

Stack After First Call to f



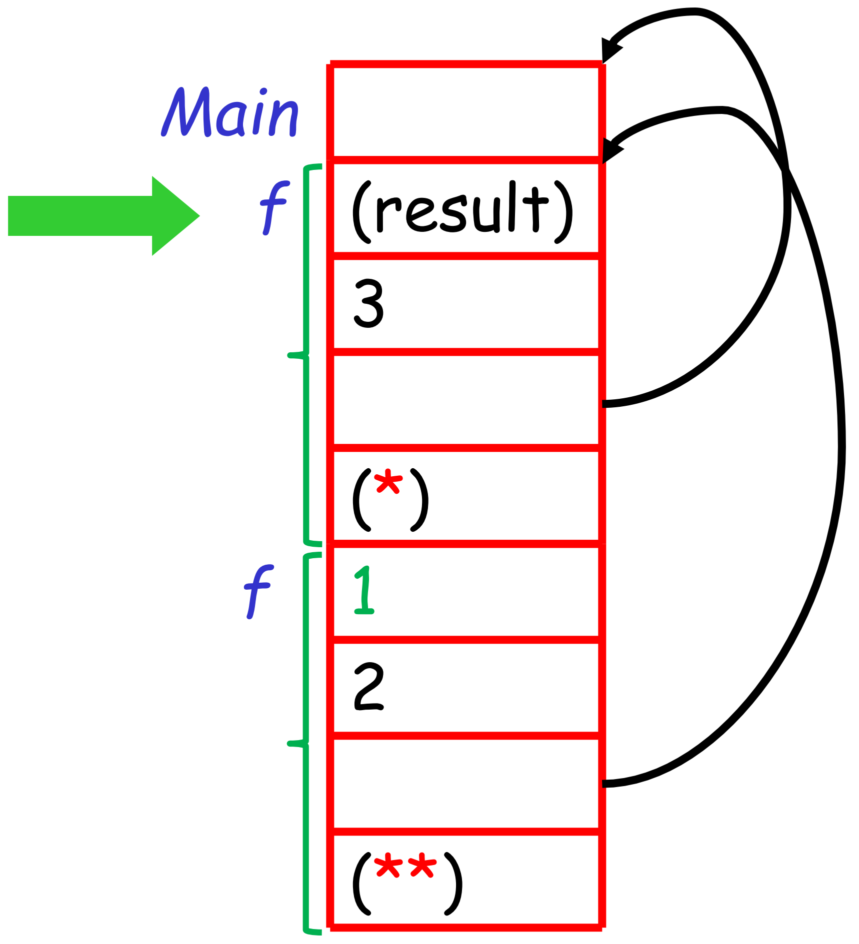

```
Class Main {
    g() : Int { 1 };
    f(x:Int):Int {if x=0 then
    g() else f(x - 1)(**)fi};
    main(): Int {{f(3); (*)
}};}
```

Stack After Second Call to *f*



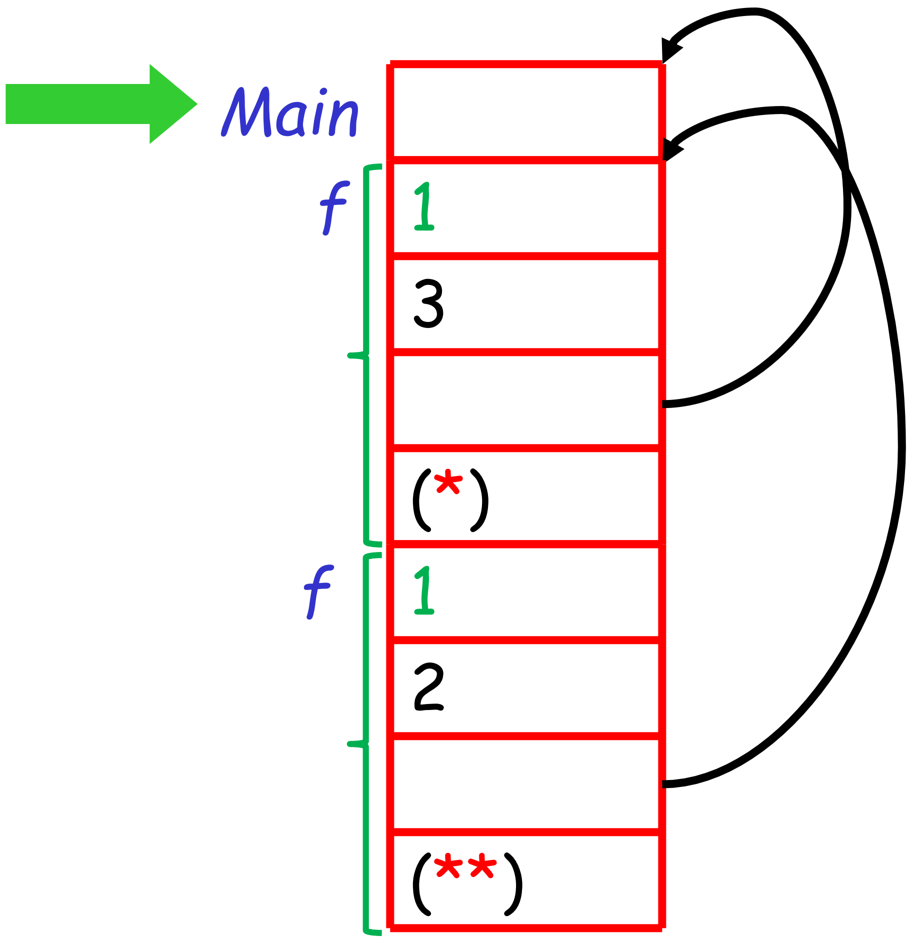

```
Class Main {
    g() : Int { 1 };
    f(x:Int):Int {if x=0 then
    g() else f(x - 1)(**)fi};
    main(): Int {{f(3); (*)
}};}
```

Stack After Return from the 2nd Call to *f*




```
Class Main {
    g() : Int { 1 };
    f(x:Int):Int {if x=0 then
    g() else f(x - 1)(**)fi};
    main(): Int {{f(3); (*)
}};}
```

Stack After Return from the 1st Call to f




```
Class Main {
    g() : Int { 1 };
    f(x:Int):Int {if x=0 then
    g() else f(x - 1)(**)fi};
    main(): Int {{f(3); (*)
}};}
```

Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

Discussion (Cont.)

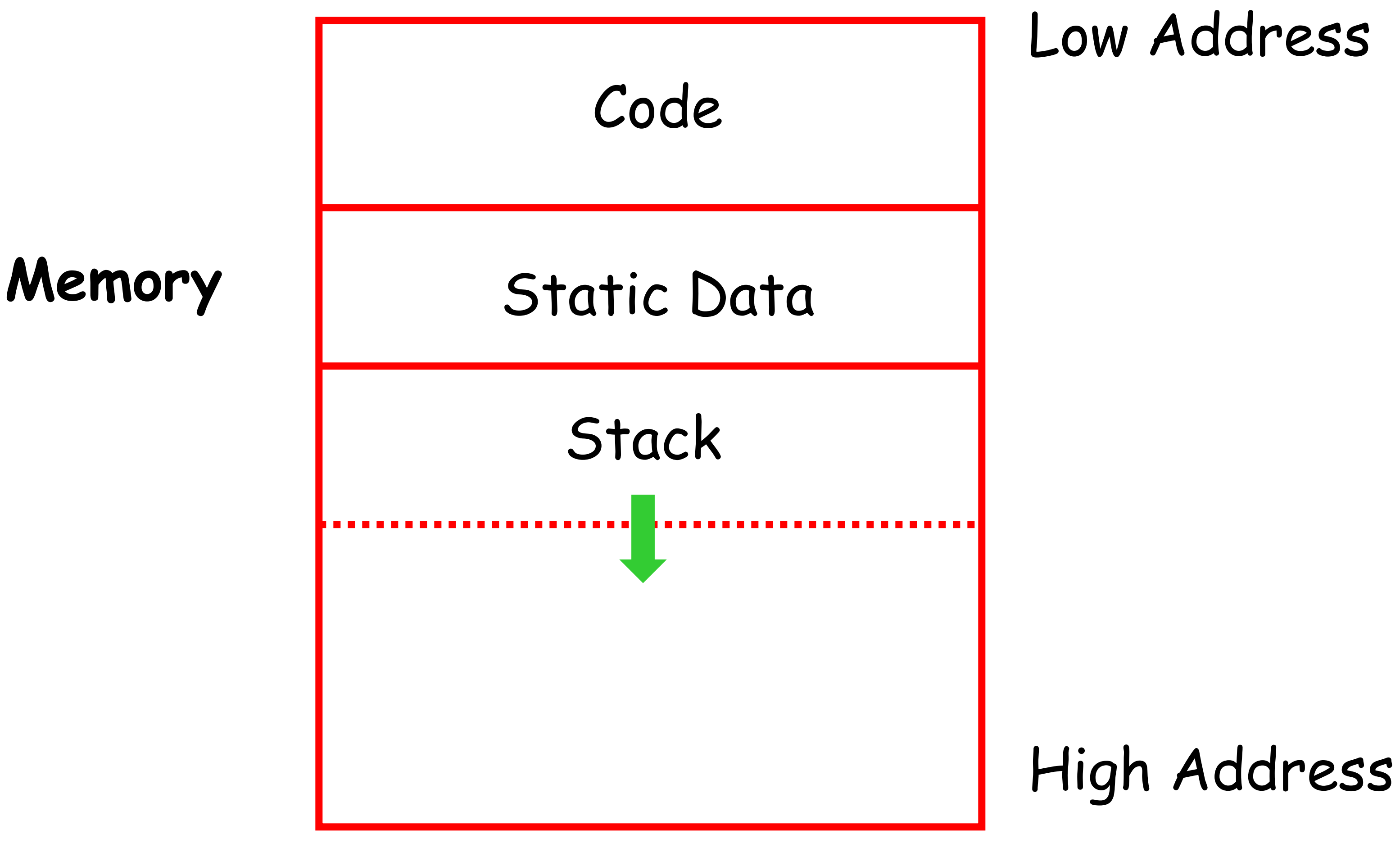
- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

Globals

- All references to a global variable point to the same object
  - Can't store a global in an activation record

- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values

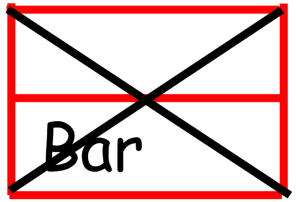
Memory Layout with Static Data



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

foo



method foo() { new Bar }

The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use a *heap* to store dynamic data

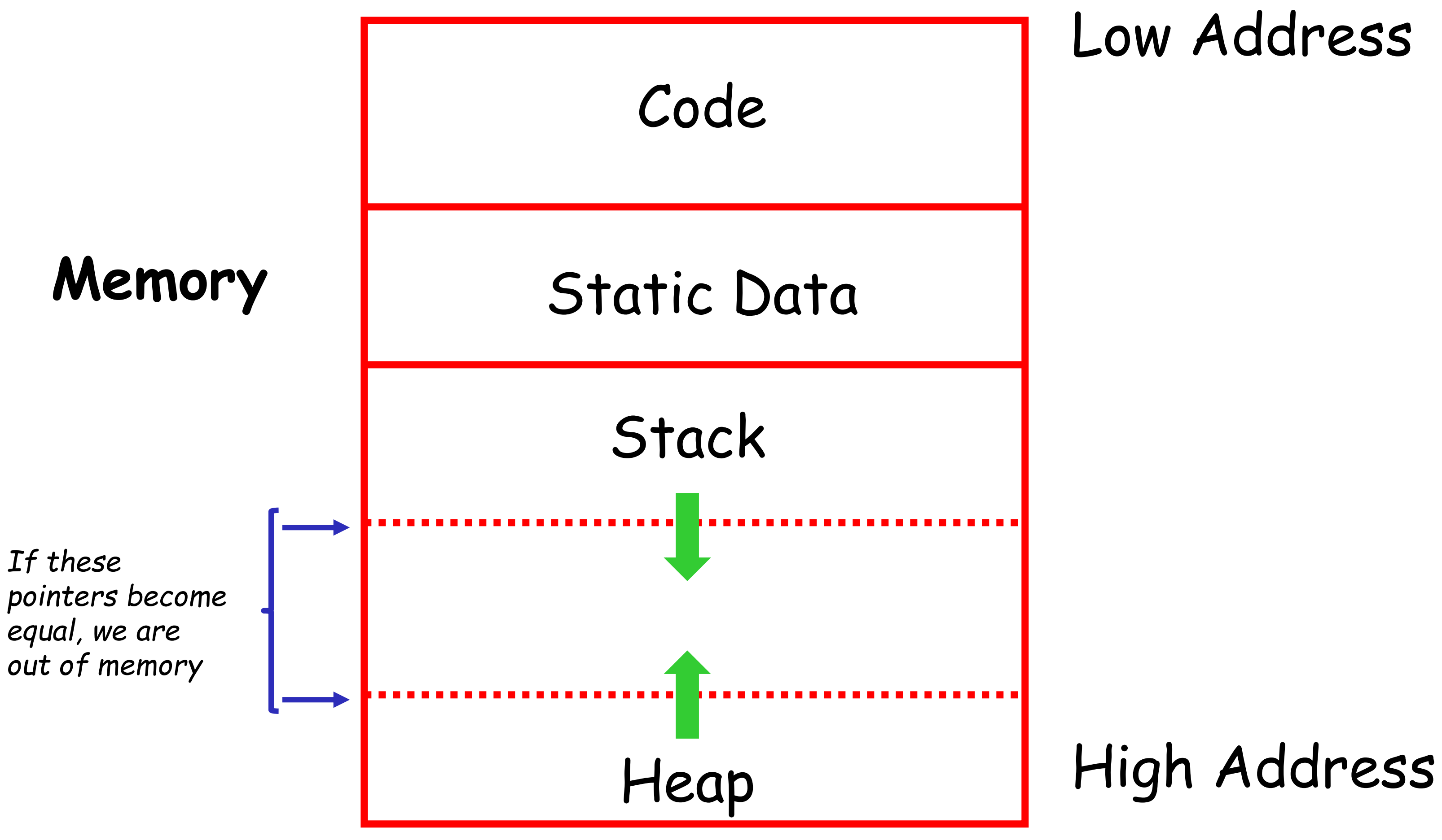
Notes

- The code area contains object code
  - For many languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

Notes (Cont.)

- Both the heap and the stack grow
- Must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

Memory Layout with Heap



Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is *alignment*

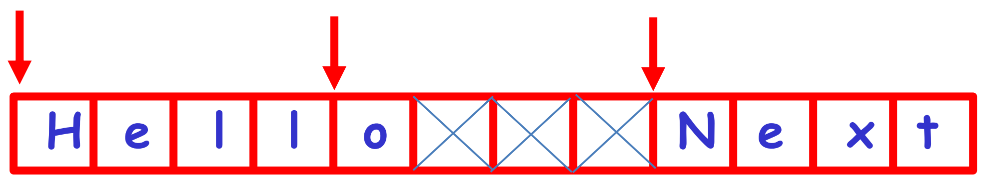
Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is *word aligned* if it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Alignment (Cont.)

- Example: A string "Hello"

  Takes 5 characters (without a terminating \0)



- To word align next datum, add 3 "padding" characters to the string

- The padding is not part of the string, it's just unused memory

Activation Records (more details)

return value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

The field for saved machine status holds information about the state of the machine before the procedure is called.

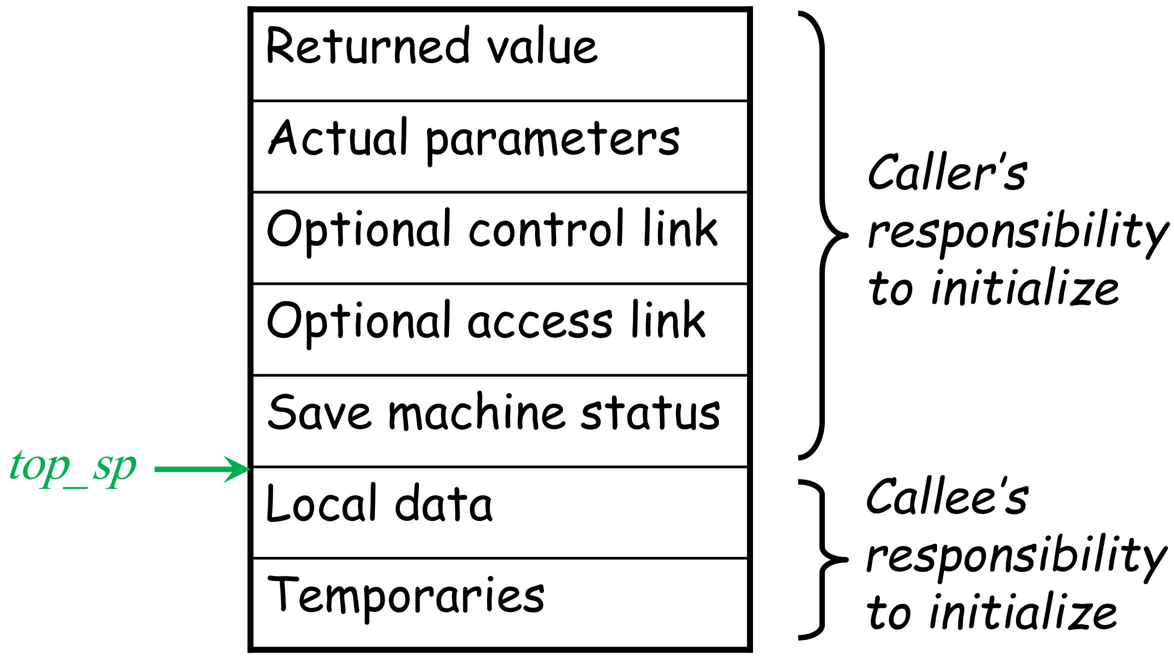
The field of local data holds data that local to an execution of a procedure..

Temporary variables is stored in the field of temporaries.

Activation Records (more details)

- *Activation records* (subroutine frames) on the run-time stack hold the state of a subroutine
- *Calling sequences* are code statements to create activations records on the stack and enter data in them
  - Caller's calling sequence enters actual arguments, control link, access link, and saved machine state
  - Callee's calling sequence initializes local data
  - Callee's return sequence enters return value
  - Caller's return sequence removes activation record

Activation Records (more details)



Calling sequence is divided between Caller and Callee

Most tasks are devoted to the Callee. Why?

Access to Nonlocal Data

- Scope rules of a language determine the treatment of references to nonlocal names.

- **Lexical Scope (Static Scope)**
  Determines the declaration that applies to a name by examining the program text alone at compile-time.
  Most-closely nested rule is used.
  Pascal, C, ..

- **Dynamic Scope**
  Determines the declaration that applies to a name at run-time.
  Lisp, APL, ...

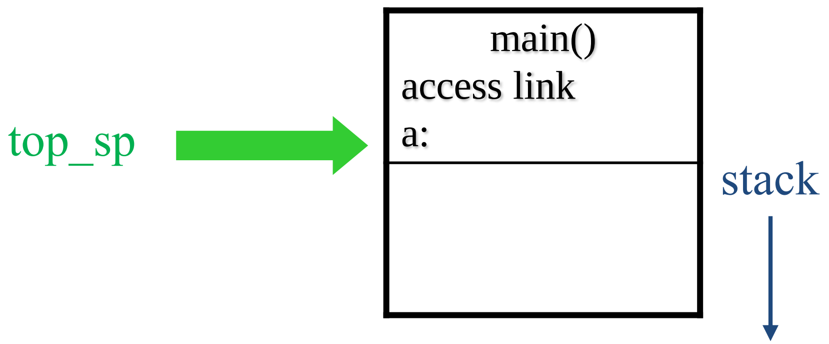
Accessing Nonlocal Data using Access Links

- If procedure F is (immediately) located inside procedure G, every time F is invoked, the access link in the AR of F will be set to point to the access link in the AR of G

- If there are more than one ARs of G in the run time stack, the access link is set to point to the most recent AR of G.

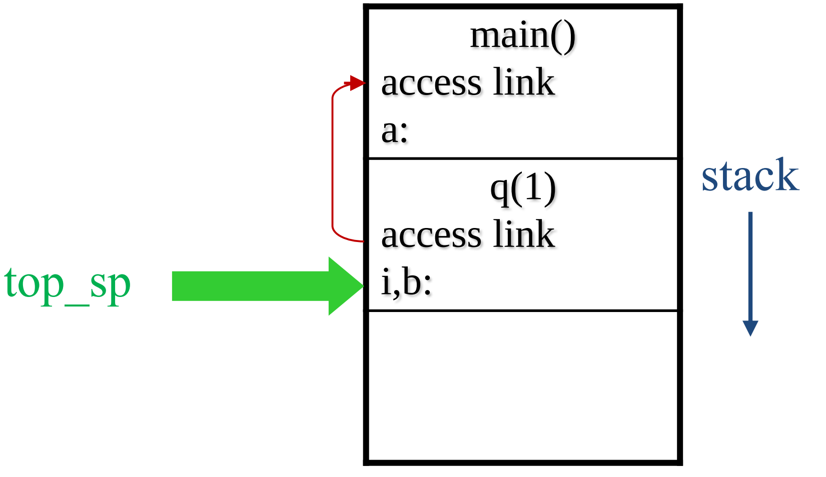
Access Links (Example)

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



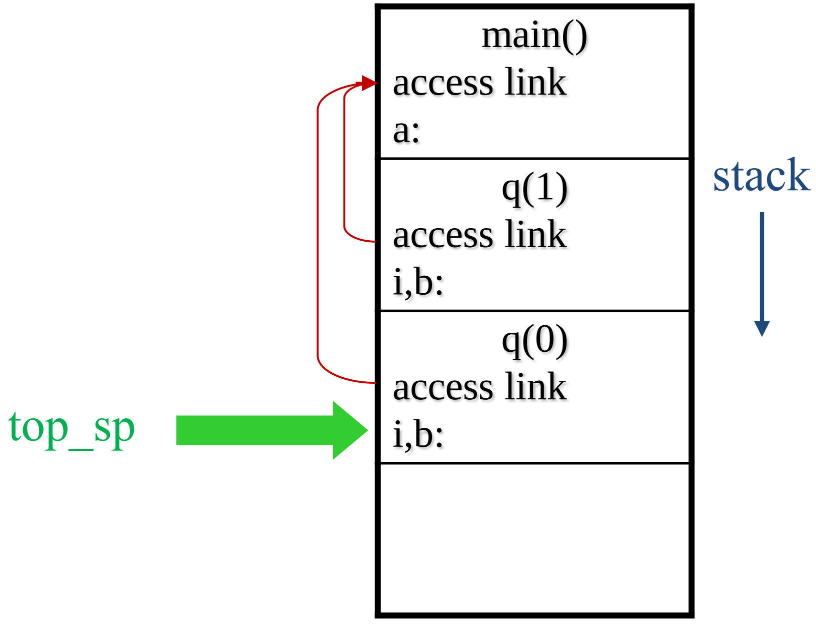
Example (cont.), when q(1) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



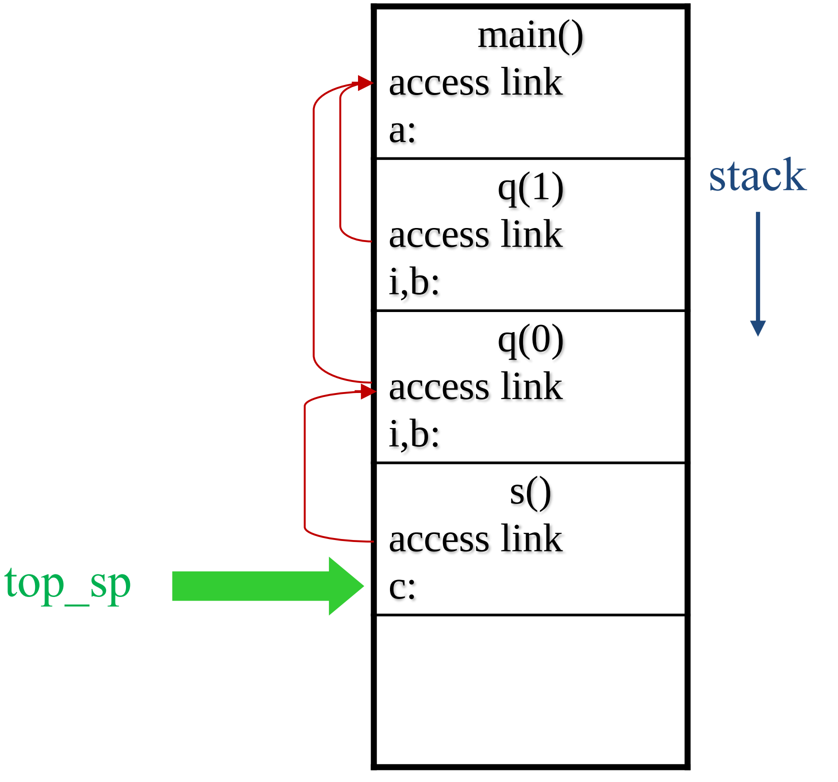
Example (cont.), when q(0) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



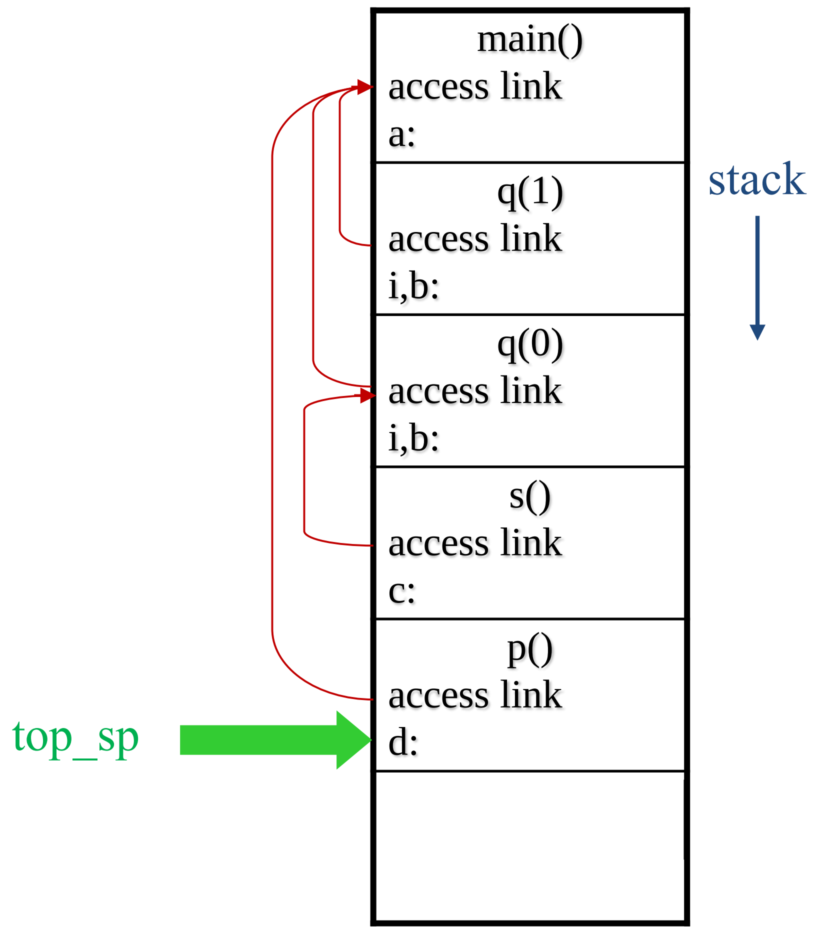
Example (cont.), when s() is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



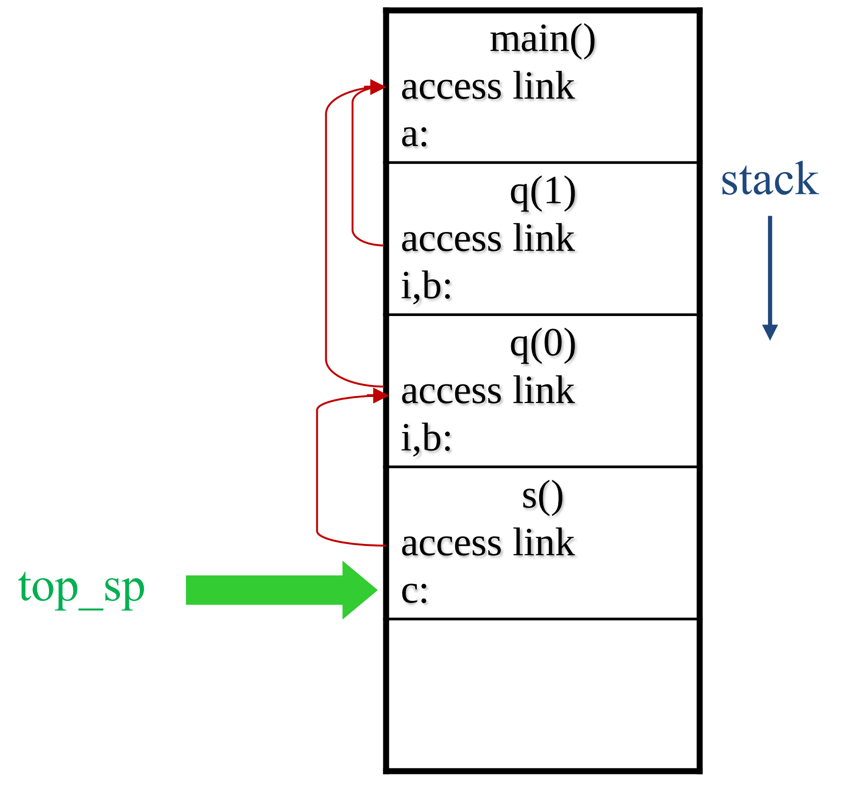
Example (cont.), when p() is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



Example (cont.), after returning to s()

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



Accessing Nonlocal Data using Access Links

To implement access to nonlocal data a in procedure q, the compiler generates code to traverse n_q - n_a access links to reach the activation record where a resides

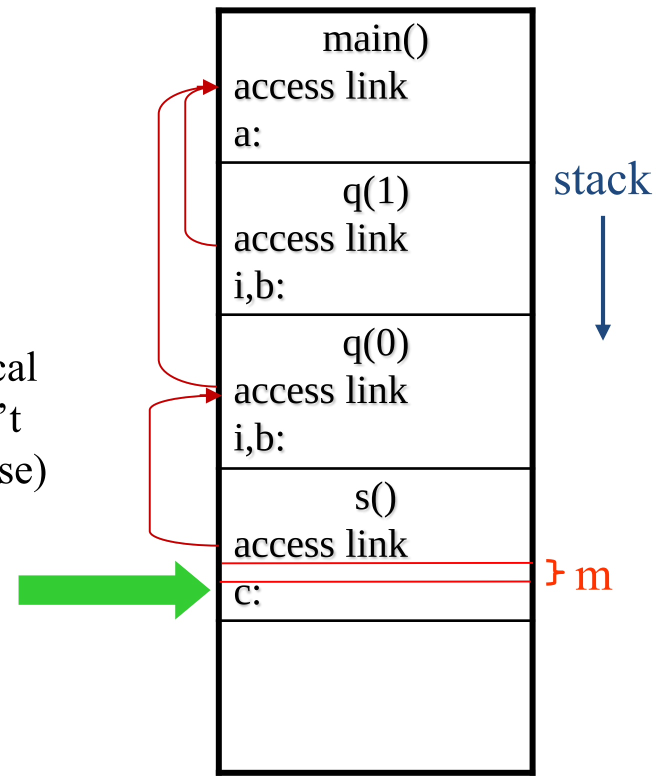
- n_q *is the nesting depth of procedure* q
- n_a *is the nesting depth of the procedure containing* a

Example (cont.), Access to variables (via A.L.)

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
    ──► c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```

m is the size of other fields between access link and local data (e.g., m=1, if there isn't any other field between these)

top_sp



Addresses of variables are computed at compile time

- address c: top_sp + #0
- address b: @(top_sp - #m) + #m + #1
- address a: @(@(top_sp - #m)) + #m + #0

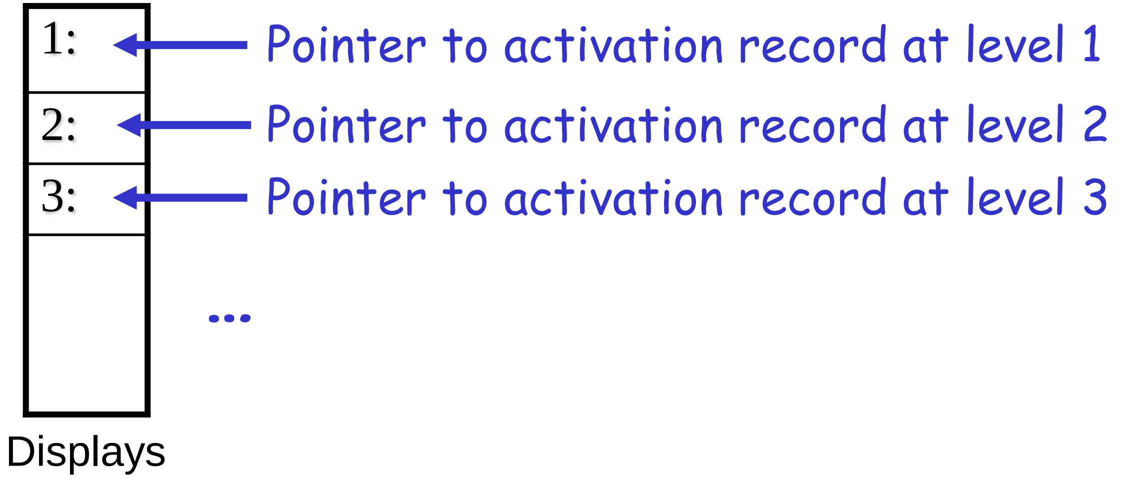
Accessing Nonlocal Data using Displays

- An array of pointers to activation records can be used to access activation records.

- This array is called as *displays*.

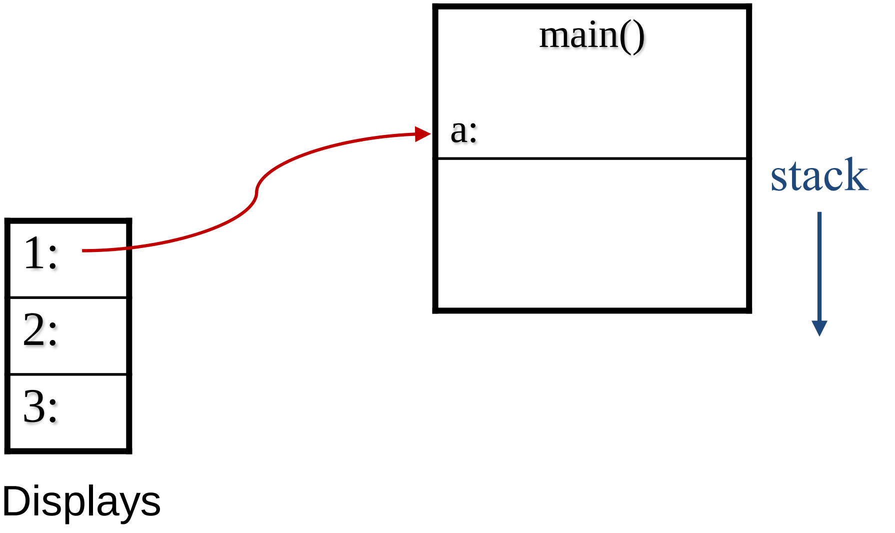
- For each level, there will be an array entry.

- The number of required entries is known at the compile time.



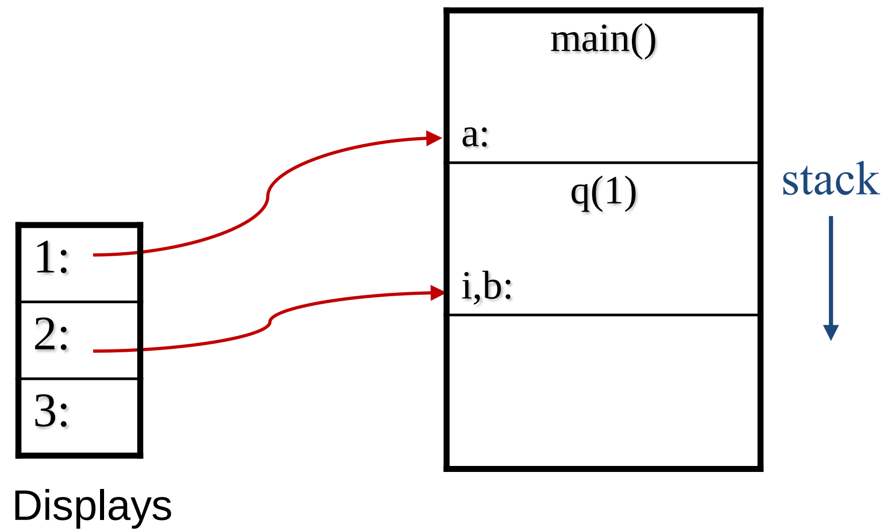
Displays (Example)

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



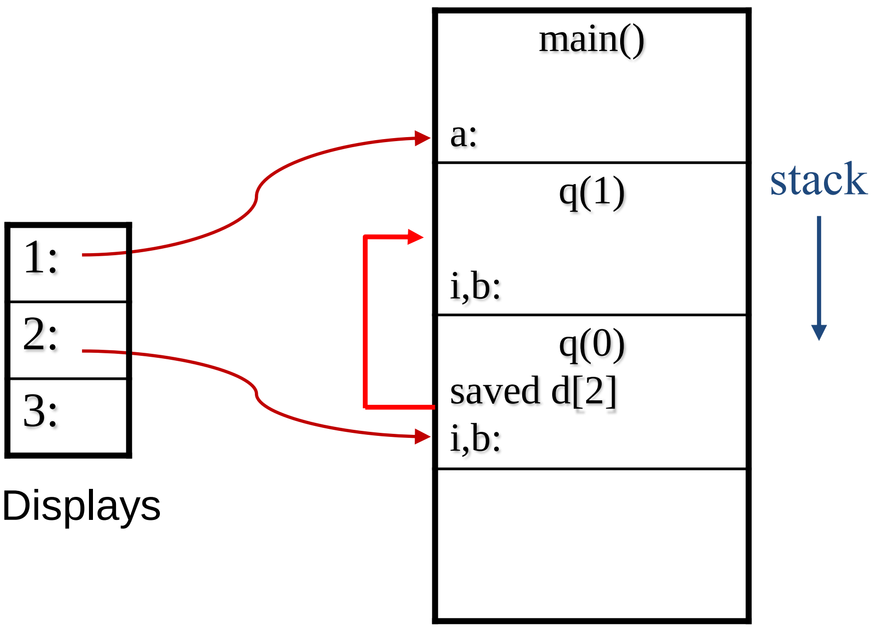
Example (cont.), when q(1) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



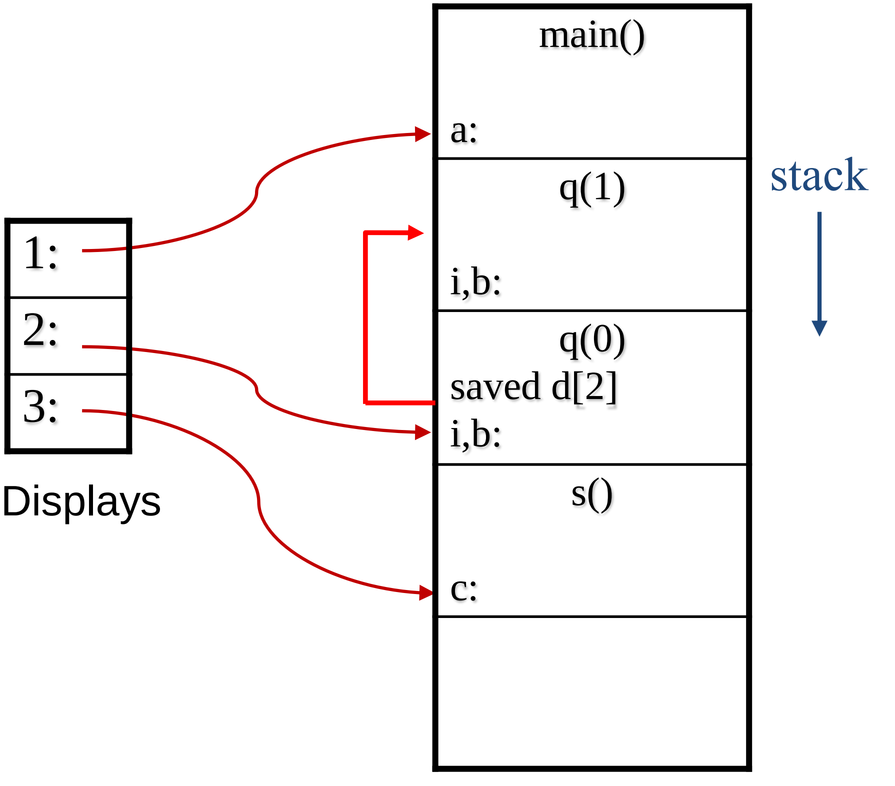
Example (cont.), when q(0) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



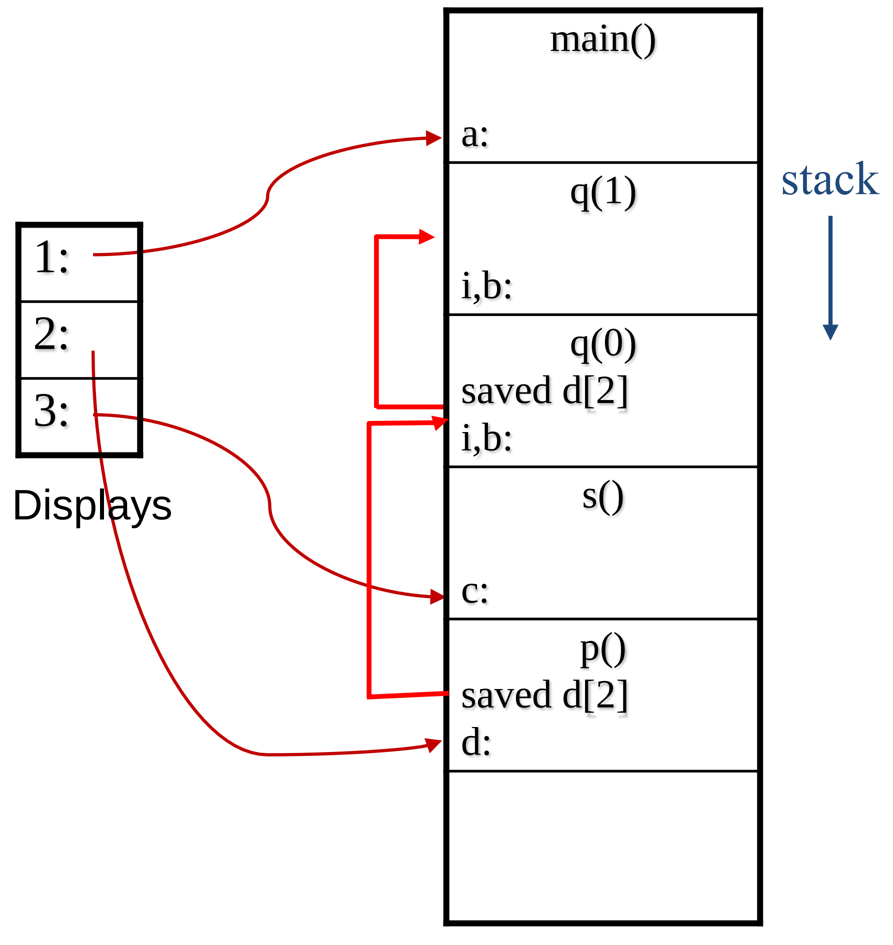
Example (cont.), when s() is executed

```
program main();
   var a:int;
   procedure p();
      var d:int;
      a:=1;
   end p;
   procedure q(i:int);
      var b:int;
      procedure s();
         var c:int;
         p();
         c := b + a
      end s;
      if (i<>0) then q(i-1)
      else s();
   end q;
   q(1);
end main;
```



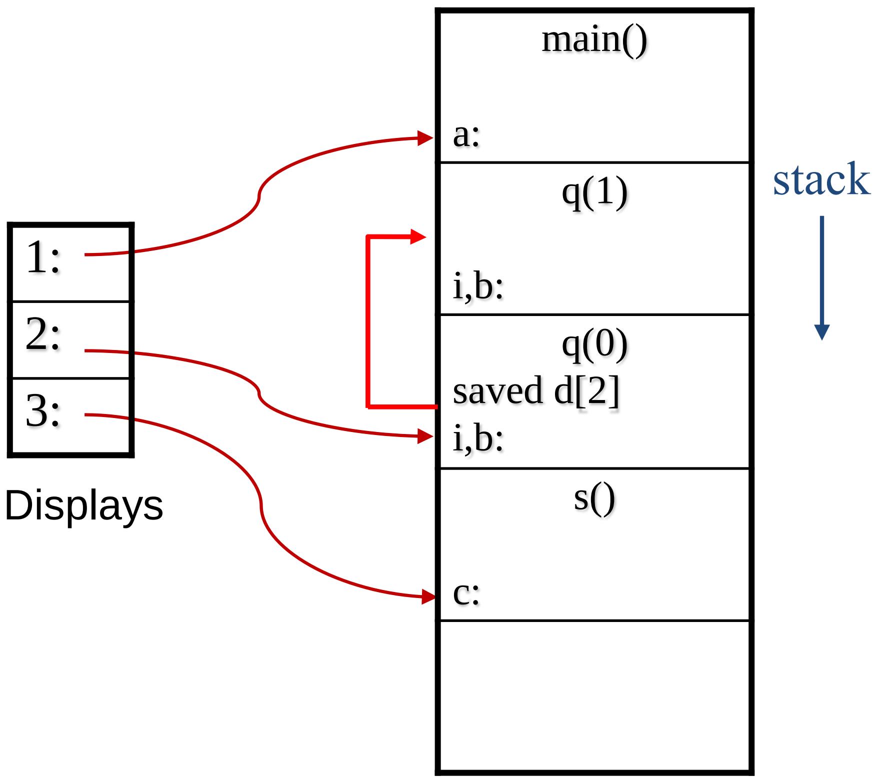
Example (cont.), when p() is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



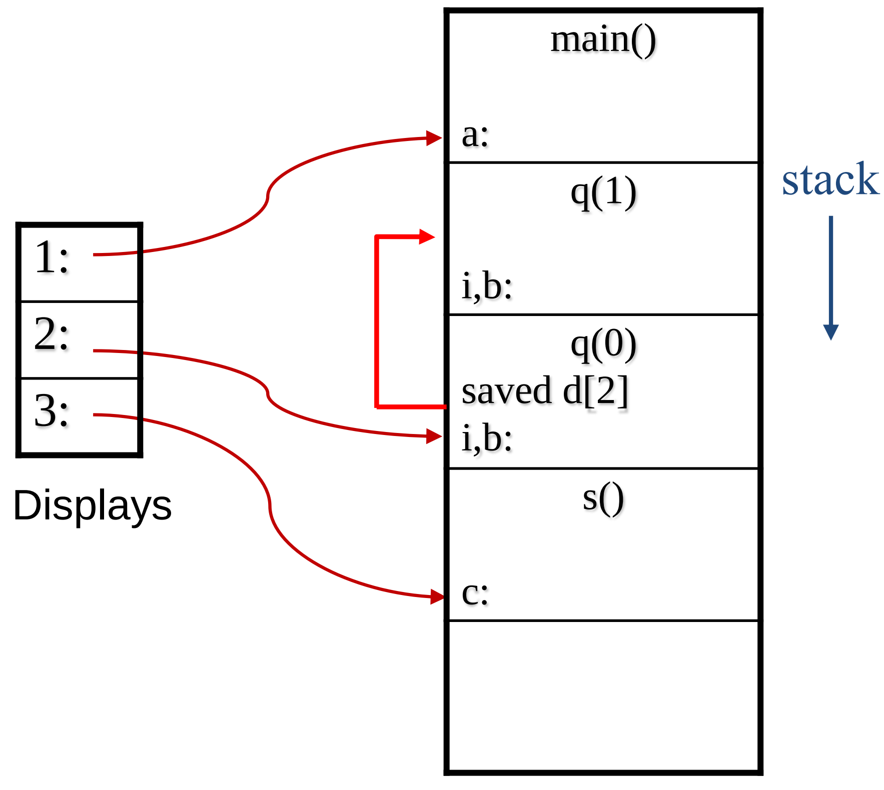
Example (cont.), after returning to s()

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```



Example (cont.), Access to vars. (via Displays)

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b:int;
        procedure s();
            var c:int;
            p();
            c := b + a
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(1);
end main;
```

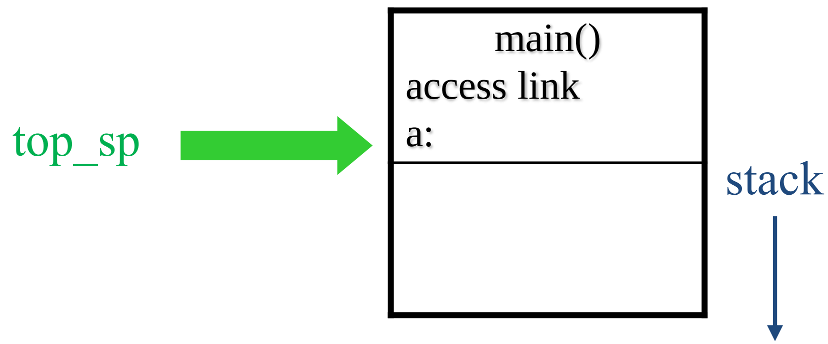


Computed at the Compile time:

- address c: d[3] + #0
- address b: d[2] + #1
- address a: d[1] + #0

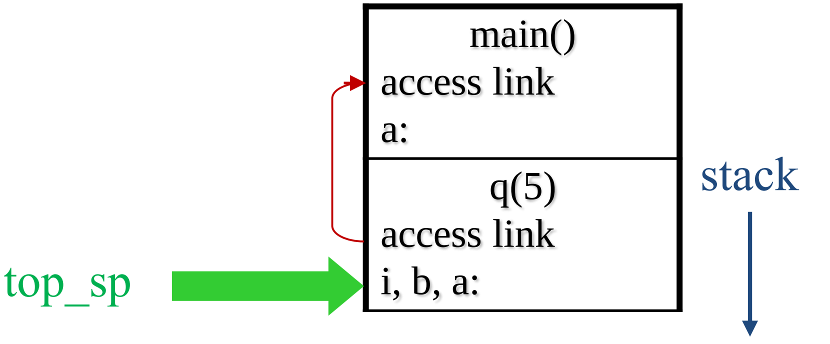
Variable Length Data (Example)

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```



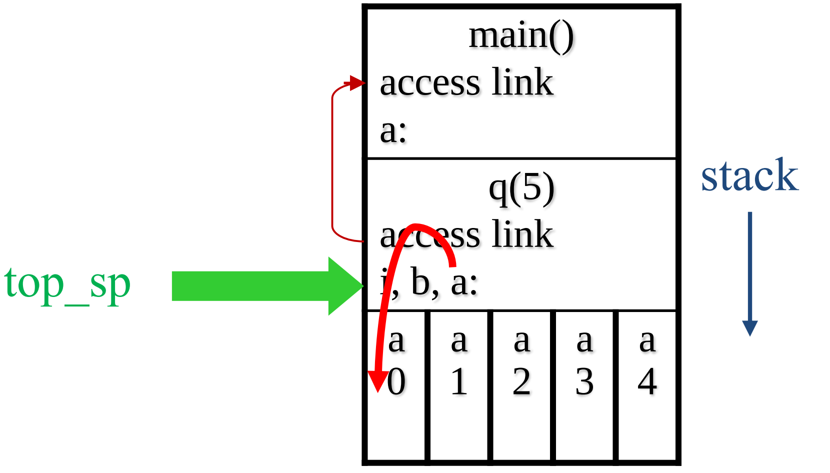
Example (cont.), when q(5) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```



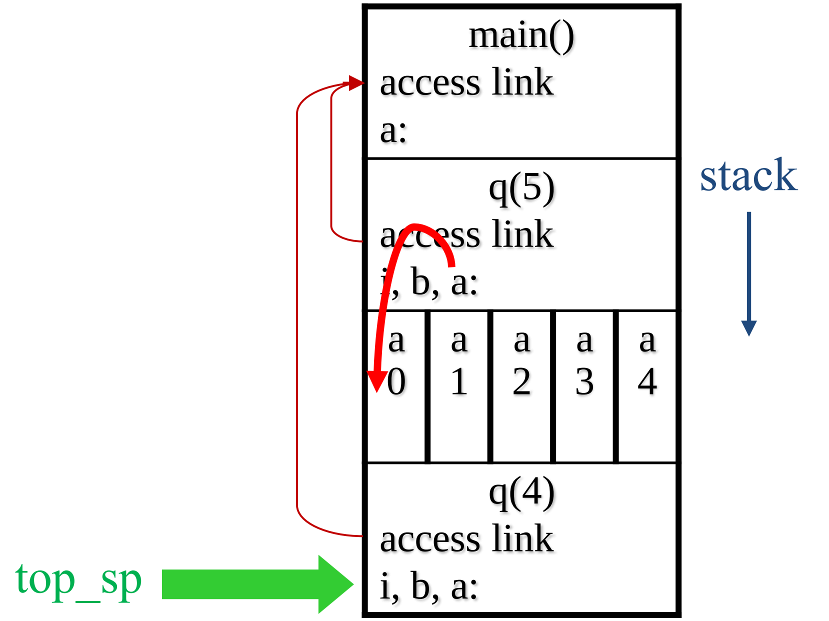
Example (cont.), when a(5) is allocated

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```



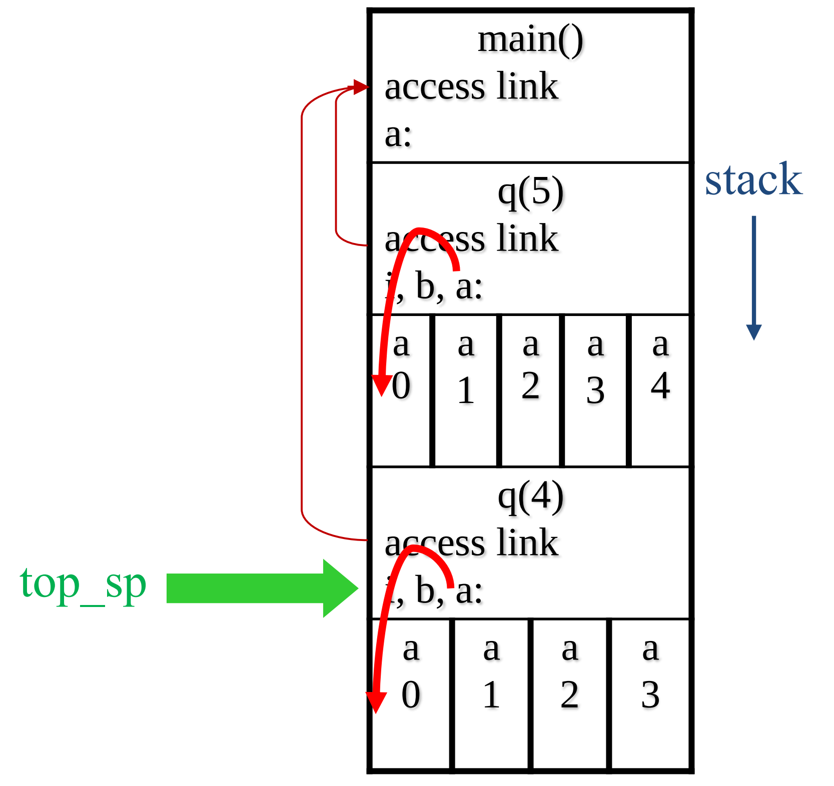
Example (cont.), when q(4) is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```



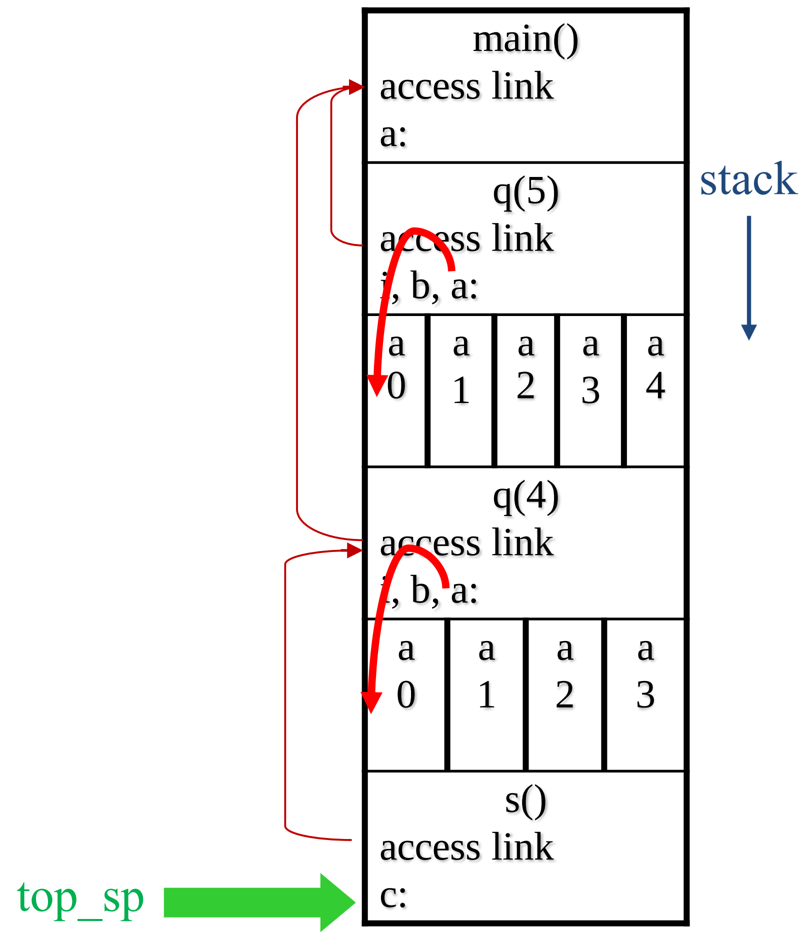
Example (cont.), when a(4) is allocated

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```



Example (cont.), when s() is executed

```
program main();
  var a:int;
  procedure p();
    var d:int;
    a:=1;
  end p;
  procedure q(i:int);
    var b, a[i] :int;
    procedure s);
      var c:int;
      c := a[3]
      p();
    end s;
    if (i<>0) then q(i-1)
    else s();
  end q;
  q(5);
end main;
```

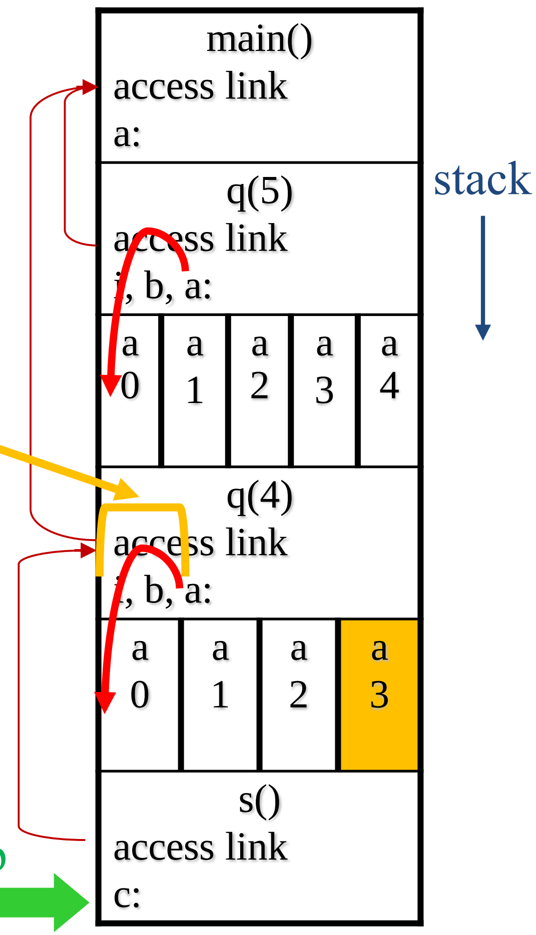


Example (cont.), when s() is executed

```
program main();
    var a:int;
    procedure p();
        var d:int;
        a:=1;
    end p;
    procedure q(i:int);
        var b, a[i] :int;
        procedure s);
            var c:int;
            c := a[3]
            p();
        end s;
        if (i<>0) then q(i-1)
        else s();
    end q;
    q(5);
end main;
```

m is the size of space between access link and local data (e.g., m=1, if these two fields are adjacent)

distance of a from start of local data



distance of c from start of local data

index

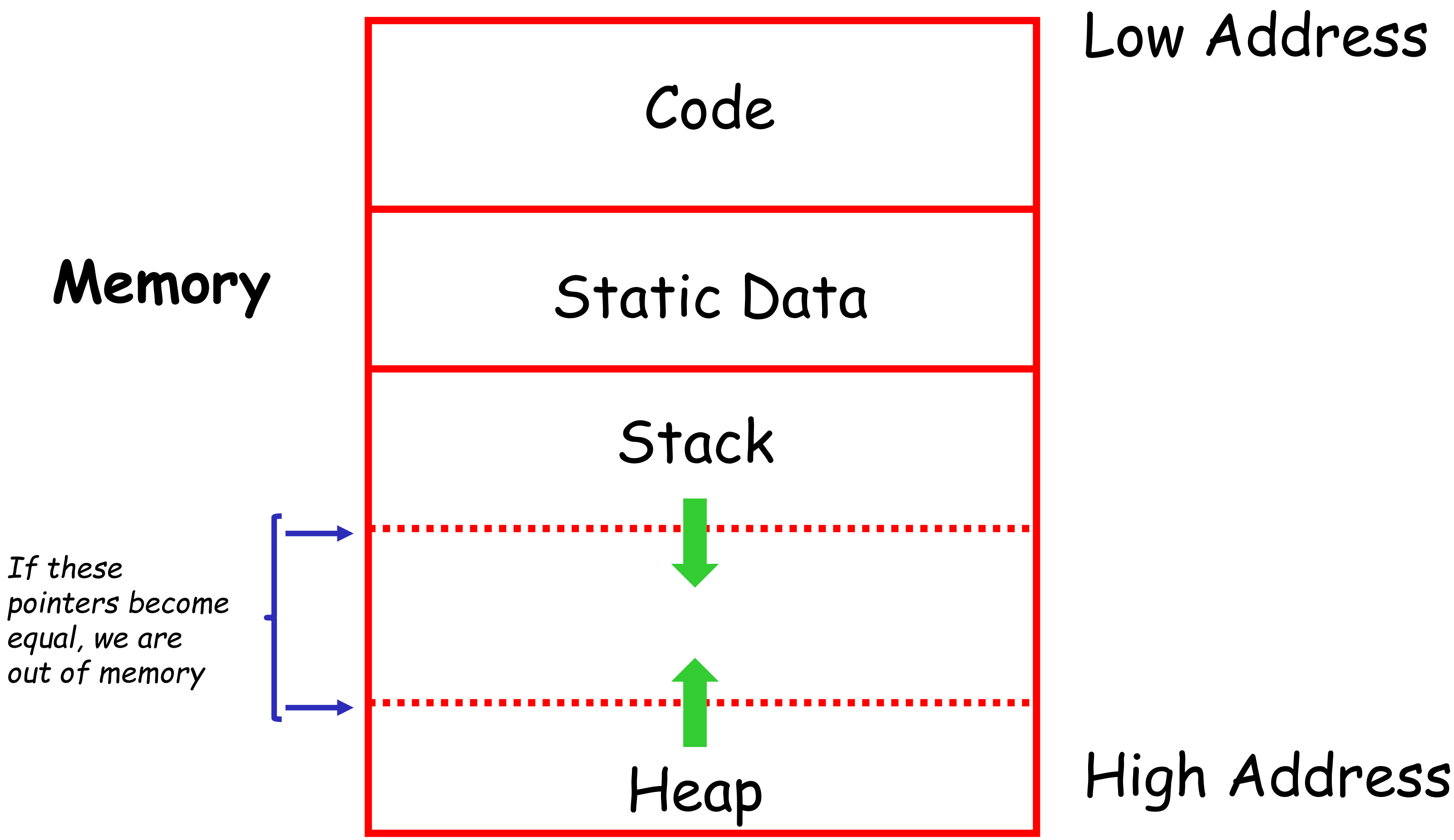
top_sp

Address of c: top_sp + #0

Address of a[3]: @(@(top_sp - #m) + #m + #2)) + (#3 * #1)

Size of each cell

Memory Layout with Heap (revisited)



Heap Manager

- Requests for Heap are handled via calls to the heap manager.

- There is generally some time and space overhead associated with using a heap manager.

- It is helpful to handle small records of a predictable size as a special case.

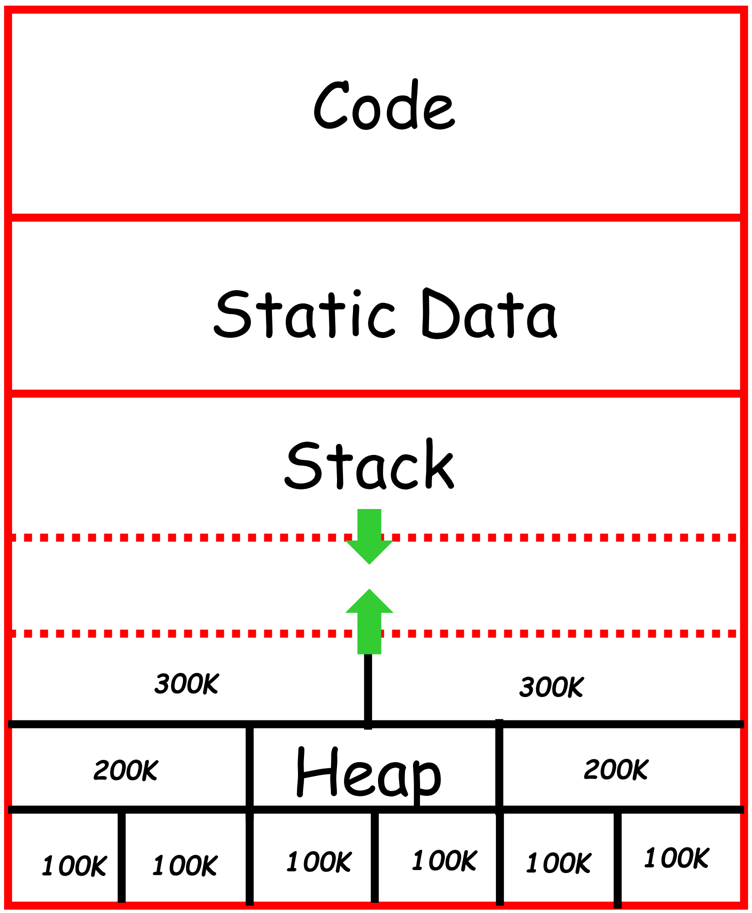
Using an Internal Heap Manager

1. For each size of interest, keep a linked list of free blocks of that size.

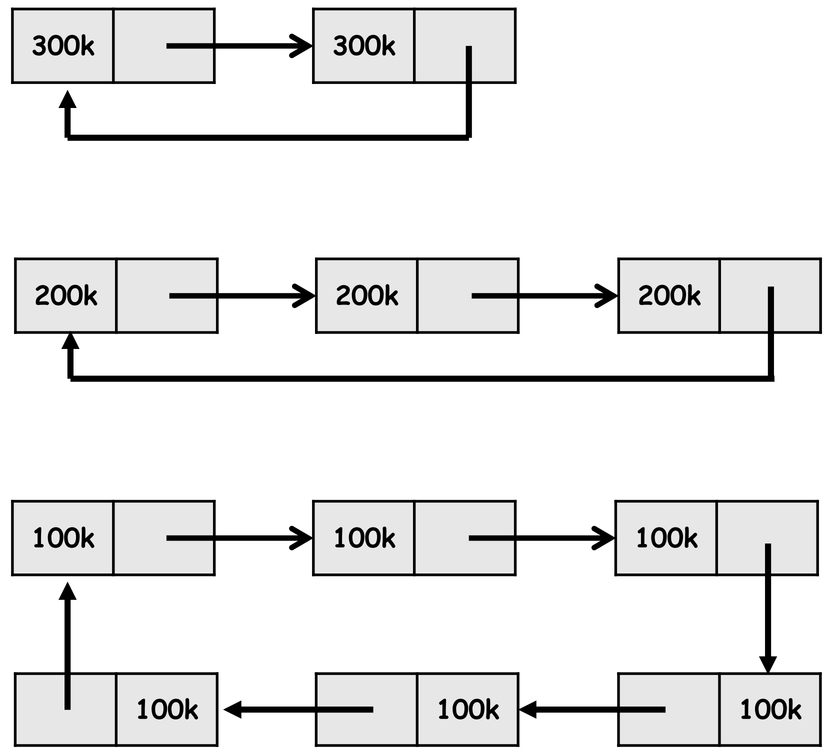
2. If possible, fill a request for size s with a block of size s', where s' is the smallest size greater than or equal to s. When the block is eventually deallocated, it is returned to the linked list it came from.

3. For large blocks of storage call the heap manager.

Using an Internal Heap Manager (Cont.)



Block Addresses



Using an Internal Heap Manager (Cont.)

- This results in fast allocation and deallocation of small amounts of storage.
- Because, taking and returning a block from a linked list are efficient operations.
- For large amounts of storage, it takes some time to use up the storage, so the time taken by the heap manager is often negligible.

Question?

The powerOfTwo() function, shown to the right, returns true if its argument is a power of two, false otherwise. What is the activation tree for powerOfTwo(4)?

```
isEven(x:Int) : Bool { x % 2 == 0 };
isOne(x:Int) : Bool { x == 1 };
powerOfTwo(x:Int) : Bool {
   if isEven(x) then powerOfTwo(x / 2)
   else isOne(x)
};
```

```
powerOfTwo
    |
  isEven
    |
powerOfTwo
    |
  isEven
    |
powerOfTwo
    |
  isEven
    |
  isOne
```

```
      powerOfTwo
      /        \
  isEven    powerOfTwo
             /       \
         isEven    powerOfTwo
                       |
                     isOne
```

```
powerOfTwo
    |
powerOfTwo
    |
powerOfTwo
    |
  isOne
```

```
      powerOfTwo
      /        \
  isEven    powerOfTwo
             /       \
         isEven    powerOfTwo
                    /      \
                isEven    isOne
```

Question?

```
program main ()
    var a, b : int;
    procedure f (m : int);
        var v: int;
        procedure g (y : int);
            var i, j : int;
            f (y);
        end g;
        procedure p (z : int);
            var l, m : int;
            procedure q (i : int);
                var k : int;
                g (i - 1);
  ⇒           b := m + v;
            end q;
           if (z > 4 ) f (z) else q (z - 1)
        end p;
       if  m > 1 then p (3)
    end f;
    f (4)
end main;
```

Assume that in running the given program, the procedures are invoked in the order that follows: f (4), p (3), q (2), g(1), and f(1).

Draw the runtime stack at the end of these invocations. Assume that non-local variables are addressed using Access Links.

Assume there is any gap between access links and local data

What is the addresses of variables in assignment b := m + v in procedure q?

Question?

```
program main ()
    var a, b : int;
    procedure f (m : int);
        var v: int;
        procedure g (y : int);
            var i, j : int;
            f (y);
        end g;
        procedure p (z : int);
            var l, m : int;
            procedure q (i : int);
                var k : int;
                g (i - 1);
          ⇒    b := m + v;
            end q;
           if (z > 4 ) f (z) else q (z - 1)
        end p;
       if  m > 1 then p (3)
    end f;
    f (4)
end main;
```

Assume that in running the given program, the procedures are invoked in the order that follows: f (4), p (3), q (2), g(1), and f(1).

Draw the runtime stack at the end of these invocations. Assume that non-local variables are addressed using Displays.

Assume there is any gap between access links and local data

What is the addresses of variables in assignment b := m + v in procedure q?