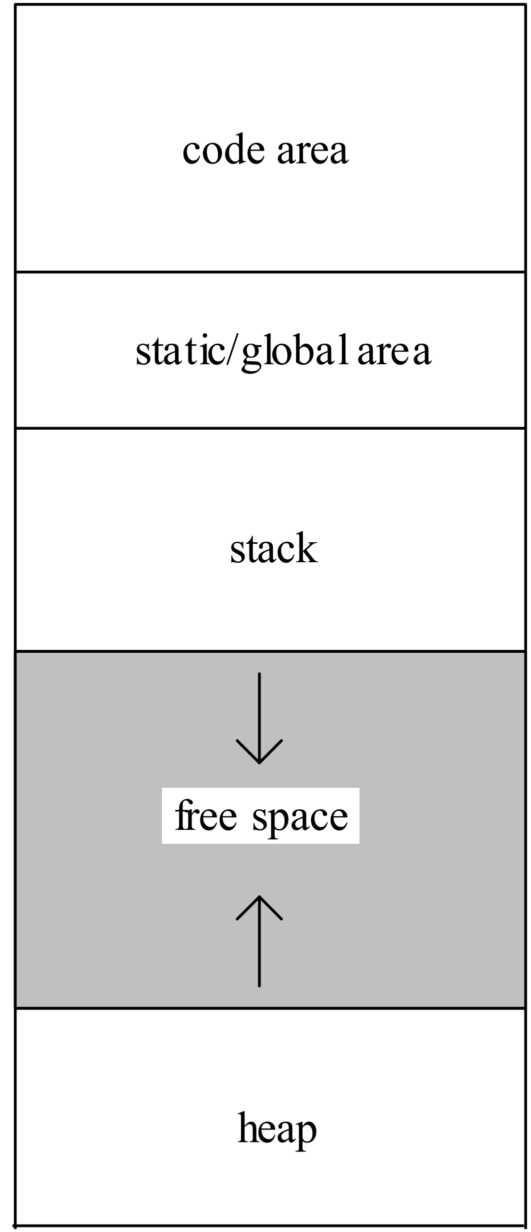
CIT 3136 - Week13 Lecture

Runtime Environments

- **During execution, allocation must be maintained by the generated code that is compatible with the scope and lifetime rules of the language.**
- **Typically there are three choices for allocating variables and parameters:**
  - **Assign them fixed locations in global memory ("static" allocation).**
  - **Put them on the processor stack.**
  - **Allocate them dynamically in memory managed by the program (the "heap").**

Schematic:



C uses all three:

- **Global storage for definitions at level 0 ("external" definitions) and statically declared locals.**
- **Stack storage for parameters and normal local variables.**
- **Heap storage for dynamically allocated data addressed through pointers (call to an allocator such as malloc required).**

Dynamic languages like Scheme and Smalltalk use mostly the heap:

- **All variables are implicitly pointers.**
- **Stack is used only to maintain the heap.**
- **Java is similar, except that simple vars are kept on the stack.**

FORTRAN and TINY use only global storage:

- **All variables are static (even global in TINY).**

What about C-Minus?

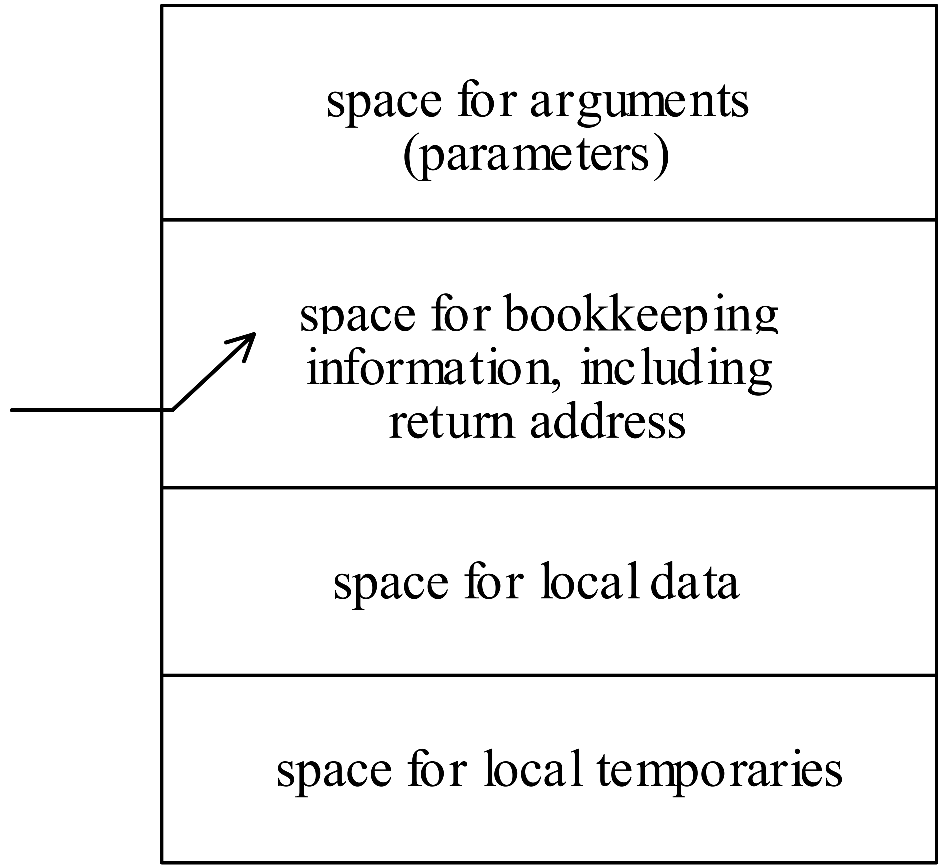
- **There are no pointers.**
- **Local variables must still be kept on the stack (we'll see why in a minute).**
- **Level 0 declarations can still be statically allocated.**
- **Thus, C-Minus can avoid having to manage a heap.**

Function calls:

- **Every call, no matter what the language, needs some allocated space to work correctly:**
  - **incoming parameter values**
  - **outgoing return value**
  - **return address**
  - **space for local and temporary variables**
- **This space is called an *activation record*.**
- **Recursion forces the activation record to be kept on the stack, since several activations of the same function can exist simultaneously.**

- **FORTRAN does not allow recursion, so activation records can be kept in static storage.**
- **TINY has no functions or procedures, so no activation records, and so needs only static storage too.**
- **Some languages allow local data within a call to be accessed after the call has finished (Scheme). Then activation record must go on the heap. (Well, C does too, but that is defined to be an error.)**
- **An activation record on the stack is called a *stack frame.***

Schematic of an activation record on the stack:



What kind of book-keeping information? Pointer to previous activation so that whole activation can be “popped” at once.

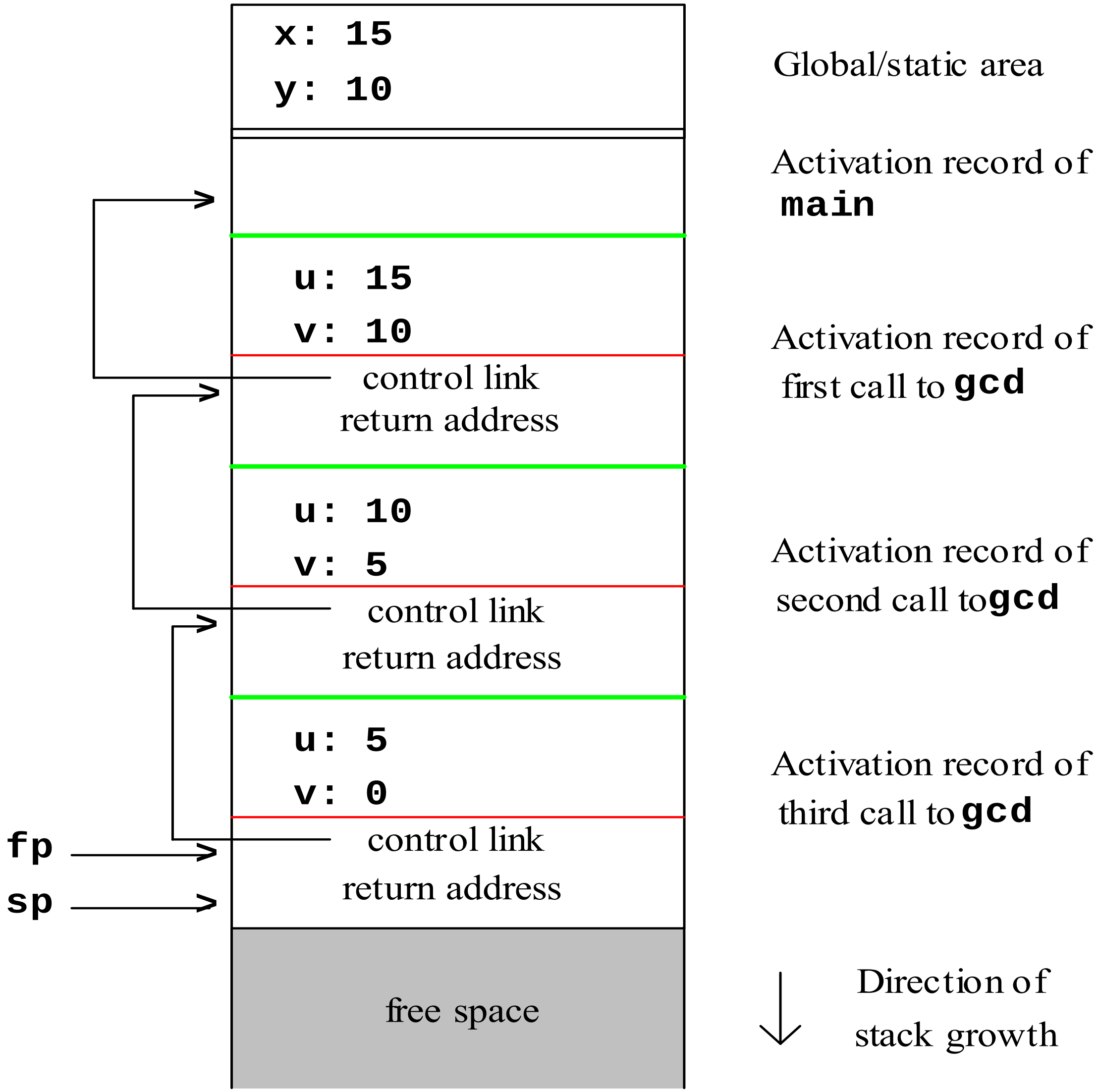
Example in C:

```
int x,y;

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

Call structure on input 15, 10:



- **Sp is the "stack pointer", typically a register managed by the processor.**
- **Fp is the "frame pointer", which may or may not be a register (if it isn't a register, it is usually not there).**
- **Arguments are computed by caller and pushed onto the stack; thus, they may be viewed as belonging to the previous frame (red lines above).**
- **However, from the point of view of scope, they are part of the callee (green lines above).**

Calling Sequence: standard code inserted at call sites, and on entry and exit from functions:

- **Caller responsibility:**
  - **Before call, compute arguments and push in order onto the stack**
  - **Store return address at call**
  - **On exit, pop remaining old frame info from stack (like args, return value)**
- **Callee responsibility:**
  - **Set up and remove current frame**
  - **Record return value**

Example in C (gcc):

```
_gcd:
  pushl %ebp
  movl %esp, %ebp
  subl $4, %esp
  cmpl $0, 12(%ebp)
  jne L2
  movl 8(%ebp), %eax
  jmp L1
  .p2align 4,,7
L2:
```

```
  subl $8, %esp
  movl 8(%ebp), %edx
  movl %edx, %eax
  sarl $31, %edx
  idivl 12(%ebp)
  pushl %edx
  pushl 12(%ebp)
  call _gcd
  addl $16, %esp
L1:
  movl %ebp, %esp
  popl %ebp
  ret
```

fp

Entry sequence

?

?

Compute arguments (reverse order)

Adjust stack

Exit sequence

Where did the return address go?

- **Handled automatically by the processor**
- **Call pushes retaddr onto stack**
- **Ret assumes retaddr is on top of stack, pops it to the program counter**
- **Thus, in such an architecture, retaddr goes *above* control link in stack frame**
- **Remember, too, in C: args go onto stack in *reverse order*.**
- **Return value typically goes into a register (eax in the case of the PC)**

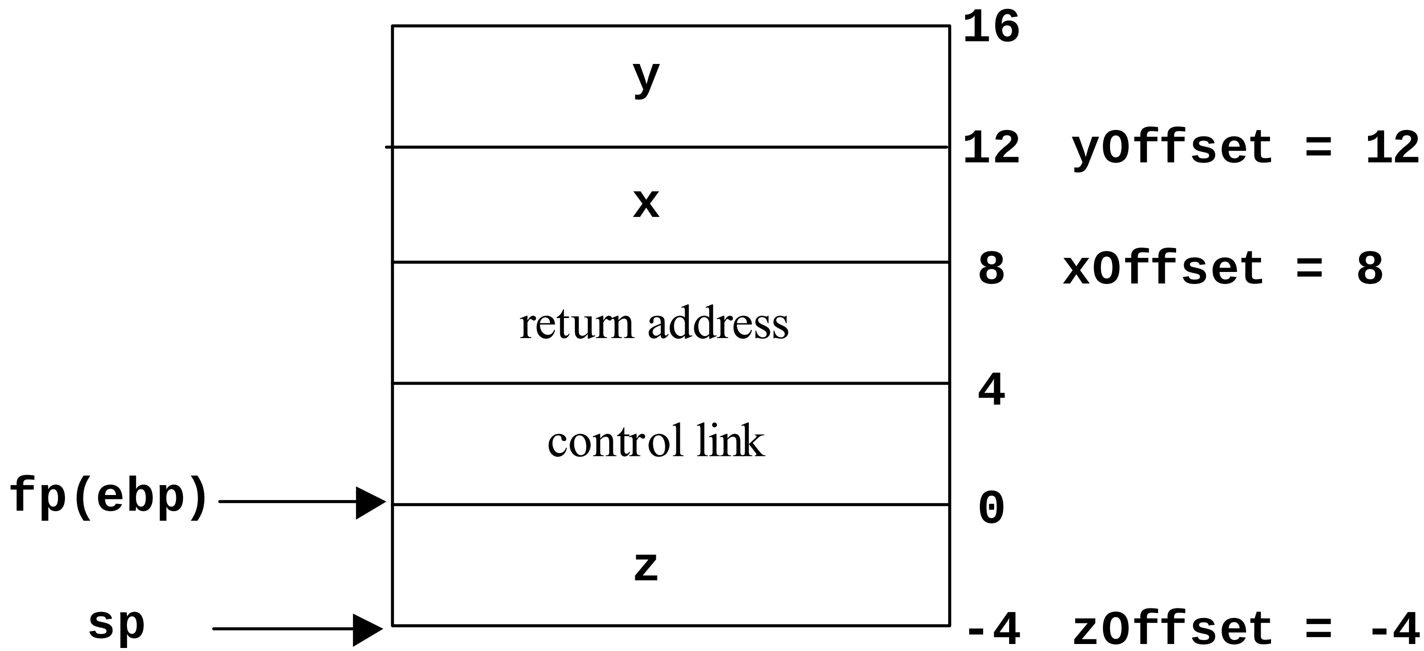
Local variable and parameter access:

- **All locals are accessed by *fixed offset* from the frame pointer.**
- **Offset is computable at compile time (keep a running total), but frame pointer is not.**
- **In a typical processor-managed stack, parameters have positive offset, local variables have negative offset.**

Example in C:

```
int f( int x, int y)
{ int z = 42; return x + y + z;}
-------------
_f:    pushl %ebp
       movl  %esp, %ebp
       subl  $4, %esp // reserve space for z
       movl  $42, -4(%ebp) // initialize z
       movl  12(%ebp), %eax // move y to reg eax
       addl  8(%ebp), %eax // add x to it
       addl  -4(%ebp), %eax // add z to it
       movl  %ebp, %esp
       popl  %ebp
       ret
```

Picture (32-bit architecture):



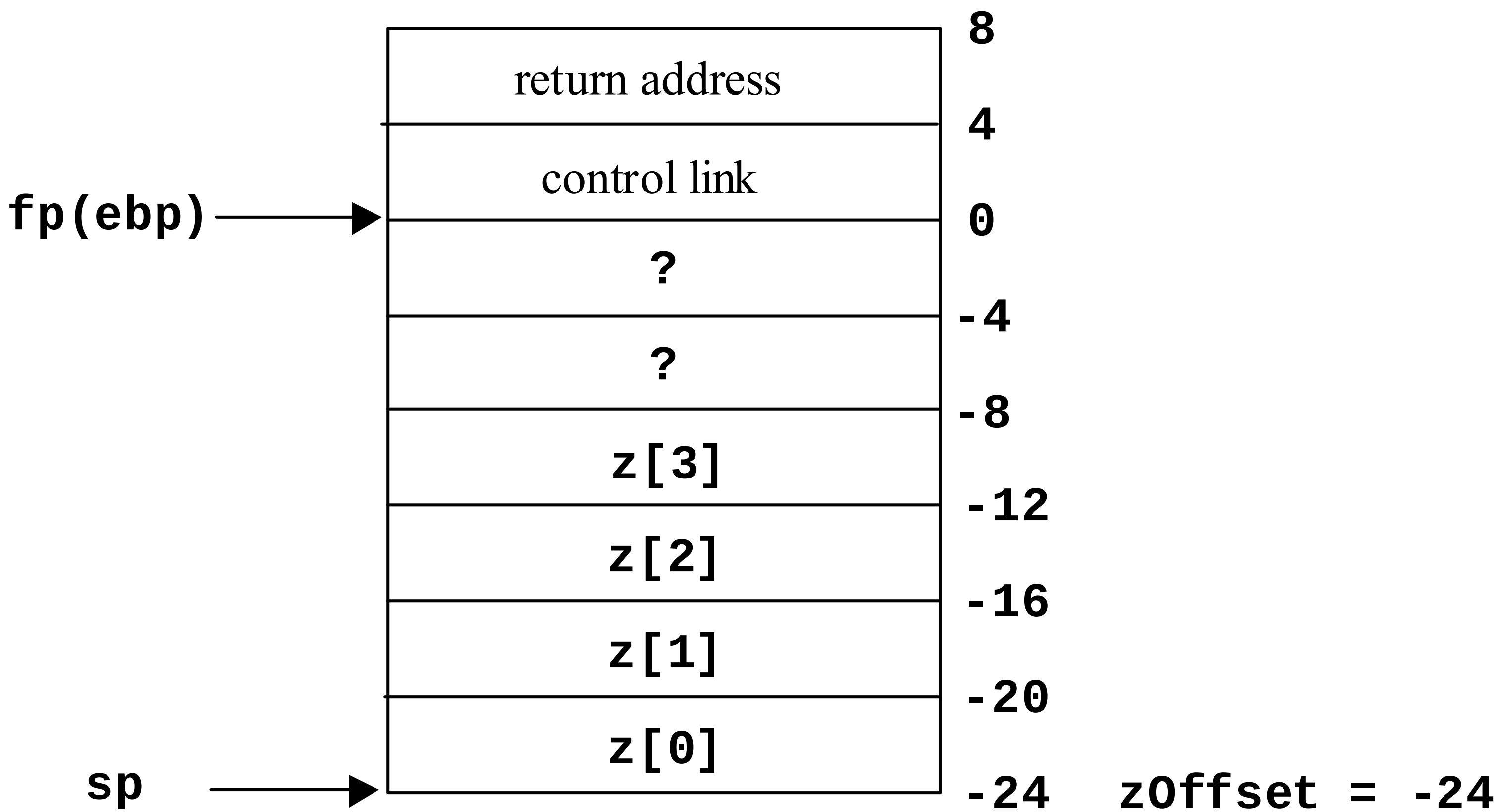
Non-local references - always global in C:

```
int x; // non-local
int f(void)
{ int z = 42; return x + z;}
------------
.globl _f
_f: . . .
  movl      $42, -4(%ebp)
  movl      -4(%ebp), %eax // load z
  addl      _x, %eax
  . . .
  ret
.comm _x,16 // allocate 16 bytes of "common"
            // storage, give it the name _x
```

Arrays - allocate in stack (size must be fixed in C):

```
int f(void)
{ int z[4]; return z[0] + z[2];}
-------------
_f:
  pushl     %ebp
  movl      %esp, %ebp
  subl      $24, %esp
  movl      -16(%ebp), %eax // load z[2]
  addl      -24(%ebp), %eax // add z[0]
  movl      %ebp, %esp
  popl      %ebp
  ret
```

Array picture:



Computed array subscripts:

```
int f(int i)
{ int z[4]; return z[i];}
-------------
_f:    pushl %ebp
       movl  %esp, %ebp
       subl  $24, %esp
       movl  8(%ebp), %eax // move i to eax
       leal  0(,%eax,4), %edx // mult by 4
       leal  -24(%ebp), %eax // load z addr
       movl  (%edx,%eax), %eax // *(z+4*i) -> eax
       movl  %ebp, %esp
       popl  %ebp
       ret
```

Nested scopes:

```
void f(void)
{ int x;
  { int y = 2;
    { int z = 3;
      x = y + z;
    }
  }
  { int w = 4;
    int v = 5;
    x = v + w;
  }
}
```

```
_f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $12, %esp
    movl    $2, -8(%ebp)
    movl    $3, -12(%ebp)
    movl    -12(%ebp), %eax
    addl    -8(%ebp), %eax
    movl    %eax, -4(%ebp)
    movl    $4, -12(%ebp)
    movl    $5, -8(%ebp)
    movl    -12(%ebp), %eax
    addl    -8(%ebp), %eax
    movl    %eax, -4(%ebp)
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Functions as parameters - they are just pointers:

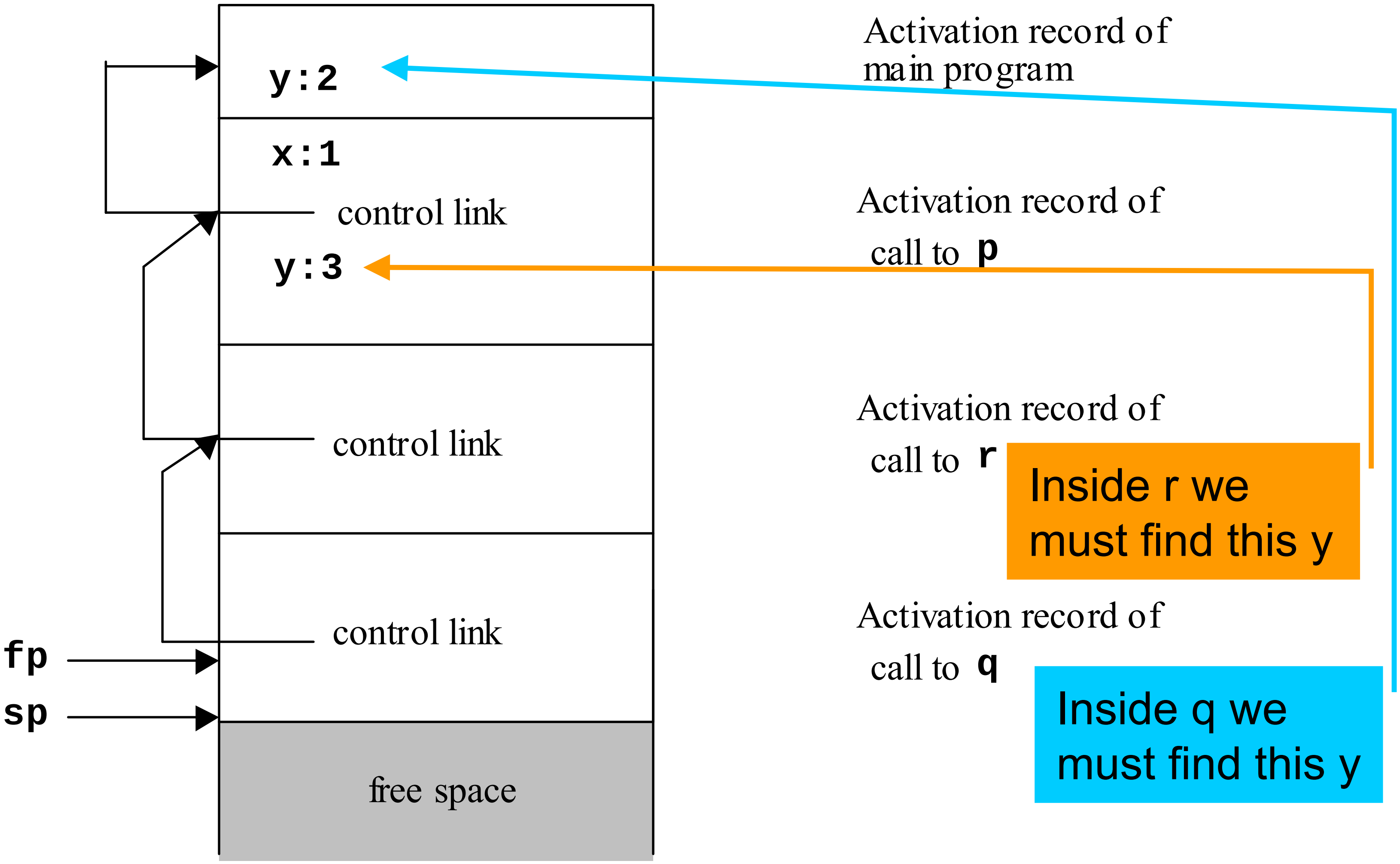
```
int f(int (*g)(int),int x)
{ return g(x); }
------------
_f:    pushl %ebp
       movl  %esp, %ebp
       subl  $8, %esp // ??
       subl  $12, %esp // ??
       pushl 12(%ebp) // push x
       call  *8(%ebp) // call g
       addl  $16, %esp
       movl  %ebp, %esp
       popl  %ebp
       ret
```

```
int g(int y)
{ return f(g,y); }
------------
_g:
  pushl     %ebp
  movl      %esp, %ebp
  subl      $8, %esp
  subl      $8, %esp
  pushl     8(%ebp) // push y
  pushl     $_g // push global g
  call      _f
  addl      $16, %esp
  movl      %ebp, %esp
  popl      %ebp
  ret
```

Nested functions - a major complication! (Pascal, Ada, Scheme)

```
procedure main is -- Ada example
  y : integer := 2;
  procedure p(x: integer) is
    function q return integer is
        begin return x + y; end;
    y: integer := 3; -- nonlocal/nonglobal to r!
    function r return integer is
        begin return q + y; end;
  begin -- p
    put(r);
  end;
begin
  p(1); -- prints 6
end;
```

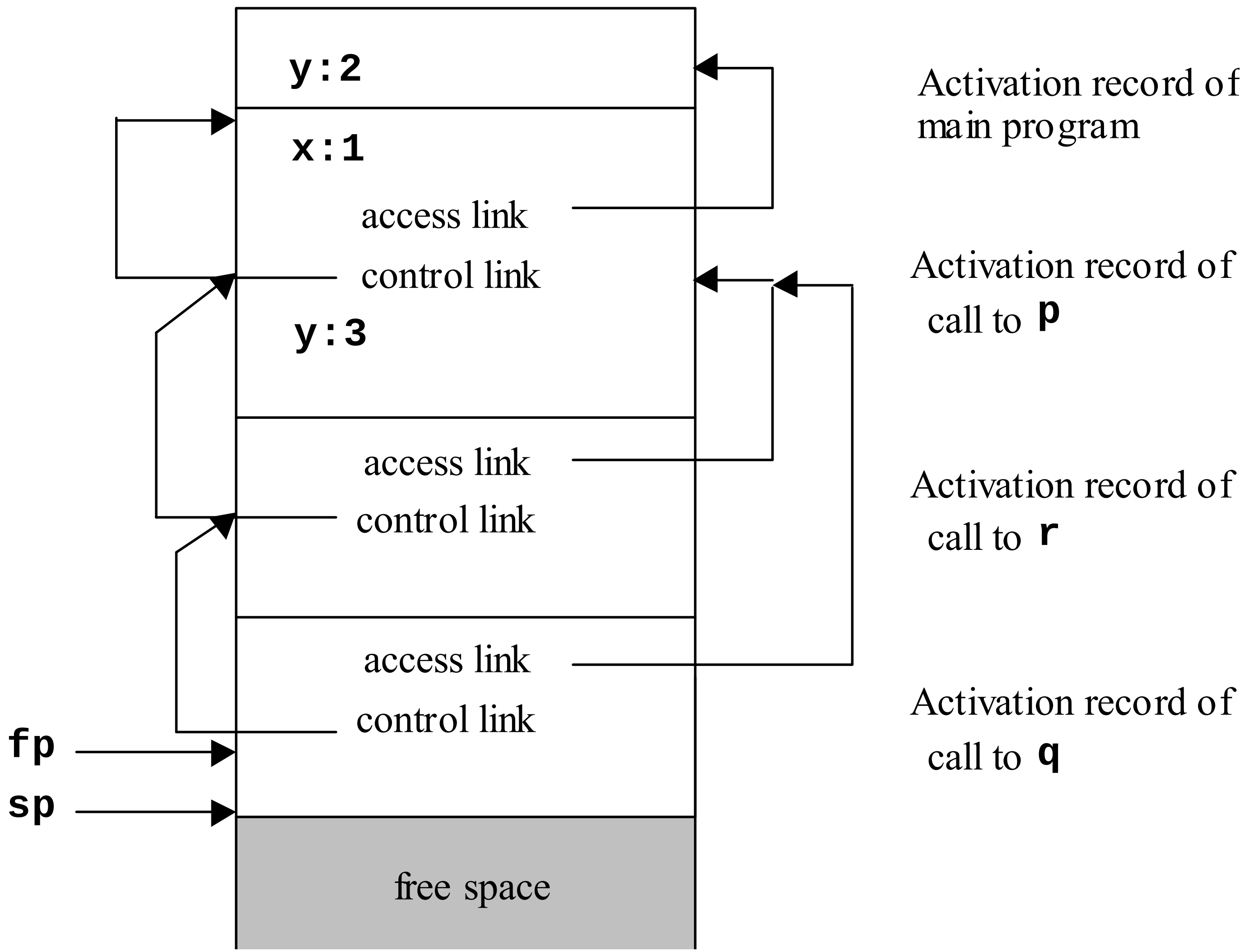
Stack during call to q:



New piece of bookkeeping info - the access link:

- **Each activation record must keep a record of the location of the activation record in which its corresponding function was defined.**
- **This can be computed and stored at the time of call, but not before.**
- **It is typically called the *access link.***

Stack with access links:



Use access link for nonlocal references:

- **Inside r, follow one access link to find y.**
- **Inside q, follow one access link to find x, *two* access links to find y.**
- **Number of access links to follow = difference in nesting levels.**
- **Following multiple access links is called *access chaining.***

Nested functions as parameters:

- **When a function is passed to another function, its access link must be passed as well as its code (instruction) pointer.**
- **Functions become *pairs* of pointers, called the ip (*instruction pointer*) and the ep (*environment pointer*): <ip,ep>. This is called a *closure*.**
- **ip can be computed at compile time, but ep cannot.**

Parameters: values or references?

- **In C, all parameters are *value parameters*: arguments are copied values, which become initial values of the parameters during each call. This means that simple variables in the calling environment cannot be changed by the callee.**
- **Well, not quite: arrays are *implicitly pointers* or references, so the values stored in an array *can* be changed by a callee.**
- **Sometimes we say arrays are passed in C *by reference.***

- **In some languages (Pascal, Ada), you can specify explicitly that you want a reference parameter instead of a value parameter: `p(var x:integer)` (Pascal).**
- **Any such reference parameter needs an extra level of indirection to fetch its value.**
- **In some languages (Scheme, Java), certain parameters (lists in Scheme, objects in Java) are, like C arrays, *implicitly* references. So a similar indirection is necessary, and values can be changed by a callee.**