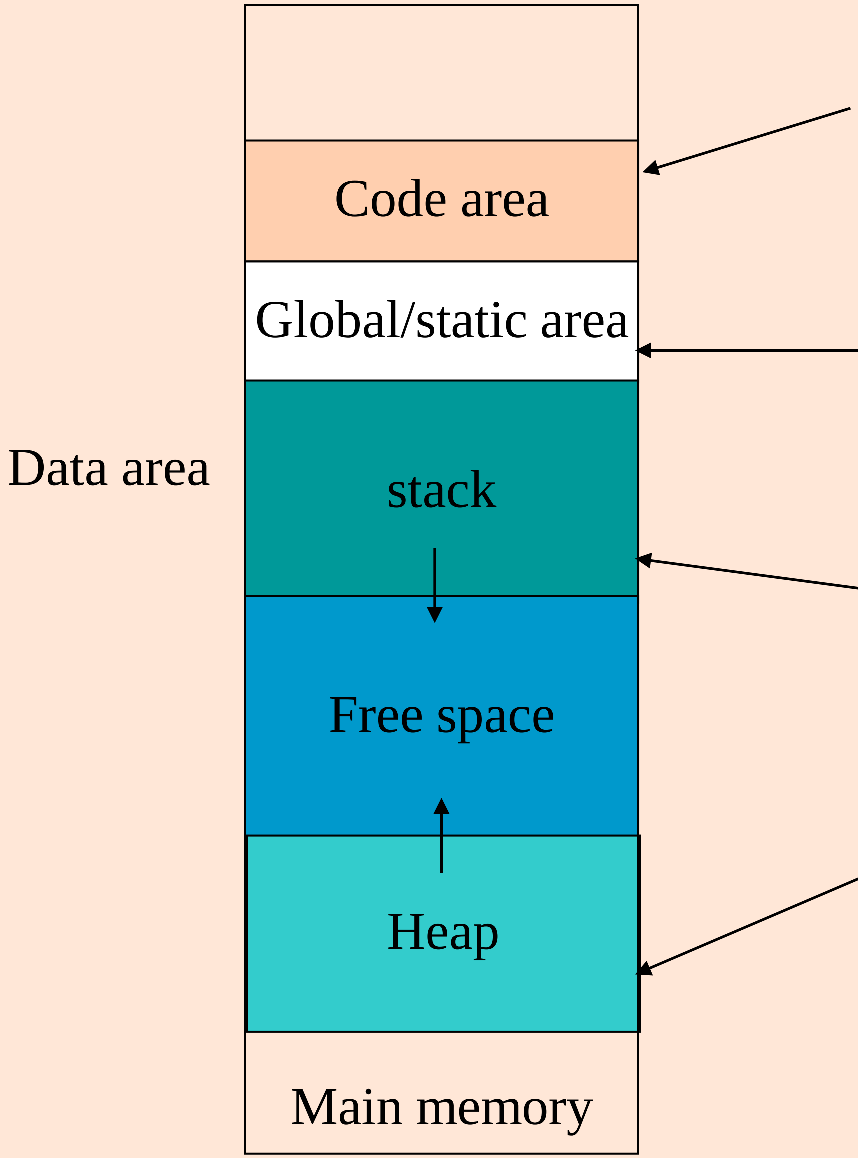
Run Time Environment
OR
Memory Organization in Compiler

MEMORY ORGANIZATION DURING PROGRAM EXECUTION



Memory locations for code are determined at compile time.

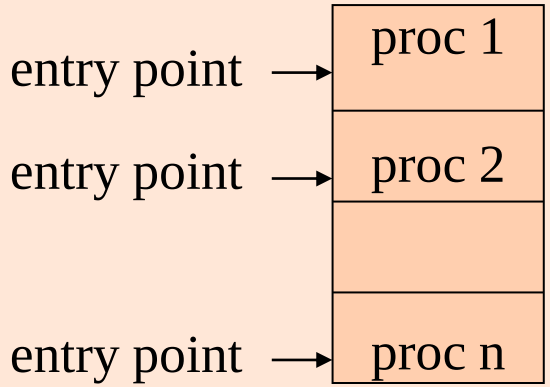
Locations of static data can also be determined at compile time

Data objects allocated at run-time. (Activation Records)

Other dynamically allocated data objects at run-time. (For example, malloc area in C).

Code Area

- Addresses in code area are static (i.e. no change during execution) for most programming language.
- Addresses are known at compile time.



3

DATA AREA

- Addresses in data area are static for some data and dynamic for others.
  - Static data are located in static area.
  - Dynamic data are located in stack or heap.
    - Stack (LIFO allocation) for procedure activation record, etc.
    - Heap for user allocated memory, etc.

RUNTIME ENVIRONMENTS

Three types

- Fully Static
  - Fortran77
- Stack-based
  - C, C++, Pascal, Ada
- Fully Dynamic
  - LISP

Static runtime environments

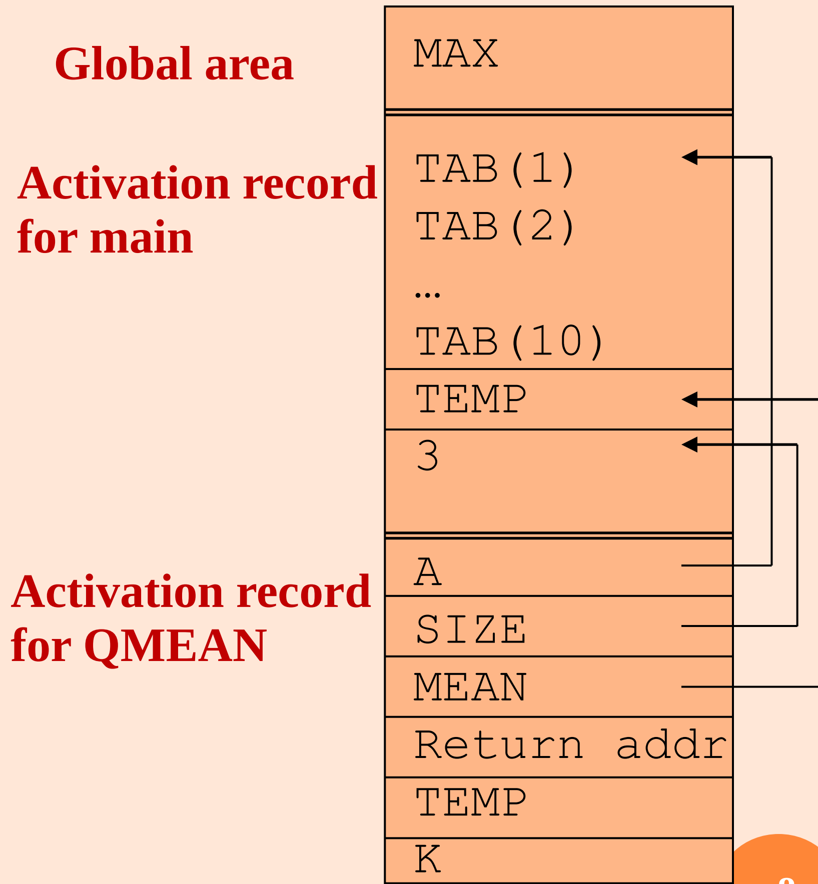
- Static data
  - Both local and global variables are allocated once at the beginning and deallocated at program termination
  - Fixed address
- No dynamic allocation
- No recursive call
  - Procedure calls are allowed, but no recursion.
  - One activation record for each procedure, allocated statically
- Example:FORTRAN 77

Static Code and Data

- The code area of a program is fixed prior to execution
- Thus the address for code can be computed at compile time
  - Actually, the location of the code is usually relative to some base register, or some other memory addressing scheme
- The address for data (variables etc.) is in general not assigned fixed locations at compile time
- Static data however, does have a fixed memory address and is known at compile time
- Constants can also be assigned a fixed address
  - This is usually reserved for strings and other large constants
  - Other constants like ‘0’ or ‘1’ are inserted directly into the code

Memory Organization for Static Runtime Environment

```
PROGRAM TEST
COMMON MAX
INTEGER MAX
REAL TAB(10), TEMP
…
QMEAN(TAB, 3, TEMP)
…
END

SUBROUTINE QMEAN(A, SIZE, MEAN)
COMMON MAX
INTEGER MAX, SIZE
REAL A(SIZE), MEAN, TEMP
INTEGER K
…
END
```



8

DYNAMIC DATA ALLOCATION

- The memory for dynamic data is typically organized into two major areas
  - Stack
    - Used for local variables, parameter variables, return addresses and return values
  - Heap
    - Used for dynamically allocated variables

STACK-BASED RUNTIME ENVIRONMENTS

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

HEAP-BASED RUNTIME ENVIRONMENTS

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

PROCEDURE ACTIVATIONS

- An execution of a procedure starts at the beginning of the procedure body;
- When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
- Each execution of a procedure is called as its ***activation.***
- ***Lifetime*** of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
- If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested.
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.

ACTIVATION TREE

- We can use a tree (called **activation tree**) to show the way control enters and leaves activations.

- In an activation tree:
  - Each node represents an activation of a procedure.
  - The root represents the activation of the main program.
  - The node a is a parent of the node b iff the control flows from a to b.
  - The node a is left to the node b iff the lifetime of a occurs before the lifetime of b.

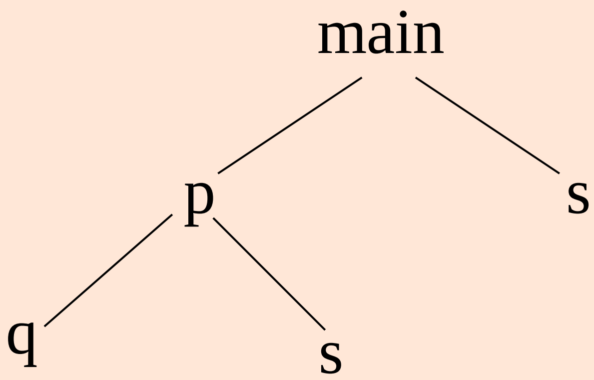
Activation Tree (cont.)

```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```

enter main
enter p
enter q
exit q
enter s
exit s
exit p
enter s
exit s
exit main

A Nested Structure

ACTIVATION TREE (CONT.)



enter main
enter p
enter q
exit q
enter s
exit s
exit p
enter s
exit s
exit main

A Nested Structure

CONTROL STACK

- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - starts at the root,
  - visits a node before its children, and
  - recursively visits children at each node in a left-to-right order.
- A stack (called **control stack**) can be used to keep track of live procedure activations.
  - An activation record is pushed onto the control stack as the activation starts.
  - That activation record is popped when that activation ends.
- When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

VARIABLE SCOPES

- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declaration of a name applies when the name appears in the program.
- An occurrence of a variable (a name) is:
  - **local**: If that occurrence is in the same procedure in which that name is declared.
  - **non-local**: Otherwise (*i.e.* it is declared outside of that procedure)

```
procedure p;
  var b:real;
  procedure p;
    var a: integer;
    begin a := 1;  b := 2; end;
  begin ... end;
```

a is local

b is non-local

ACTIVATION RECORDS

- Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.

- An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exits.

- Size of each field can be determined at compile time (Although actual location of the activation record is determined at run-time).
  - Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.

ACTIVATION RECORDS (CONT.)

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.

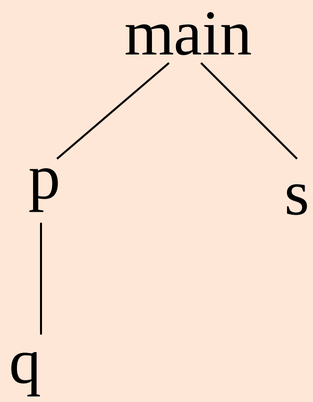
The field for saved machine status holds information about the state of the machine before the procedure is called.

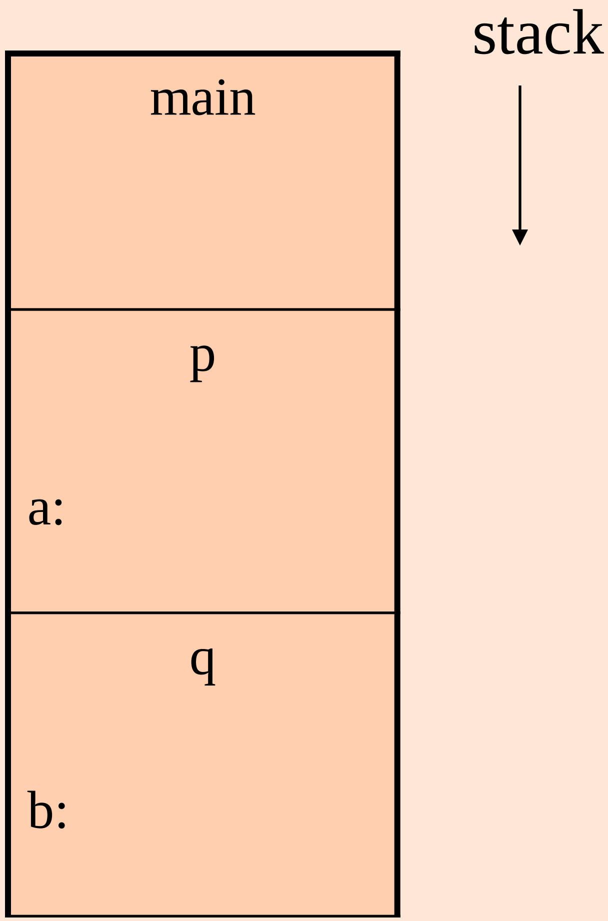
The field of local data holds data that local to an execution of a procedure.

Temporay variables is stored in the field of temporaries.

ACTIVATION RECORDS (EX. 1)

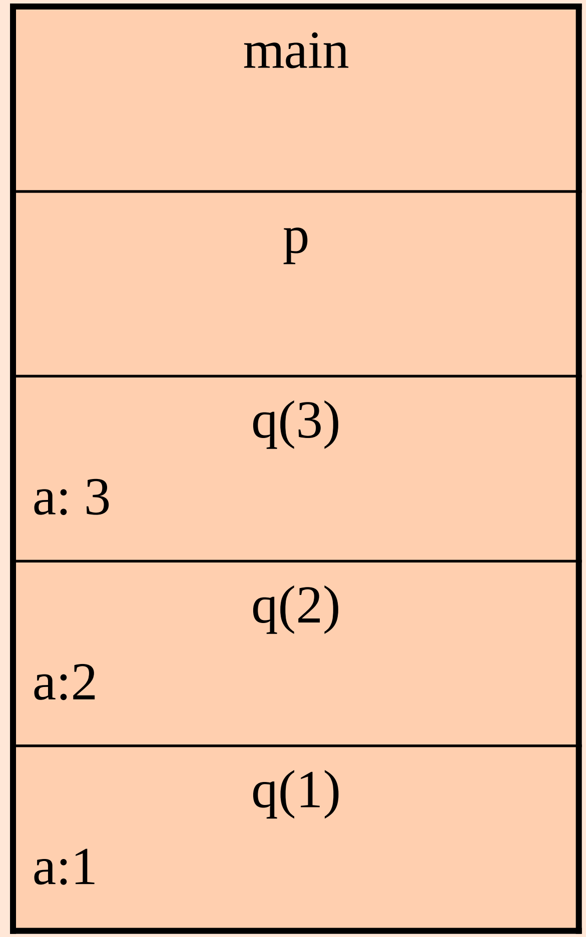
```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```



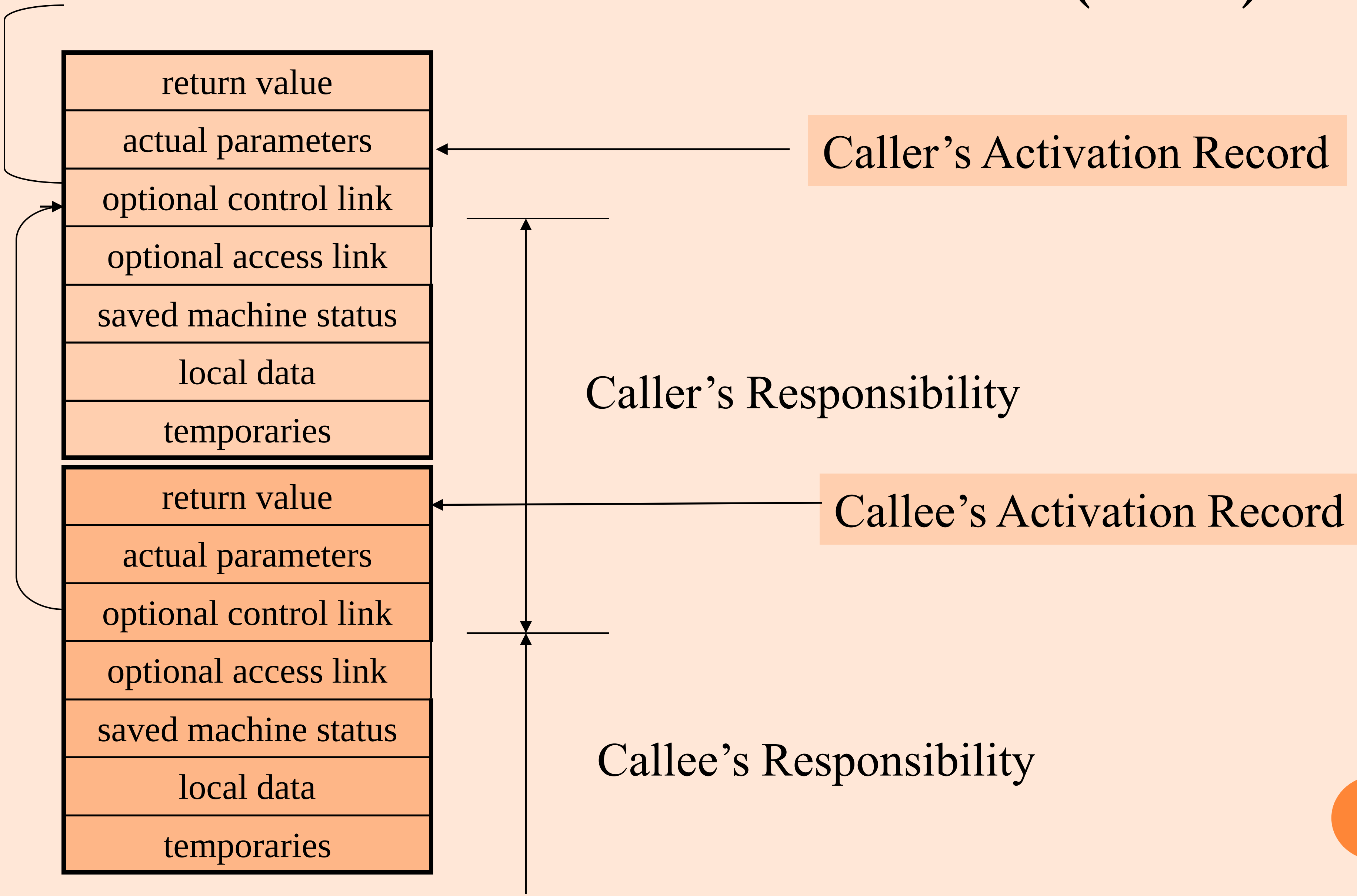


Activation Records for Recursive Procedures

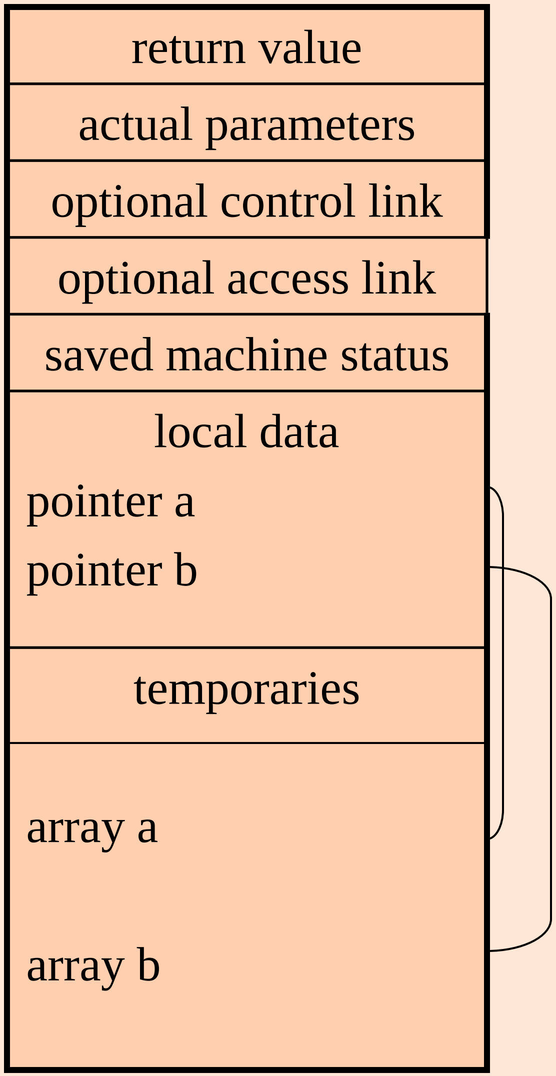
```
program main;
  procedure p;
    function q(a:integer):integer;
      begin
        if (a=1) then q:=1;
        else q:=a+q(a-1);
      end;
    begin q(3); end;
  begin p; end;
```



Creation of An Activation Record (cont.)



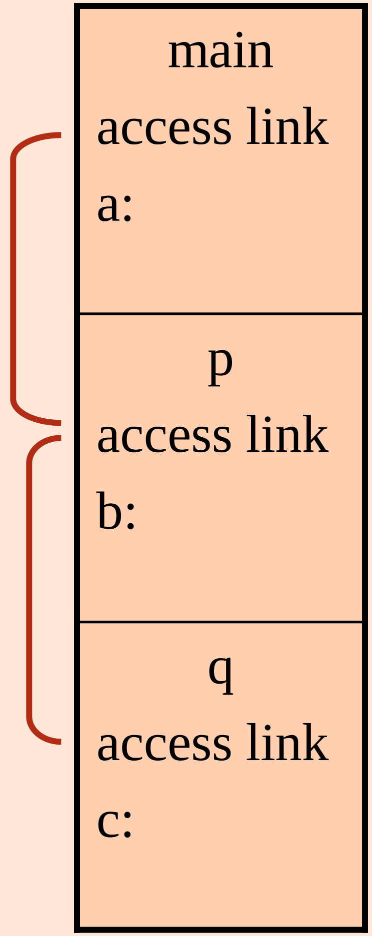
Variable Length Data



Variable length data is allocated after temporaries, and there is a link from local data to that array.

Accessing Nonlocal Variables

```
program main;
  var a:int;
  procedure p;
    var b:int;
    begin q; end;
  procedure q();
    var c:int;
    begin
      c:=a+b;
    end;
  begin p; end;
```




```
addrC := offsetC(currAR)
t := *currAR
addrB := offsetB(t)
t := *t
addrA := offsetA(t)
ADD addrA,addrB,addrC
```

ACCESS TO NONLOCAL NAMES

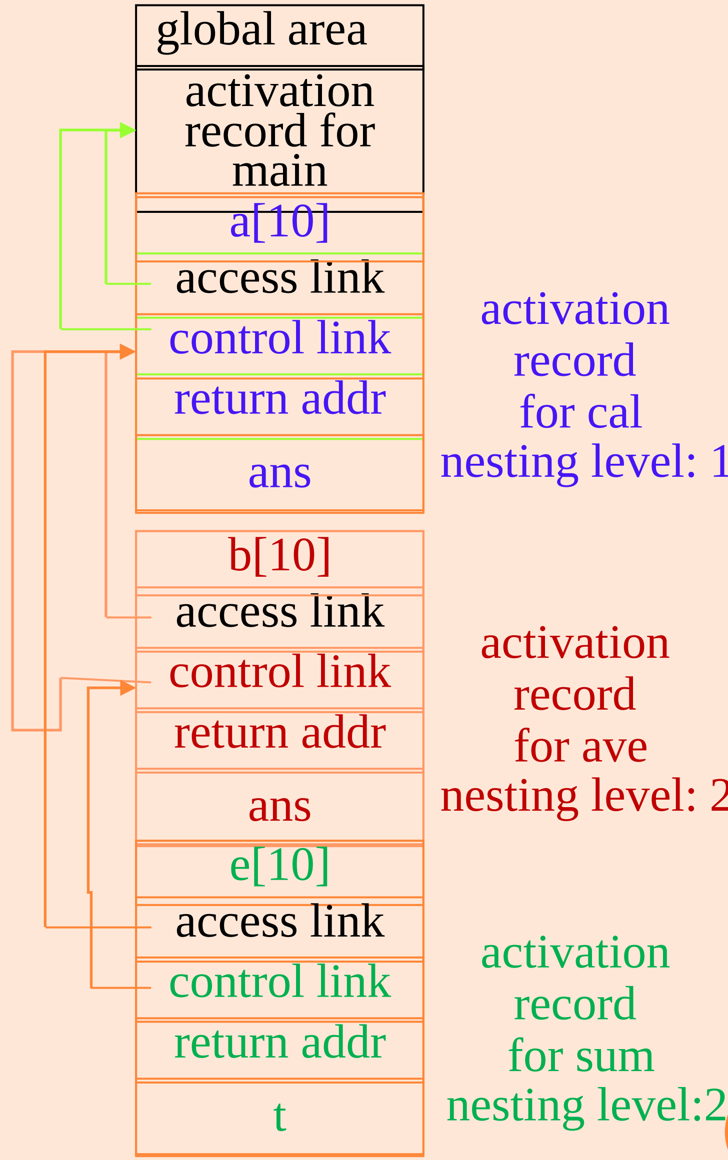
- Scope rules of a language determine the treatment of references to nonlocal names.
- Scope Rules:
  - **Lexical Scope (Static Scope)**
    - Determines the declaration that applies to a name by examining the program text alone at compile-time.
    - Most-closely nested rule is used.
    - Pascal, C, ..
  - **Dynamic Scope**
    - Determines the declaration that applies to a name at run-time.
    - Lisp, APL, ...

ACCESSING VARIABLES IN LOCAL PROCEDURES

- Access Chaining
  - Follow access links from one activation record until the required variable is found
  - Inefficient
- Display
  - Access links are stored in an array called *display*

Accessing Chaining

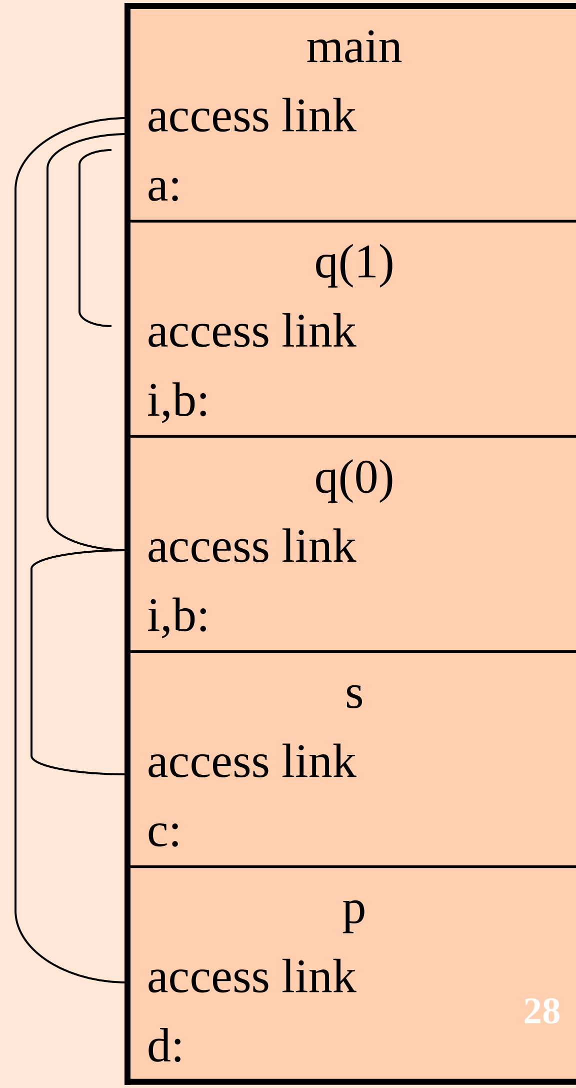
- Nesting level
  - Indicates the depth of the procedure in program definition
    - Level 0: outermost
  - Need to be stored in the symbol table
- To find the activation record for the n^{th} nesting level from the activation record in the i^{th} nesting level
  - Follow the access link (n-i) times
- Examples:
  - To access `ans` in `cal`
    - Follow links 2-1 times
  - To access `n` in `main`
    - Follow links 2-0 times



Access Chaining

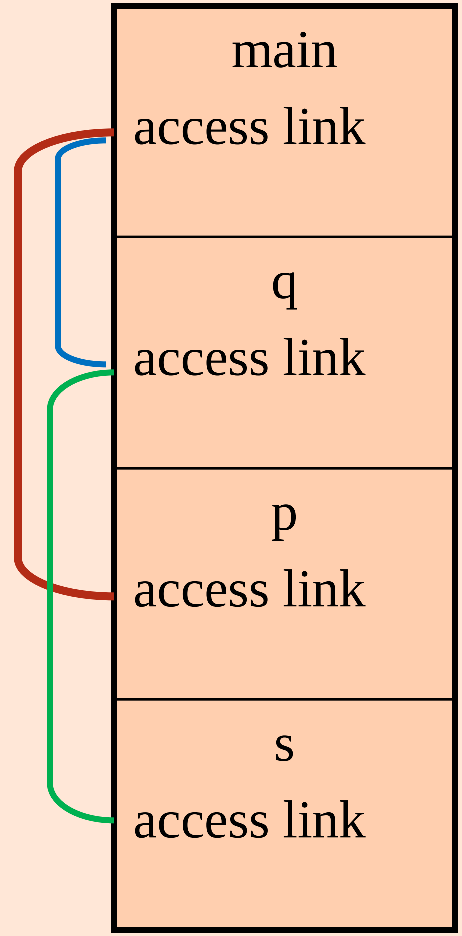
```
program main;
  var a:int;
  procedure p;
    var d:int;
    begin a:=1; end;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      begin p; end;
    begin
      if (i<>0) then q(i-1)
      else s;
    end;
  begin q(1); end;
```

Access Links



Procedure Parameters

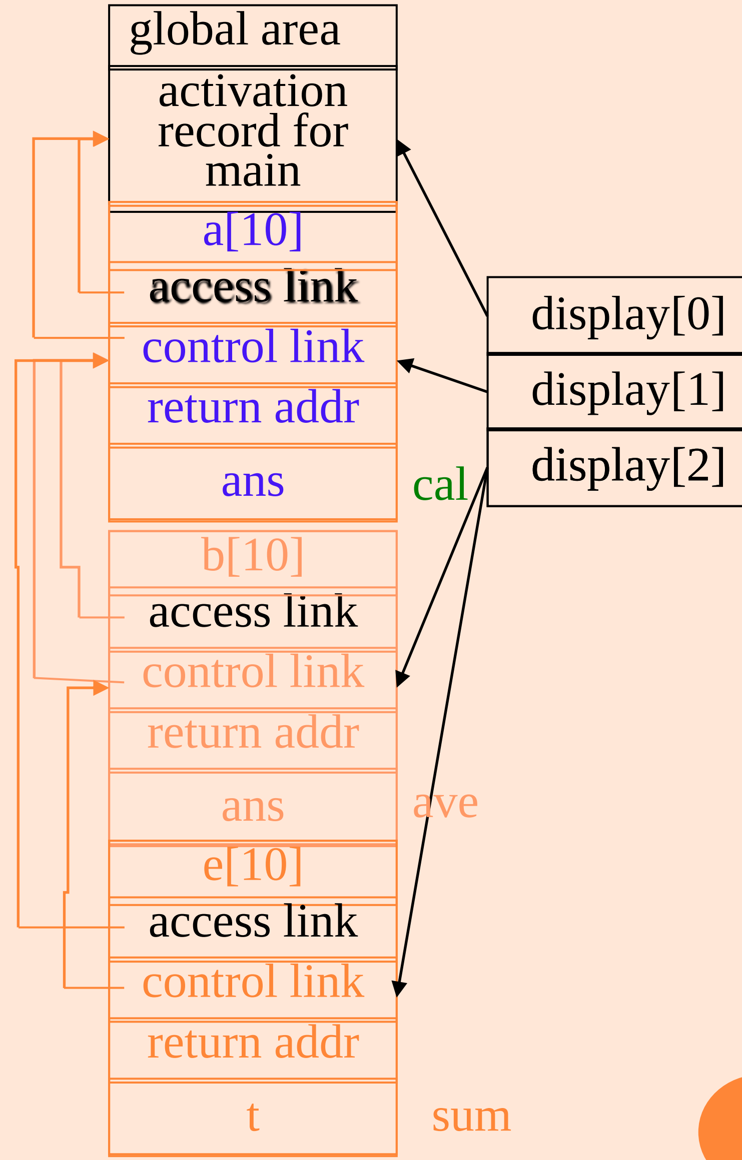
```
program main;
  procedure p(procedure a);
    begin a; end;
  procedure q;
    procedure s;
      begin ... end;
    begin p(s) end;
  begin q; end;
```



Access links must be passed with procedure parameters.

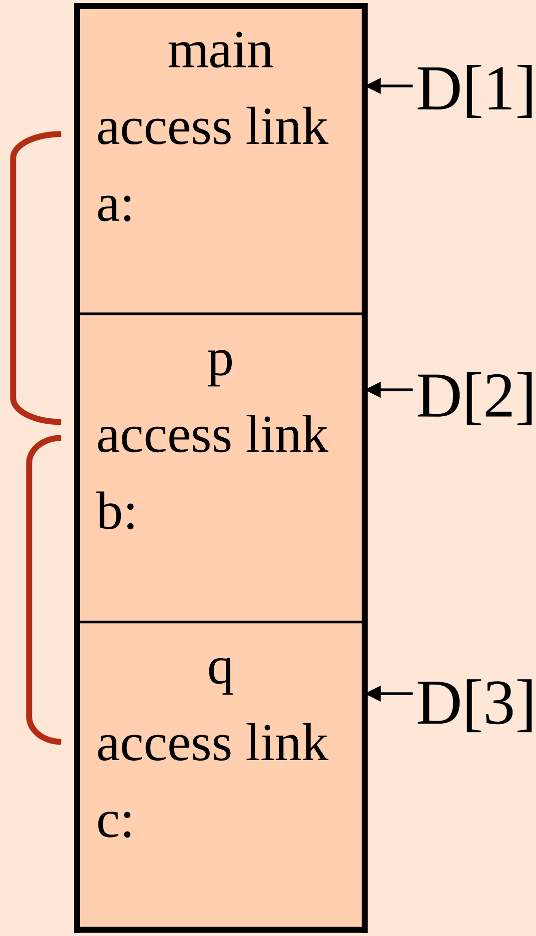
DISPLAY

- Access links are stored in the array display.
- The activation record of a procedure in the i^{th} nesting level is stored in display[i].



Accessing Nonlocal Variables using Display

```
program main;
  var a:int;
  procedure p;
    var b:int;
    begin q; end;
  procedure q();
    var c:int;
    begin
      c:=a+b;
    end;
  begin p; end;
```



addrC := offsetC(D[3])
addrB := offsetB(D[2])
addrA := offsetA(D[1])
ADD addrA,addrB,addrC

PARAMETER PASSING

- Pass by value
- Pass by reference
- Pass by value-result

PASS BY VALUE

- Value parameters are not changed during the execution
- Only the value is sent into the procedure, and are used locally
- When the control is returned from the callee, the value of the parameter is not passed back to the caller.

```
void  change(int x)
{ x++;
  return;
}

void  main()
{ int y=0;
  change(y);
  printf("%d\n",y);
  return;
}
```

Output:

0

Pass by Reference

- The reference to a variable is passed to the callee.
- The callee uses the reference (address) to refer to the variable.
  - Indirect addressing is needed to refer to parameters
- The variable in the caller and the referenced memory in the callee share the same memory location.
- The value of the variable in the caller is also changed when the referenced in the callee is changed.

```
void  change (int &x)
{ x++;
  return;
}

void main()
{ int  y=0;
  change(y);
  printf("%d\n",y);
  return;
}
```

Output:

1

Pass by Value-Result

- The value of the parameter is copied into the callee when the callee is entered.
- New memory location is provided for the parameter in the callee's activation record.
  - No indirect address is needed
- When the control is returned to the caller, the value is copied back.

```
void  change(int x)
{ x++;
  return;
}

void  main()
{ int y=0;
  change(y);
  printf("%d\n",y);
  return;
}
```

Output:
1

Examples of Activation Record generation and termination

GCD

```
int x,y;

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

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

Activation Records

x: 15
y: 10

global static area

```
int x,y;

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

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

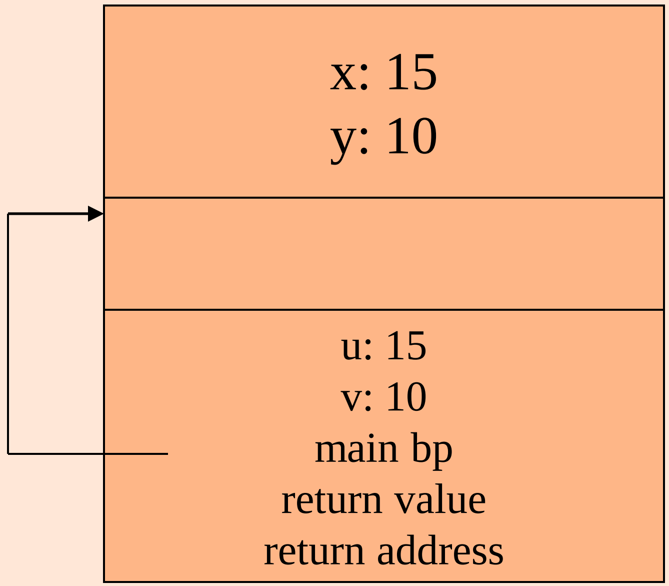
ACTIVATION RECORDS

x: 15 y: 10	global static area
	activation record of main

```
int x,y;

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

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

Activation Records

```
int x,y;

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

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



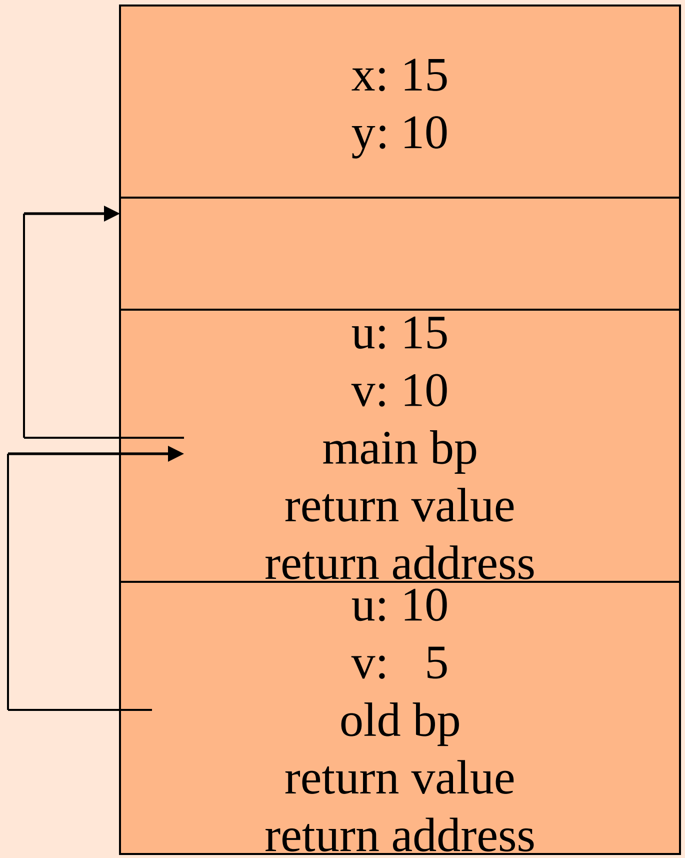
global static area

activation record of main

first call to gcd

ACTIVATION RECORDS

```
int x,y;

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

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



global static area

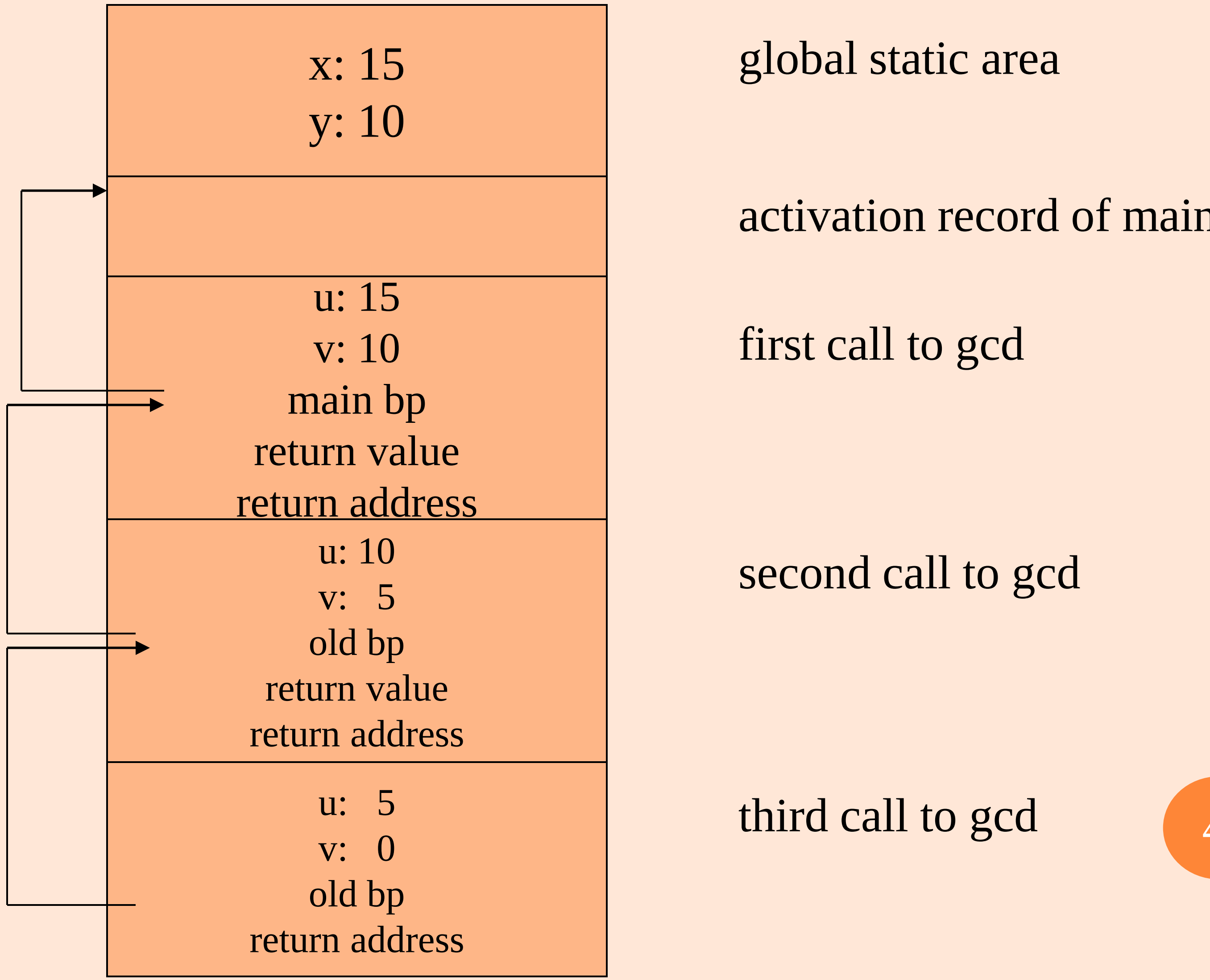
activation record of main

first call to gcd

second call to gcd

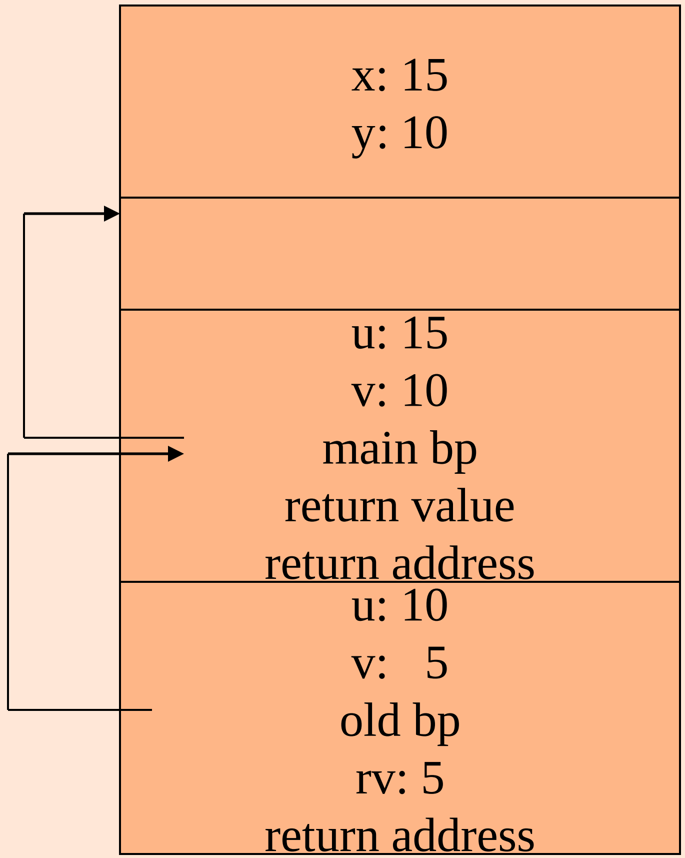
Activation Records

```
int x,y;

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

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



Activation Records

```
int x,y;

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

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



global static area

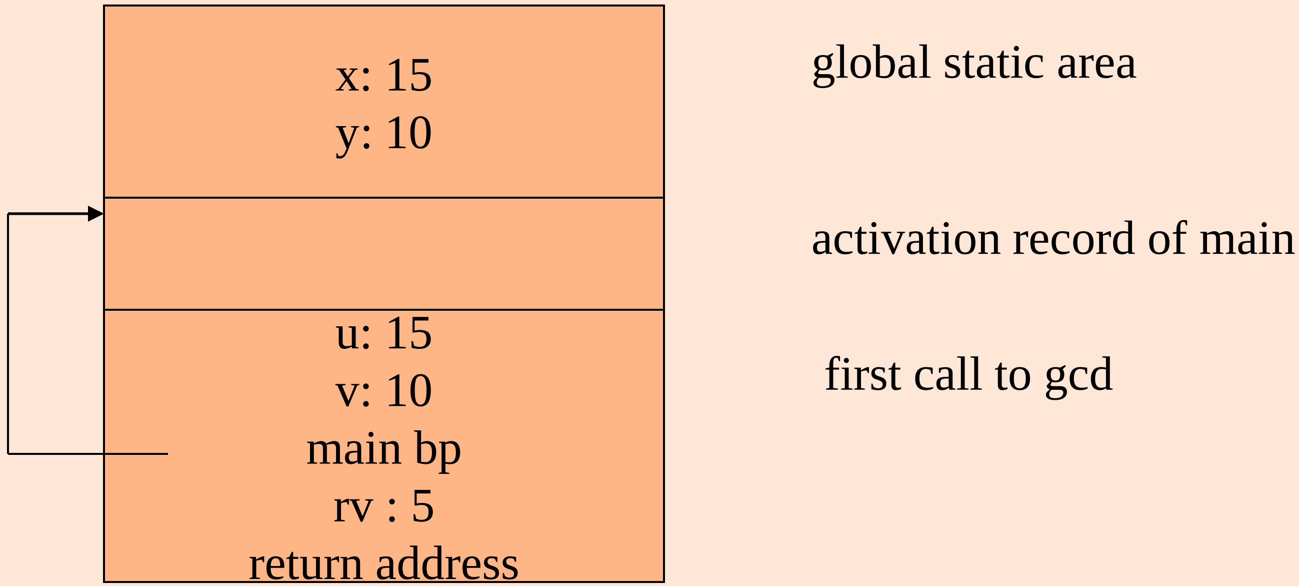
activation record of main

first call to gcd

second call to gcd

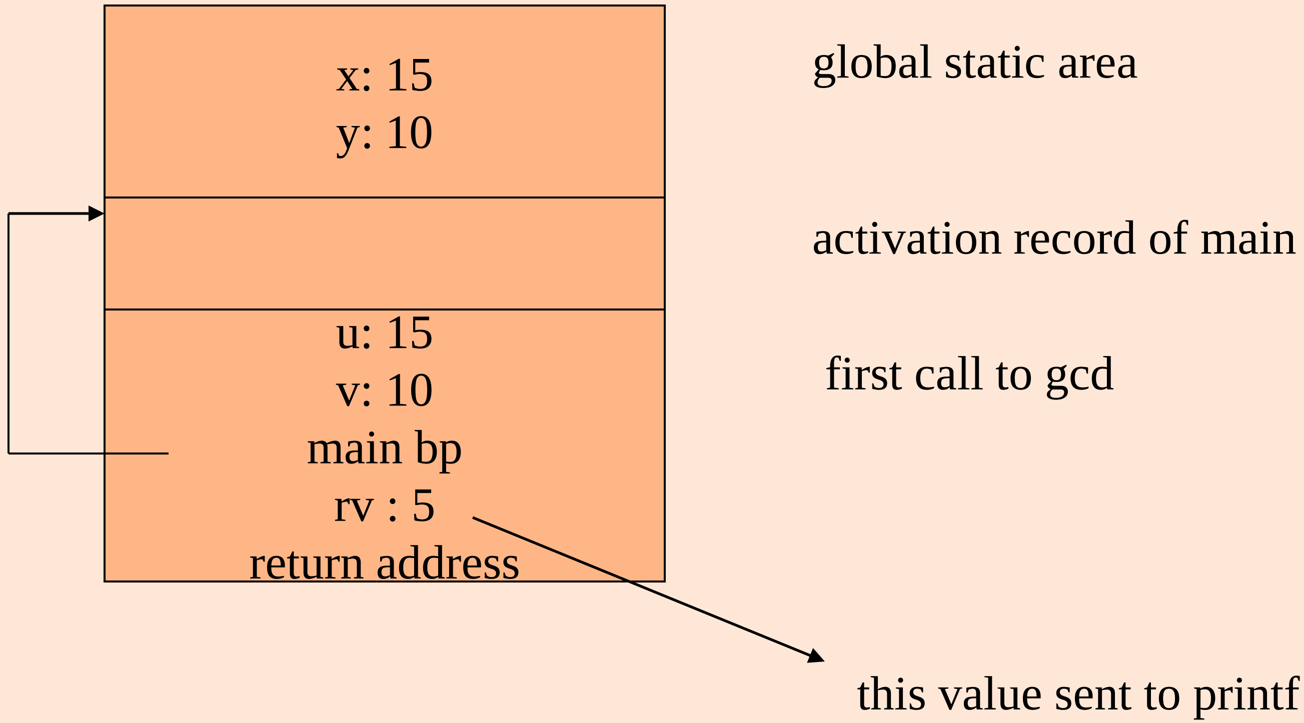
ACTIVATION RECORDS

```
int x,y;

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

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



Activation Records

```
int x,y;

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

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

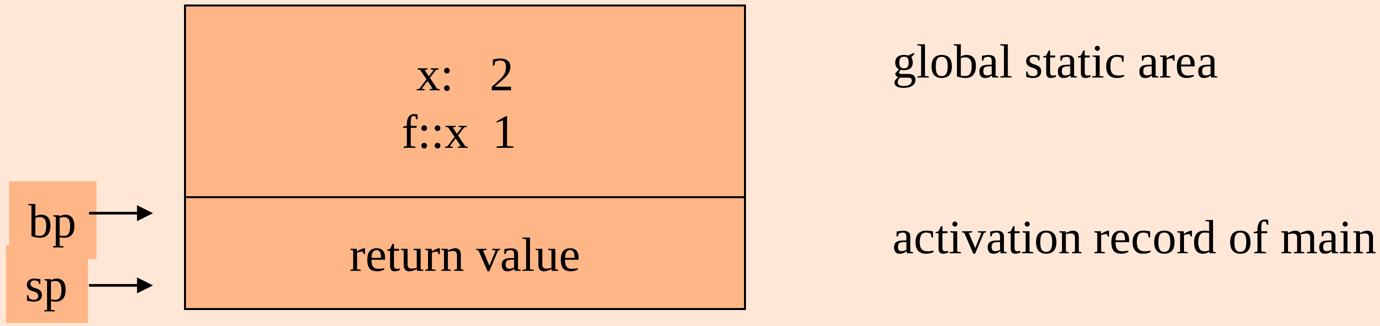


Sample code

```
int x = 2;

void f(int n)
{  static int x= 1;
   g(n);
   x--;
}
void g(int m)
{  int y=m-1;
   if (y>0)
     {  f(y);   x--; g(y); }
}
int main ()
{
   g(x)
   return 0;
}
```

Activation Records

```
int x = 2;

void f(int n)
{  static int x= 1;
   g(n);
   x--;
}
void g(int m)
{  int y=m-1;
   if (y>0)
     {  f(y);   x--; g(y); }
}
int main ()
{
   g(x)
   return 0;
}
```



ACTIVATION RECORDS

```
int x = 2;

void f(int n)
{  static int x= 1;
   g(n);
   x--;
}
void g(int m)
{  int y=m-1;
   if (y>0)
     {  f(y);   x--; g(y); }
}
int main ()
{
   g(x)
   return 0;
}
```

	x: 2 f::x 1	global static area
	return value	activation record of main
bp →	m: 2 main bp return address	call to g( )
sp →	y: 1	

ACTIVATION RECORDS

```
int x = 2;

void f(int n)
{  static int x= 1;
   g(n);
   x--;
}
void g(int m)
{  int y=m-1;
   if (y>0)
     {  f(y);   x--; g(y); }
}
int main ()
{
   g(x)
   return 0;
}
```

	x: 2 f::x 1	global static area
	return value	activation record of main
	m: 2 main bp return address y: 1	call to g( )
bp →	n: 10 g: bp	call to f( )
sp →	return address	

Activation Records

```
int x = 2;

void f(int n)
{  static int x= 1;
   g(n);
   x--;
}
void g(int m)
{  int y=m-1;
   if (y>0)
     {  f(y);   x--; g(y); }
}
int main ()
{
   g(x)
   return 0;
}
```

	Stack	
	x: 2 f::x 1	global static area
	return value	activation record of main
	m: 2 main bp return address y: 1	call to g( )
	n: 10 g: bp return address	call to f( )
bp →	m: 1 main bp return address	call to g( )
sp →	y: 0	