Chapter 7
Runtime Environments

2022 Spring&Summer

Outline

- Runtime Environment;
- Three kinds of runtime environments:

  (1) Fully static environment; FORTRAN77

  (2) Stack-based environment; C C++

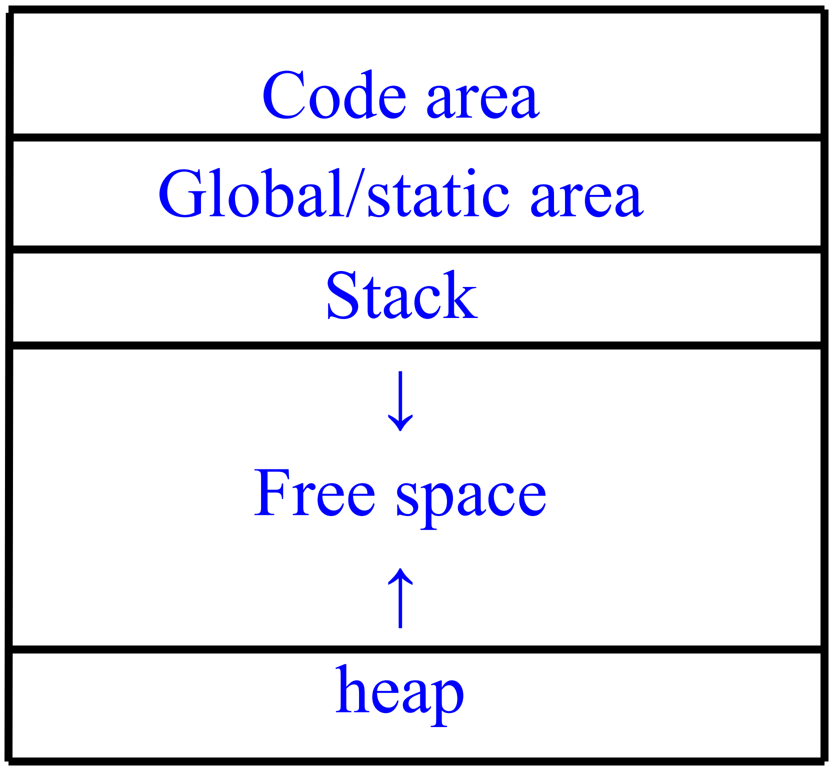
  (3) Fully dynamic environment; LISP

7.1 Memory Organization During Program Execution

- The memory of a typical computer:
  - A register area;
  - Addressable Random access memory(RAM):
    - A code area
    - A data area

7.1 Memory Organization During Program Execution

The general organization of runtime storage



7.1 Memory Organization During Program Execution

Procedure activation record

Space for arguments (parameters)
Space for bookkeeping information, including return address
Space for local data
Space for local temporaries

7.1 Memory Organization During Program Execution

Procedure activation record

- Depending on the language, activation records may be allocated in different areas:
  - Fortran77 in the static area;
  - C and Pascal in the stack area; referred to as stack frames
  - LISP in the heap area.

7.1 Memory Organization During Program Execution

Processor registers

- Processor registers are also part of the structure of the runtime environment.
- Special-purpose registers to keep track of execution

PC	program counter;
SP	stack pointer;
FP	frame pointer;
AP	argument pointer

7.1 Memory Organization During Program Execution

Calling sequence

- The allocation of memory for the activation record;
- The computation and storing of the arguments;
- The storing and setting of necessary registers to affect the call

7.1 Memory Organization During Program Execution

Returning sequence

- The placing of the return value where it can be accessed by the caller;
- The readjustment of registers;
- The possible releasing for activation record memory.

7.2 Fully Static Runtime Environments

- All data are static, remaining fixed in memory for the duration of program execution.
- Used for a language, such as FORTRAN77.

  no pointer or dynamic allocation.

  no recursive procedure calling.

7.2 Fully Static Runtime Environments

The entire program memory

Code for main procedure	
Code for procedure 1	
…	Code area
Code for procedure n	
Global data area	
Activation record of main procedure	
Activation record of procedure 1	Data area
…	
Activation record of procedure n	

7.2 Fully Static Runtime Environments

The entire program memory

- The global variables and all variables are allocated statically.
- Each procedure has only a single activation record.
- All variables (local or global) can be accessed directly via fixed address.
- No extra information about the environment needs to be kept in an activation record.

7.2 Fully Static Runtime Environments

The calling sequence (simple)

- Each argument is computed and stored into its appropriate parameter location in the activation of the procedure being called.
- The return address in the code of the caller is saved.
- A jump is made to the beginning of the code of the called procedure.
- On return, a simple jump is made to the return address.

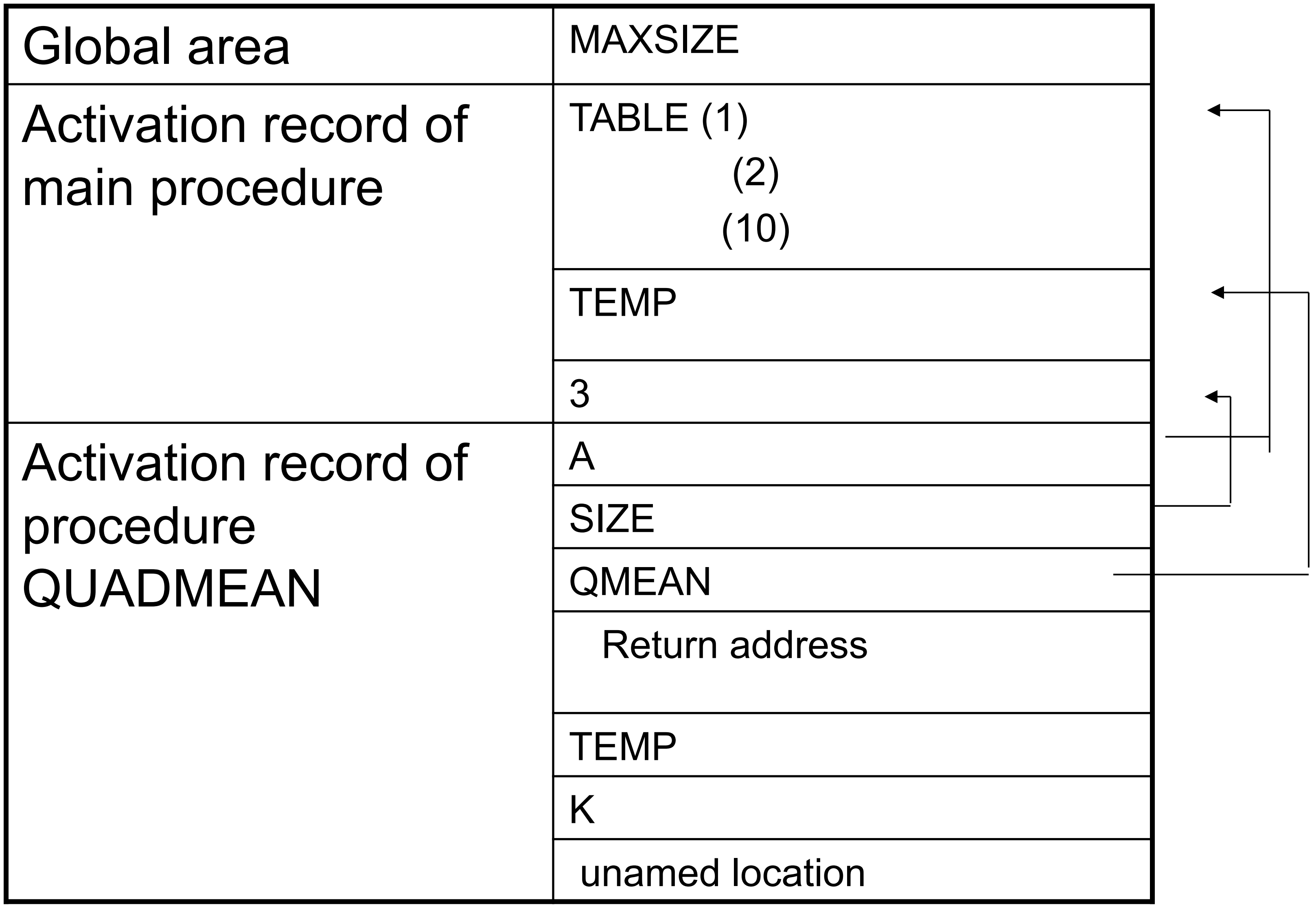
Example: A FORTRAN77 sample program

```
PROGRAM TEST
COMMON          MAXSIZE
INTEGER         MAXSIZE
REAL            TABLE(10),TEMP
MAXSIZE = 10
READ *, TABLE(1),TABLE(2),TABLE(3)
CALL QUADMEAN(TABLE,3,TEMP)
PRINT *, TEMP
END
```

Example: A FORTRAN77 sample program

```
      SUBROUTINE          QUADMEAN(A, SIZE,QMEAN)
      COMMON       MAXSIZE
      INTEGER      MAXSIZE,SIZE
      REAL A(SIZE),QMEAN,TEMP
      INTEGER K
      TEMP=0.0
      IF ((SIZE .GT. MAXSIZE) .OR. (SIZE .LT. 1) GOTO 99
      DO 10 K=1,SIZE
         TEMP=TEMP+A(K)*A(K)
10     CONTINUE
99     QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

A runtime environment for the example program



A runtime environment for the example program

- In FORTRAN77, parameter values are implicitly **memory references**, so the locations of the arguments of the call(TABLE, 3, and TEMP) are copied into the parameter locations of QUADMEAN.

1. an extra dereference is required to access parameter values.
2. array parameters do not need to be reallocated and copied.
3. constant arguments must be stored to a memory location and this location used during the call.

The unnamed location is used to store temporary value during the computation of arithmetic expression.

7.3 Stack-based runtime environments

- For a language, in which
  - Recursive calls are allowed
  - Local variables are newly allocated at each call
  - Activation records cannot be allocated statically. Instead, activation records must be allocated in a stack-based fashion.

7.3 Stack-based runtime environments

- The stack of activation records grows and shrinks with the main of calls in the executing program.
- Each procedure may have several different activation records on the call stack at one time.
- More complex strategy for bookkeeping and variable access.

7.3 Stack-based runtime environments

- For a language, in which
  - *Recursive calls* are allowed
  - Local variables are *newly* allocated at each call
  - Activation records *cannot* be allocated statically. Instead, activation records must be allocated in a *stack-based* fashion.

7.3 Stack-based runtime environments

- The stack of activation records grows and shrinks with the main of calls in the executing program.
- Each procedure may have several different activation records on the call stack at one time.
- More complex strategy for bookkeeping and variable access.

7.3.1 Stack-Based Environments Without Local Procedures

- In a language where all procedures are global(such as C language), the stack-based environment requires two things

  (1) Frame pointer or *fp*, a pointer to the current activation record to allow access to local variable.

  control link or dynamic link, a point to a record of the immediately preceding activation.

  (2) Stack pointer or *sp*, a point to the last location allocated on the call stack.

7.3.1 Stack-Based Environments Without Local Procedures

Example

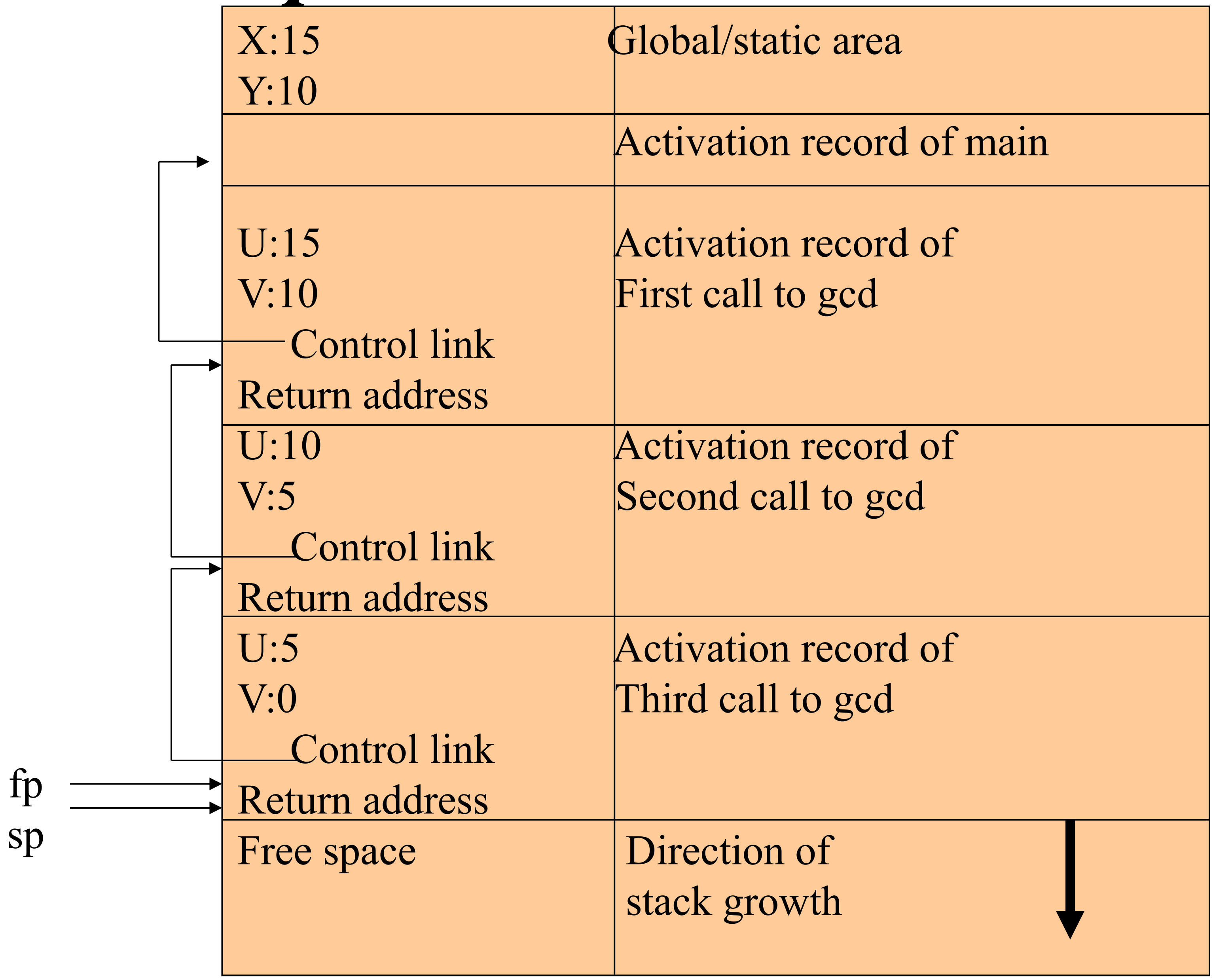
```
# include <stdio.h>
 int x,y;
 int gcd(int u,int v)
 { if (v==0) return  u;
         else return gcd(v,  u%v);
 }

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

7.3.1 Stack-Based Environments Without Local Procedures

Example

- Suppose the user inputs the values *15* and *10* to this program.
- In each new activation record, the control link points to the control link of the previous activation record.
- The *fp* points to the control link of the current activation record.

Example

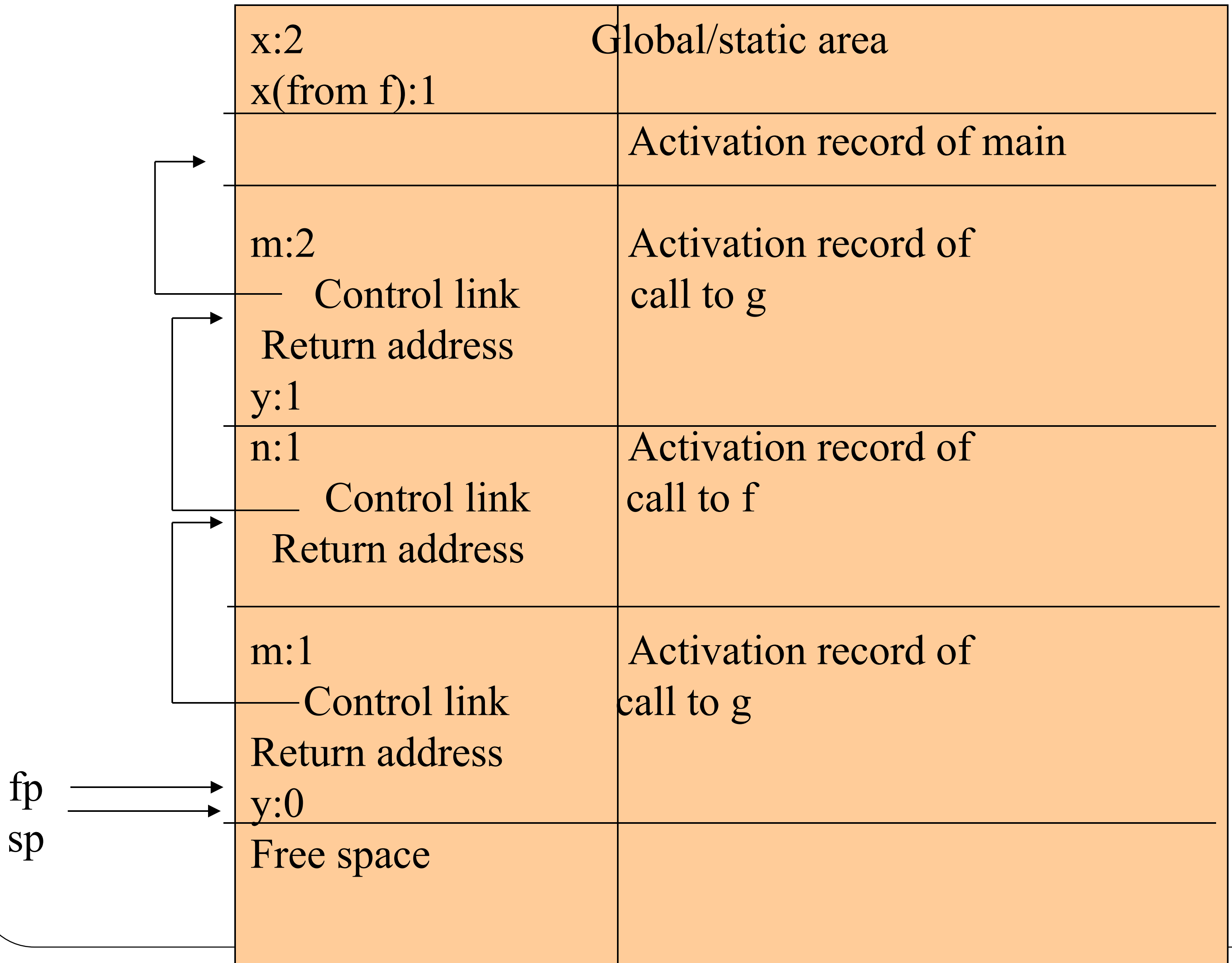

X:15
Y:10
Global/static area
Activation record of main
U:15
V:10
Control link
Return address
Activation record of
First call to gcd
U:10
V:5
Control link
Return address
Activation record of
Second call to gcd
U:5
V:0
Control link
Return address
Activation record of
Third call to gcd
fp
sp
Free space
Direction of
stack growth

Example

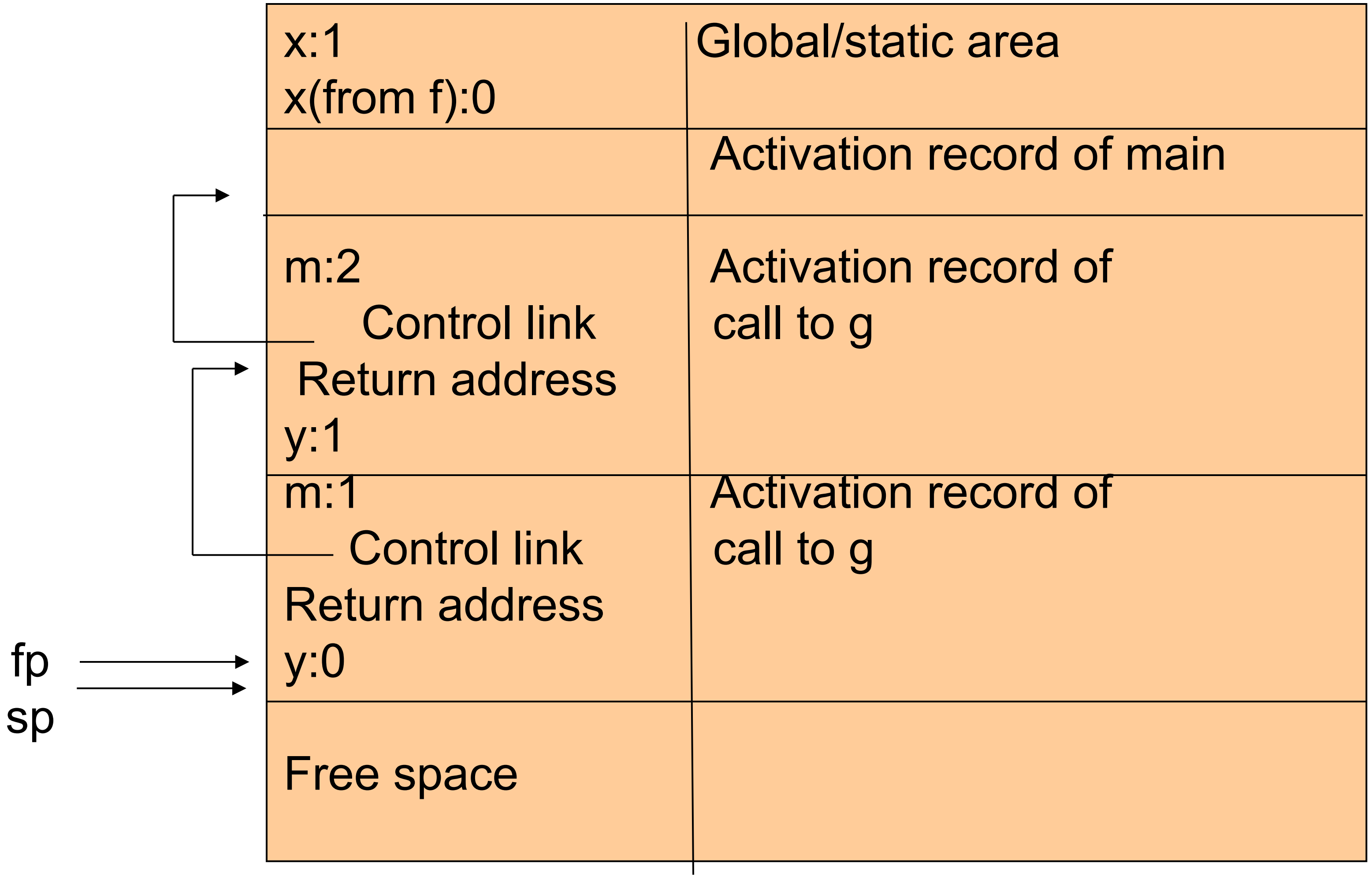
```
int x=2;
void g(int);/*prototype*/

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);
  }
 }

main( )
{g(x);
 return 0;
}
```

(a) Runtime environment of the above program during the second call to g

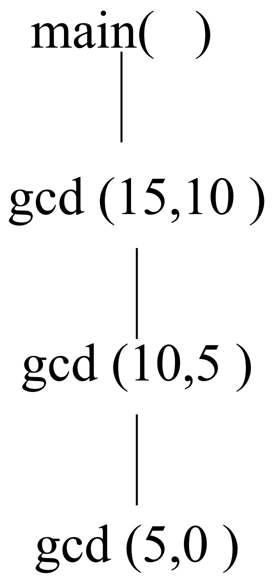


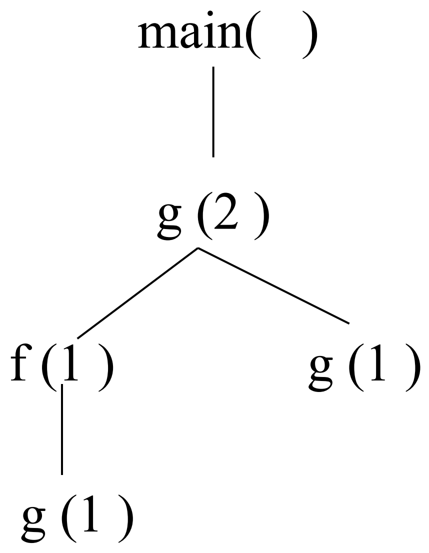
(b) Runtime environment of the above program during the third call to g



7.3.1 Stack-Based Environments Without Local Procedures

Activation tree: a useful tool for the analysis of complex calling structures. Each activation record (or call) becomes a node in this tree.





7.3.1 Stack-Based Environments Without Local Procedures

Access to Names

- Parameters and local variable must be found by offset from the current frame pointer.
- In most language, the offset can be statically computable by the compiler.

7.3.1 Stack-Based Environments Without Local Procedures

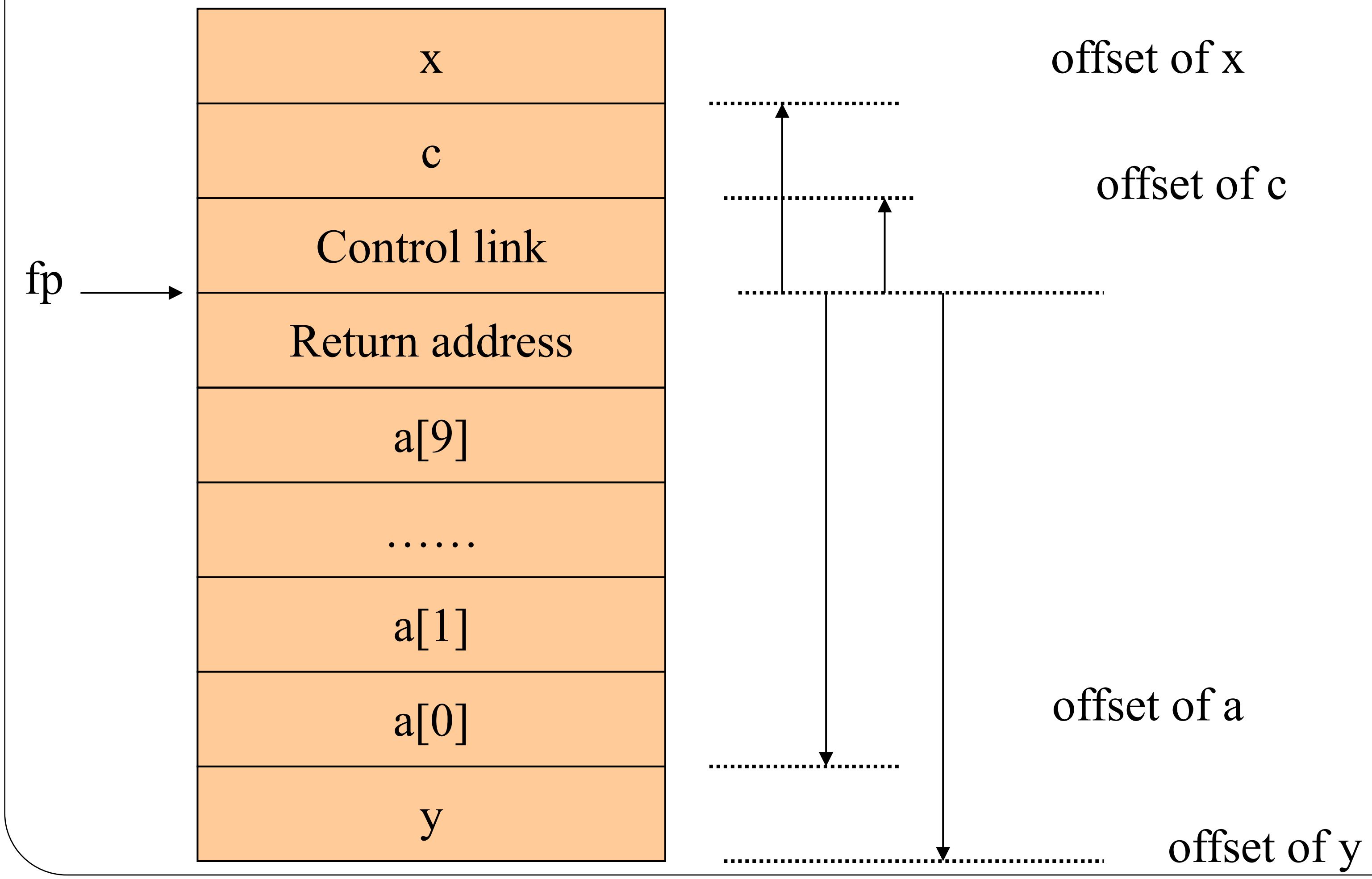
Example 7.4

C procedure：

```
void f(int x, char c)
{ int a[10];
  double y;
   …
}
```

7.3.1 Stack-Based Environments Without Local Procedures

Example 7.4



7.3.1 Stack-Based Environments Without Local Procedures

Example 7.4

Name	Offset
x	+5
c	+4
a	-24
y	-32

7.3.1 Stack-Based Environments Without Local Procedures

The calling sequence

(1) Compute the *arguments* and store them in their correct positions in the new activation record of the procedure.

(2) Store the *fp* as the control link in the new activation record;

(3) Change the *fp* so that it points to the beginning of the new activation record;

(4) Store the *return address* in the new activation record;

(5) Perform a *jump* to the code of the procedure to be called.

7.3.1 Stack-Based Environments Without Local Procedures

When a procedure exits

(1) Copy the *fp* to the *sp*.

(2) Load the control link into the *fp*.

(3) Perform a *jump* to the *return address*.

(4) Change the *sp* to pop the *arguments*.

7.3.1 Stack-Based Environments Without Local Procedures

Dealing with variable-length data

Data may vary both in the number of data objects and in the size of each object.

(1) The number of arguments in a call may vary from call to call.

(2) The size of an array parameter or a local array variable may vary from call to call

printf("%d%s%c",n,prompt,ch)	Has four arguments
printf("Hello, world\n")	Has only one argument

7.3.1 Stack-Based Environments Without Local Procedures

Dealing with variable-length data

- C compiler typically deal with this by pushing the arguments to a call *in reverse order* onto the runtime stack. The first parameter is always located at a fixed offset from the *fp* in the implementation described above.

- Another option is to use a processor mechanism such as *ap*(argument pointer) in VAX architecture.

7.3.1 Stack-Based Environments Without Local Procedures

Local Temporaries and Nested Declarations

- Local temporaries are partial results of computations that must be saved across procedure calls, for example:

  $$x[i] = (i + j) *(i/k + f(j))$$

- The three partial results need to be saved across the call to *f*:

  The *address* of x[i];

  The sum $i+j$;

  The quotient i/k;

7.3.1 Stack-Based Environments Without Local Procedures

Local Temporaries and Nested Declarations

- Nested declarations present a similar problem.

```
void p( int x, double y)
{ char a;
  int i;
…
A: {double x;
          int j;
          …
          }
…
B:{char *a;
          int k;
          …
          }
          …
}
```

7.3.1 Stack-Based Environments Without Local Procedures

- The nested local declarations do not need to be allocated until entered;
- The declarations of A and B do not need to be allocated simultaneously.
- A compiler could treat a block just like a procedure and create a new activation record each time a block is entered and discard it on exit. -- *This would be inefficient.*
- A simpler method is to treat them in a similar way to temporary expression.

7.3.2 Stack-Based Environment with local Procedures

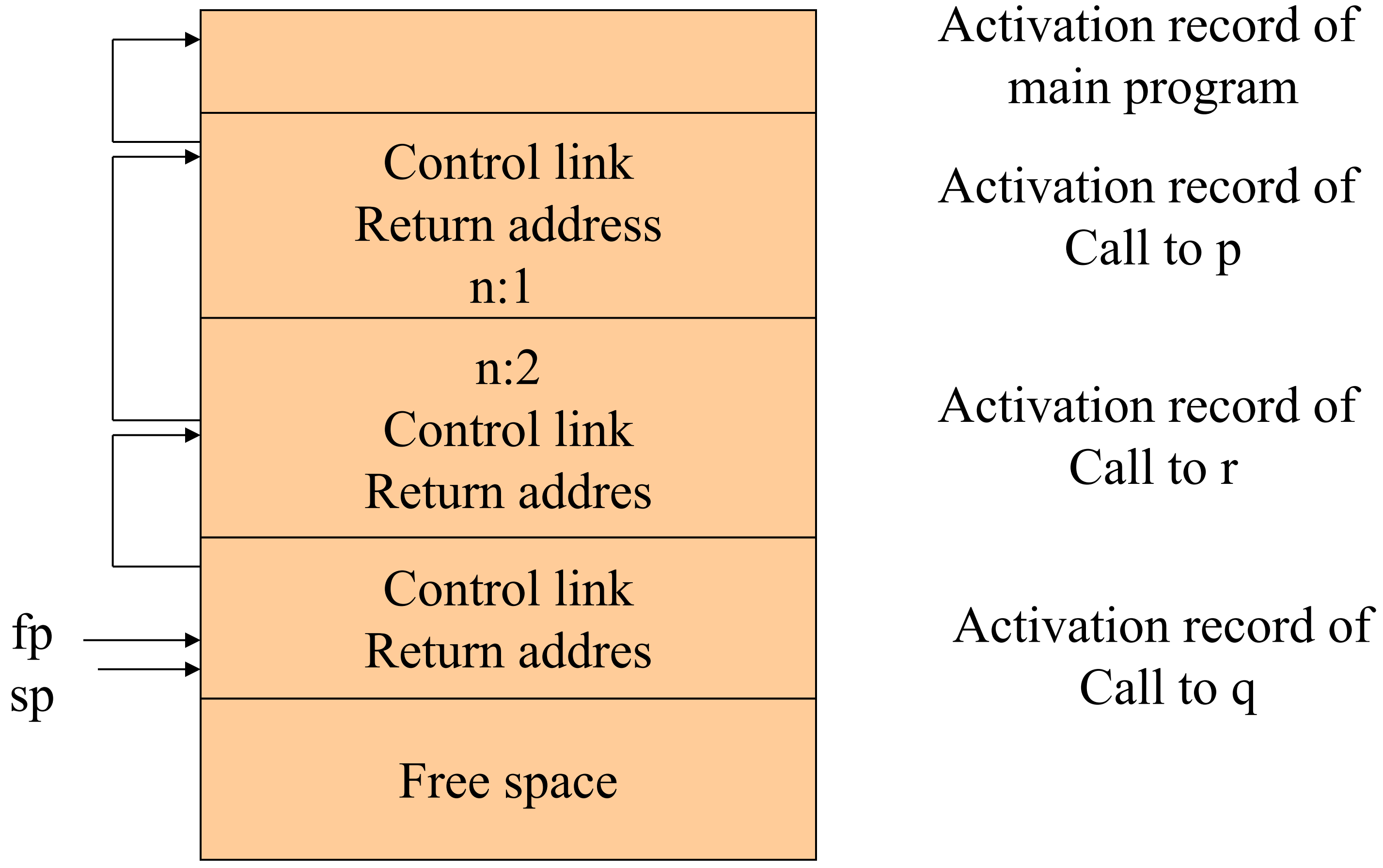
- Consider the non-local and non-global references

  Example: Pascal program showing *nonlocal,nonglobal reference*
- To solve the above problem about variable access, we add an extra piece of bookkeeping information called the *access link* to each activation record.

7.3.2 Stack-Based Environment with local Procedures

```
program nonlocalRef;

procedure  p;
var n: integer;

      procedure q;
      begin
            (* a reference to n is
            now non-local and
             non-global *)
      end;  (*q*)

      procedure r(n:integer);
      begin
          q;
      end; (*r*)

begin  (* p*)
        n :=1;
        r(2);
end;  (*p*)

begin (*main*)
   p;
end.
```

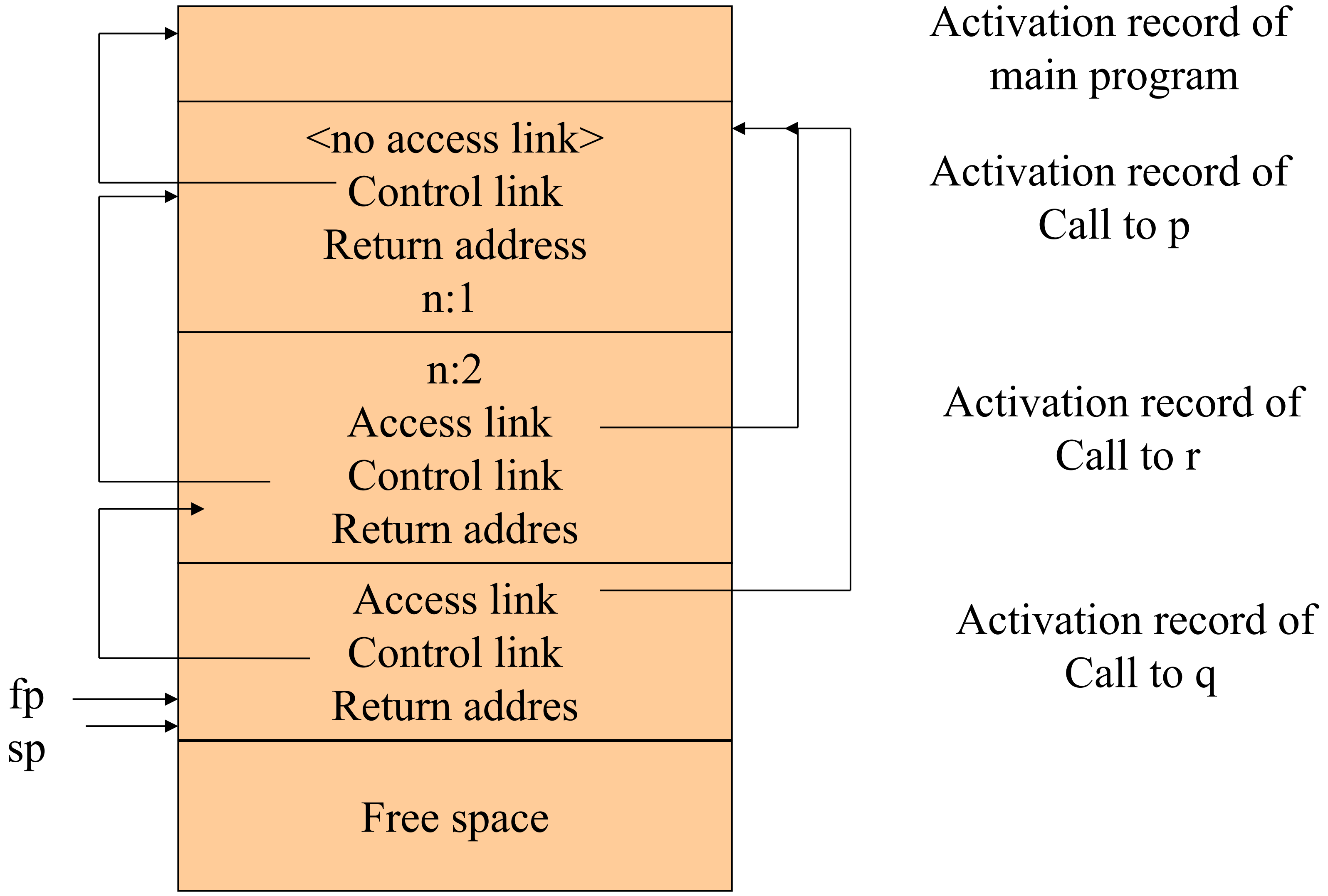
7.3.2 Stack-Based Environment with local Procedures



7.3.2 Stack-Based Environment with local Procedures

- *access link* represents the *defining environment* of the procedure; access link is sometimes also called the static link.
- *control link* represents the calling environment of the procedure.

7.3.2 Stack-Based Environment with local Procedures



7.3.2 Stack-Based Environment with local Procedures

- The calling sequence:

  (1) The access link must be pushed onto the runtime stack just before the *fp* during a call

  (2) The *sp* must be adjusted by an extra amount to remove the access link after an exit.

- How to find the *access link* of a procedure during a call.

  (1) Using the (compile-time) nesting level information attached to the declaration of the procedure;

  (2) Generate an access chain as if to access a variable at the same nesting level.

Homework of Chapter 7

7.4 Draw the stack of activation records for the following Pascal program , showing the control and access links , after the second

call to procedure **c**.Describe how the variable **x** is accessed from within **c**.

7.15 Give the output for the following program(written in C synt- ax)using the four parameter passing methods discussed in Section 7.5.

Homework of Chapter 7

7.4

```
program env;

procedure a;
var x: integer;

  procedure b;
        procedure c;
        begin
            x := 2;
            b;
        end;
  begin (* b *)
            c;
        end;

begin (* a *)
     b;
end;

begin (* main *)
       a;
  end.
```

7.15

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{
  x += 1;
  i += 1;
  y += 1;
}

main()
{
  int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```