RECURSION

Recursion

- A function defined in terms of itself is called recursive.
  Example: Factorial, Fibonacci Series, …
- The power of recursion lies in the possibility of defining a potentially infinite set of objects by a finite set of statements.
- In most High Level Languages, subprograms can be declared recursively.
- Most of the modern processors support the mechanism of recursion at the machine level.

General Definition

$f(X_0)$ =<expression> [independent on f]

$f(X_i) = G(X_i, f(X_j))$ [generally j<i]

- Since $f(X_i)$ depends on $f(X_j)$ and not on $f(X_i)$ itself , the above definition is not circular.

Types of Recursion

- **Linear Recursion:** A recursive algorithm in which only one internal recursive call is made within the body is called linearly recursive.

```
function L (…)
        {   if   base condition satisfied
                    then return some value(s)
                    else  {
                                perform some actions;
                                make call to L;
                            }
        }
```

Examples: Factorial, GCD, Binary Search, etc.

- Large depth of recursion , simple iterative solution.

Types of Recursion…

- **Binary Recursion:** An algorithm that makes two internal calls to itself is said to be binary recursive.

  Examples: Fibonacci sequence, Quicksort, Mergesort, Binary Tree algorithms, General Divide–and–conquer algorithms.

```
function B(…)
        { if  base condition satisfied
                then perform actions/ return value(s)
                else { perform some action(s);
                        make a call to B to solve one
                        smaller problem ;
                        make a call to B to solve the other
                        smaller problem ;
                    }
        }
```

Types of Recursion…

- **Non-linear recursion:** An algorithm using a number of (> 2) internal recursive calls within the body of the procedure is non-linear recursive.

```
function  N(…)
        { for j:= k to n do
                { perform some action(s);
                        if base condition not satisfied
                        then make a call to N
                        else perform some action(s);
                }
        }
```

Examples: Sample Generation, Combination generation, Permutation generation, etc.

Types of Recursion…

- **Mutual Recursion:** In this type of recursion, a function calls itself indirectly via another function that calls itself indirectly via the first function.

```
function M1(…)
        {
                :
                call to M2;
                :
        }
function M2(…);
        {
                :
                call to M1;
                :
        }
```

- M1 and M2 are interlocked.

Examples: Recursive Descent Compilation, Hilbert Curves, etc.

Rules for writing correct and efficient recursive algorithms

1. Base Cases: Some base cases must be there. These can be solved without recursion.

2. Making Progress: For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress towards a base case. [Eventually the base case must be reached !]

3. Design Rule: Assume that the recursive calls work.

4. Compound Interest Rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.

- Recursive calls are particularly appropriate when the underlying problem or data are defined in recursive terms.

- But not all recursive algorithms are efficient.

A bad example

```
void print-list (T *  Head) {
        if  (!empty_l(Head)) {
                    print_info( Head -> info);
                   print_list(Head -> next);
                              }
        }



void print_list (T * Head) {
top:   if (!empty_l(Head)){
                   print_info(Head->info);
                   Head = Head->next;
                   goto top;
              }
}
```

Recursion removed

```
void print_list (T * Head) {
        while (! (empty_l(Head))) {
                print_info (Head->info);
                Head = Head->next;
                }
        }
```

Non-terminating Recursive Programs

P1:

```
if  x == 0
then        f = 1;
else  f = f(x+1) + f(x+2);
```

- Makes progress away from the base condition!

P2:

```
if  x == 0
then  f = 1;
else  f  = f(x-2) * f(x-3);
```

- Apparently makes progress towards the base condition but may not reach the base condition.

Application of Stack in Function Call Implementation

- **As a stack is a LIFO structure, it is an appropriate data structure for applications in which information must be saved and later retrieved in reverse order.**

- Consider what happens within a computer when function main( ) calls another function.

- How does a program remember where to resume execution from, after returning from a function call?

- From where does it pick up the values of the local variables in the function main( ) after returning from the subprogram?

Function Call Implementation . . .

- As an example, let **main( )** call **a( )**. Function **a( )**, in turn, calls function **b( )**, and function **b( )** in turn invokes function **c( )**.

- **main( )** is the first one to execute, but it is the last one to finish, after **a( )** has finished and returned.

- **a( )** cannot finish its work until **b( )** has finished and returned. **b( )** cannot finish its work until **c( )** has finished and returned.

Function Call Implementation . . .

- When **a( )** is called, its calling information is pushed on to the stack (calling information consists of the address of the return instruction in **main( )** after **a( )** was called, and the local variables and parameter declarations in **main( )**.

- When **b( )** is called from **a( )**, **b( )**'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **a( )** after **b( )** was called, the local variables of **b(),** and parameter declarations in **a( )**).

Function Call Implementation . . .

- Then, when **b( )** calls **c( )**, **c( )**'s calling information is pushed onto the stack (calling information consists of the address of the return instruction in **b( )** after **c( )** was called, the local variables of **c( )** and parameter declarations in **b( )**).

- When **c( )** finishes execution, the information needed to return to **b( )** is retrieved by popping the stack.

- Then, when **b( )** finishes execution, its return address is popped from the stack to return to **a( )**

Function Call Implementation . . .

- Finally, when **a( )** completes, the stack is again popped to get back to **main( )**.
- When **main( )** finishes, the stack becomes empty.
- Thus, a stack plays an important role in function calls.
- The same technique is used in recursion when a function invokes itself.

Advantages of Function

- Functions facilitate the factoring of code. Every C program consists of one main( ) function typically invoking other functions, each having a well-defined functionality.

- Functions therefore facilitate:
  - Reusability
  - Procedural abstraction

- By procedural abstraction, we mean that once a function is written, it serves as a **black box**. All that a programmer would have to know to invoke a function would be to know its name, and the parameters that it expects.

Function Calls – A Top Level Overview

- When a function call is encountered, it involves the following steps:

1. Each expression in the argument list is evaluated.
2. The value of the expression is converted, if necessary, to the type of the formal parameter, and that value is assigned to the corresponding formal parameter at the beginning of the body of the function.
3. The body of the function is executed.

Function Calls – A Top Level Overview. . .

4. If the return statement includes an expression, then the value of the expression is converted, if necessary, to the type specified by the type specifier of the function, and that value is passed back to the calling function.

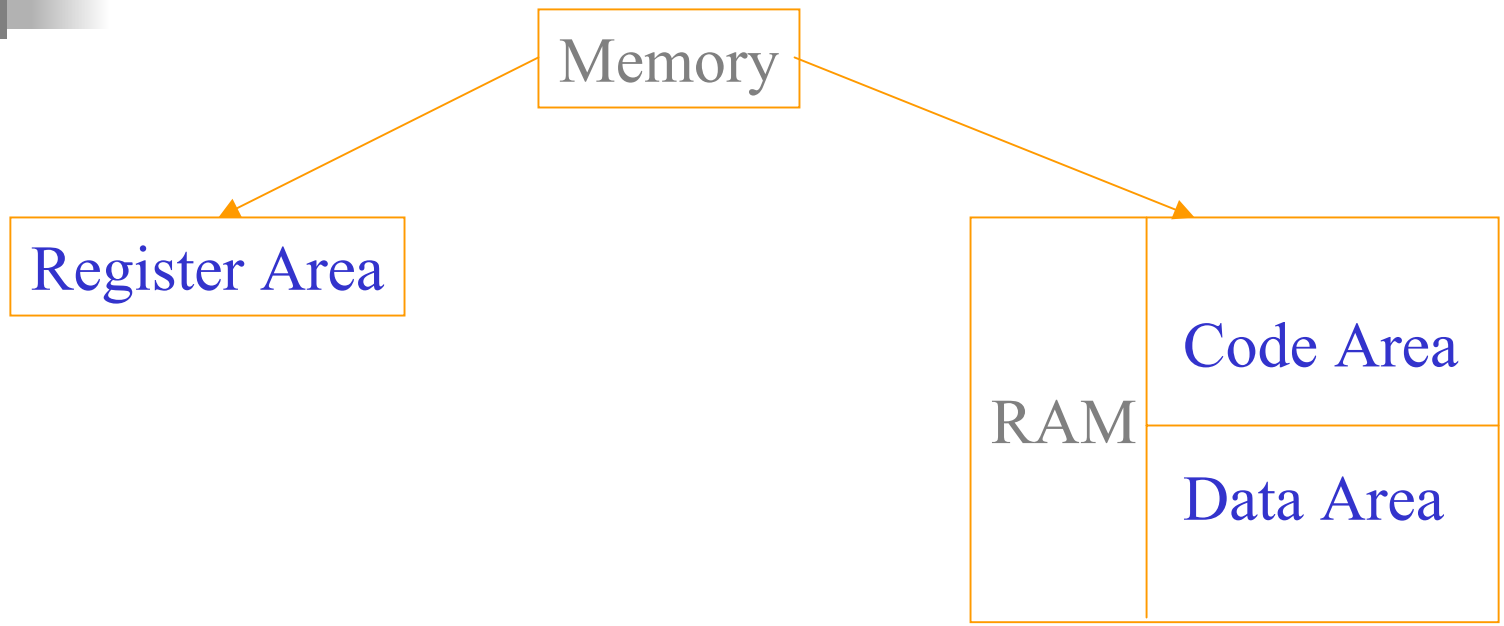
5. If no return statement is present, the control is passed back to the calling function when the end of the body of the function is reached. No useful value is returned.

Function Calls & The Runtime Stack

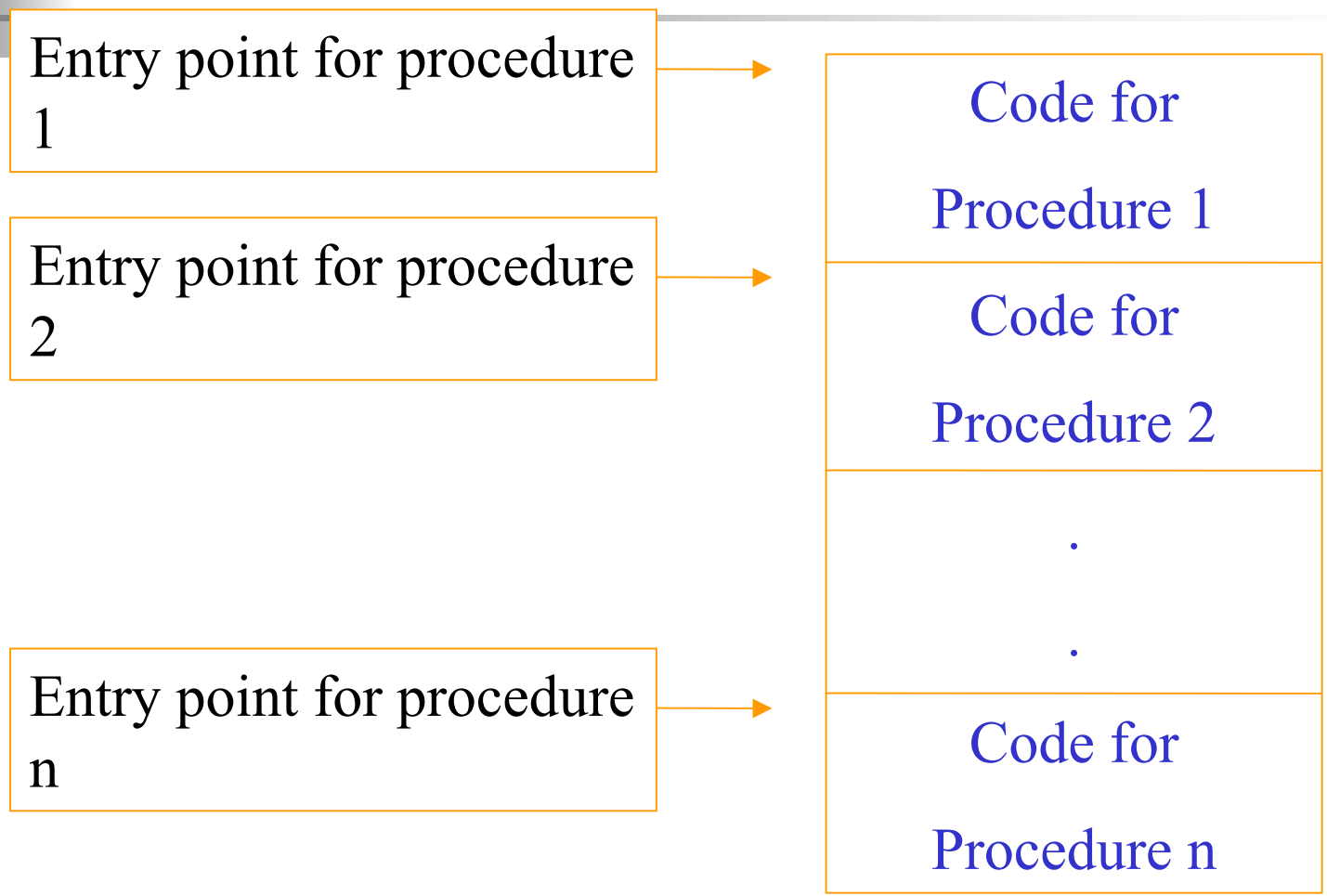
- **Runtime Environment:** Runtime Environment is the structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process.

- The C language uses a stack-based runtime environment, which is also referred to as a runtime stack, or a call stack.

Function Calls & The Runtime Stack. . .



Code Area


Entry point for procedure 1
Entry point for procedure 2
Entry point for procedure n
Code for Procedure 1
Code for Procedure 2
.
.
Code for Procedure n

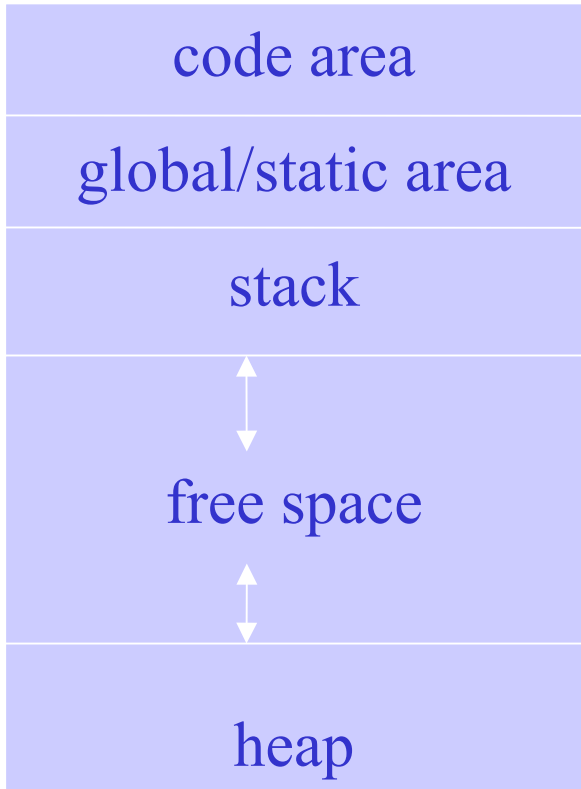
Data Area

- Only a small part of data can be assigned fixed locations before execution begins
  - Global and/or static data
  - Compile-time constants
    - Large integer values
    - Floating-point values
  - Literal strings

Dynamic Memory

- The memory area for the allocation of dynamic data can be organized in many different ways.
- A typical organization divides the dynamic memory into
  - **stack area (LIFO)**
  - **heap area**

Memory Organization



Procedure Activation Record

- Procedure activation record contains memory allocated for the local data of a procedure or function when it is called, or activated.

- When activation records are kept on stack, they are called stack frames. Details depend on the architecture of target machine and properties of the language.

A Procedure Activation Record or a Stack Frame

Registers

- Registers may be used to store temporaries, local variables, or even global variables.
- When a processor has many registers, the entire static area and whole activation records may be kept in the registers.
- Special purpose registers:
  - Program counter (PC)
  - Stack pointer (SP)

Calling Sequence

- The calling sequence is the sequence of operations that must occur when a procedure or function is called.
  - Allocation of memory for the activation record
  - The computation and storing the arguments
  - Storing and setting registers

Return Sequence

- The return sequence is the sequence of operations needed when a procedure or function returns.
  - The placing of the return value where it can be accessed by the caller
  - Readjustment of registers
  - Releasing of activation record memory

Stack-based Runtime Environment

- In a language in which recursive calls are allowed, activation records are allocated in a stack
- This stack is called the stack of activation records (runtime stack, or, simply stack).
- Each procedure may have several different activation records at one time.

Global Procedures

- In a language where all procedures are global (like the C language), a stack-based environment requires two things:
  1. A pointer to the current activation record to allow access to local variables.
     - This pointer is called the frame pointer (fp) and is usually kept in a register.
  2. The position or size of the caller's activation record
     - This information is commonly kept in the current activation record as a pointer to the previous activation record and referred as the control link or dynamic link.
     - Sometimes, the pointer is called the old fp
  3. Additionally, there is a stack pointer (sp)
     - It always points to the top of the stack

Tracing Function Calls

```
int z;
main( )
{
  int x;
  fn_a( );
  ..... .
.
.
.
}
```

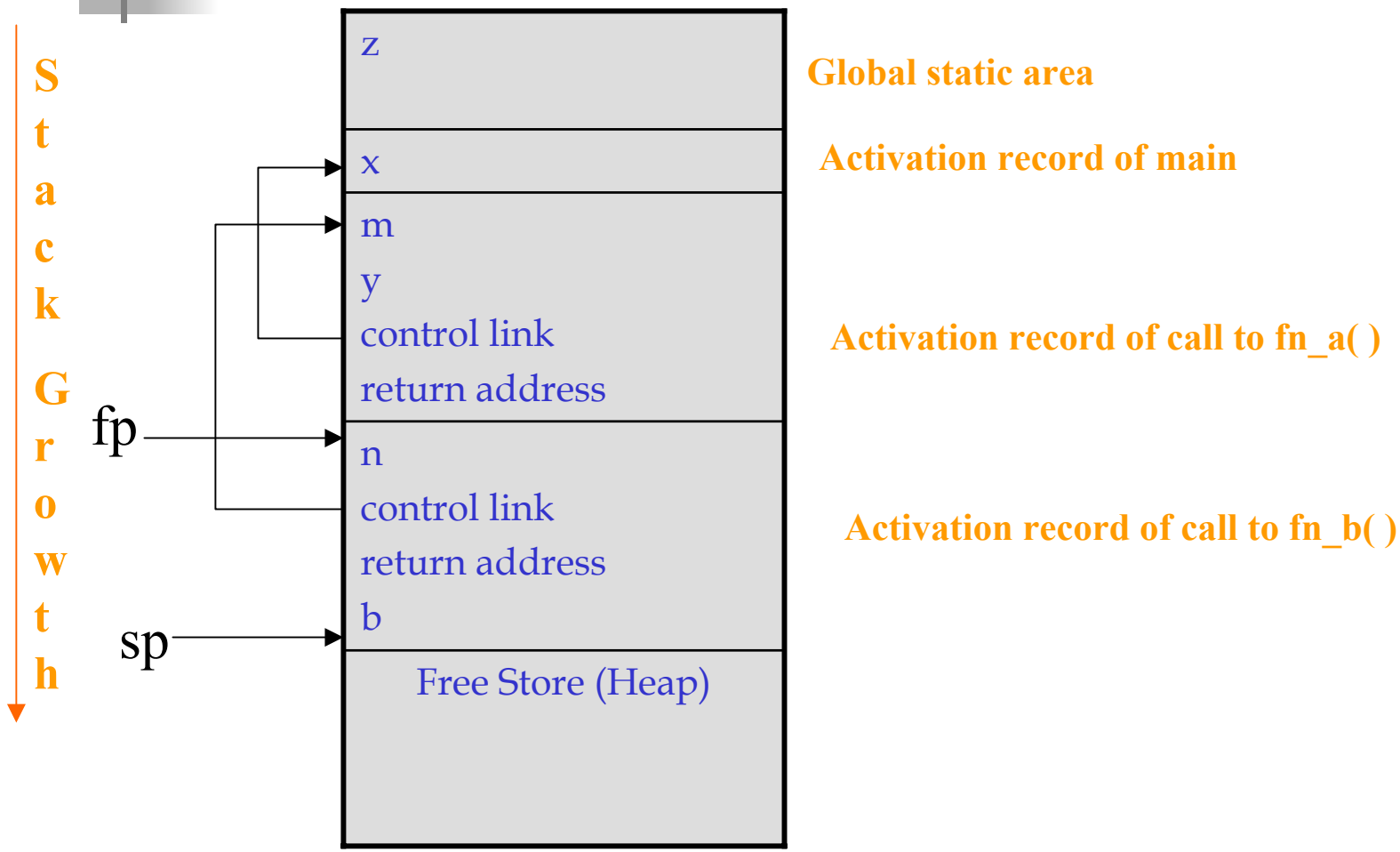
Point of return

```
fn_a( int m )
{
 int y;
 fn_b( );
 .....
.
.
.
}
```

point of return

```
fn_b( int n )
{
 int b;
  .....
.
.
.
}
```

A View of the Runtime Stack



Access to Variables

- In a stack-based environment, variables must be found by offset from the current frame pointer.
- In most languages, the offset for each local variable is still statically computable by compiler.
  - The declarations of a procedure are fixed at compile time and the memory size to be allocated for each declaration is fixed by its data type.

Calling Sequence

1. Compute the arguments and store them in their correct positions in the new activation record (pushing them in order onto the runtime stack)
2. Store (push) 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 (fp=sp)
4. Store the return address in the new activation record.
5. Jump to the code of the procedure to be called.

Return Sequence

1. Copy the fp to the sp (sp=fp)
2. Load the control link into the fp
3. Change the sp to pop the arguments.
4. Jump to the return address