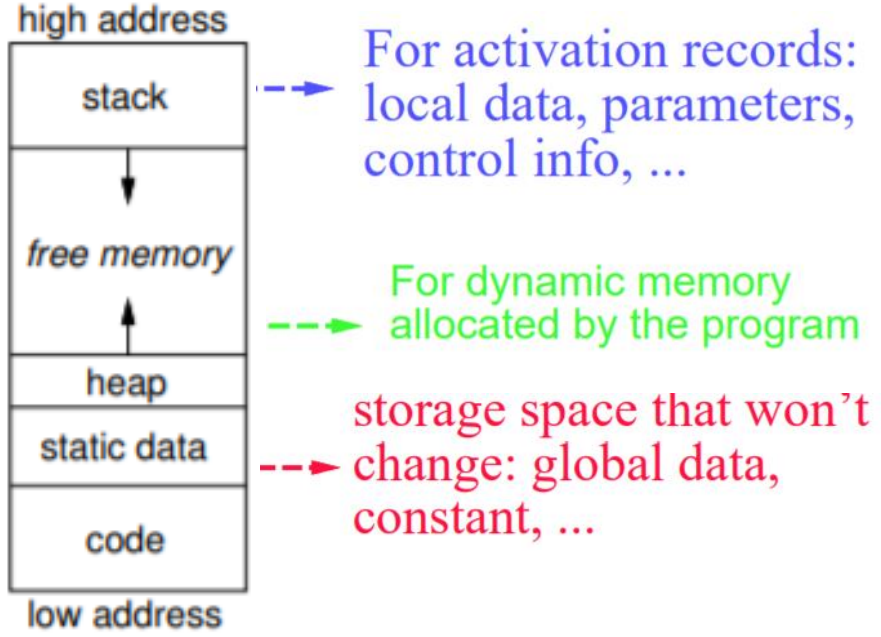
Runtime Storage Organization

Storage Organization

- The runtime storage is subdivided into
  - Target code
  - Data objects
  - Stack to keep track of procedure activation
  - Heap for dynamic memory allocation-
    - other dynamically allocated data objects at runtime



Procedure Abstraction

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point.
- Formal parameter are the on that appear in declaration.
- Actual parameters ar the one that appear in when a procedure is called.

Activation tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body.
- It returns the control to place where procedure was called from.
- A tree can be used, called **an activation tree** to depict the way control enters and leaves activations.
- The root represents the activation of main program
- Each node represents an activation of procedure
- The node **a** is parent of **b** if control flows from **a** to **b.**
- The node a is to the left of node **b** if lifetime of **a** occurs before **b**.

Example

```
main()
{
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

```
void quicksort(int m, int n)
{
    int i;
    if (n > m)
    {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

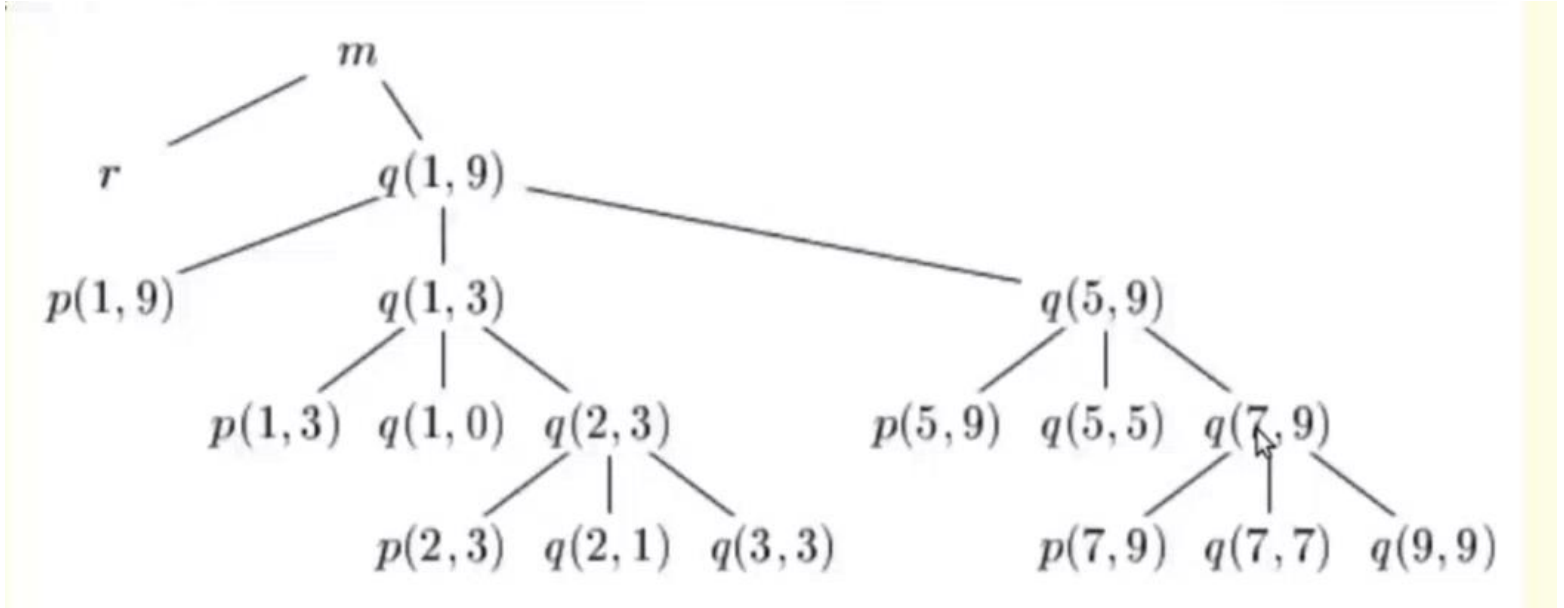
```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Example

```
main()
{
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

```
void quicksort(int m, int n)
{
    int i;
    if (n > m)
    {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
```

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```



Example - 2

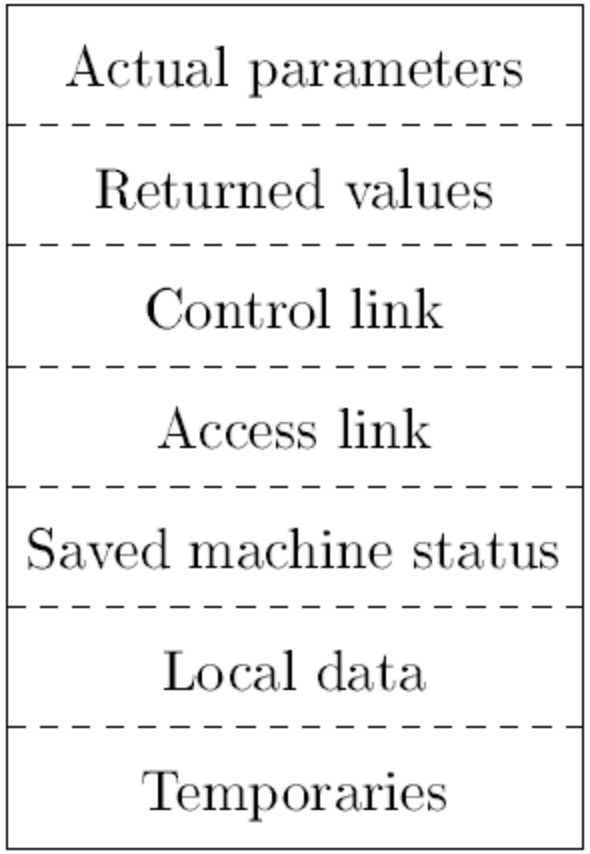
```
void Output(int n, int x){
  printf("The value of %d! is %d.\n",n,x);
}

int Fat(int n){
  int x;
  if(n > 1)
    x = n * Fat(n-1);
  else
    x = 1;
  Output(n,x);
  return x;
}

void main(){
  Fat(4);
}
```

Activation Records

- Procedure calls and returns are usually managed by a run-time stack called the **control stack**
- Each live activation has an **activation record** (sometimes called a **frame**) on the control stack.
  - with the root of the activation tree at the bottom
  - and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

Activation record contents

- The contents of activation records vary with the language being implemented.
- A pointer to the current activation record is maintained in a register
- The list of the kinds of data that might appear in an activation record

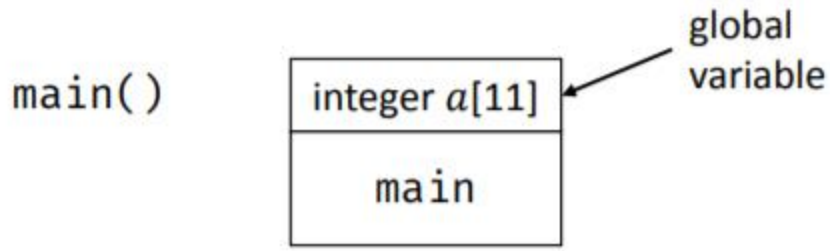


Activation record contents

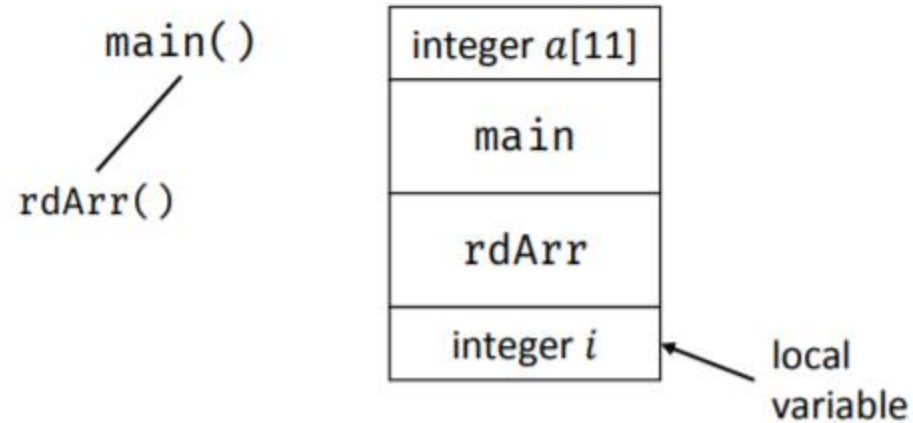
- **Temporary values** - those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
- **Local data** belonging to the procedure whose activation record
- A **saved machine status** –
  - information about the state of the machine just before the call to the procedure.
    - Includes the return address (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs
- An **access link** may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record
- **A control link**, pointing to the activation record of the caller
- **Returned values** – Space for the value to be returned
- **Actual parameters** – Space for actual parameters

Contents and position of fields may vary with language and implementations

Example of Activation Records

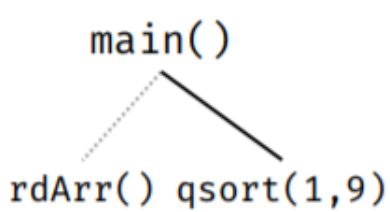


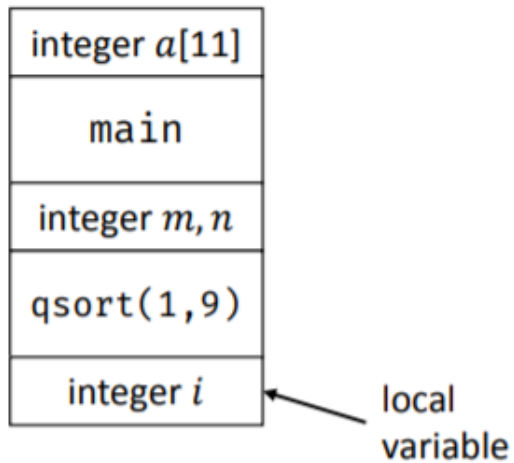
Frame for `main()`



`rdArr()` is activated

Example of Activation Records





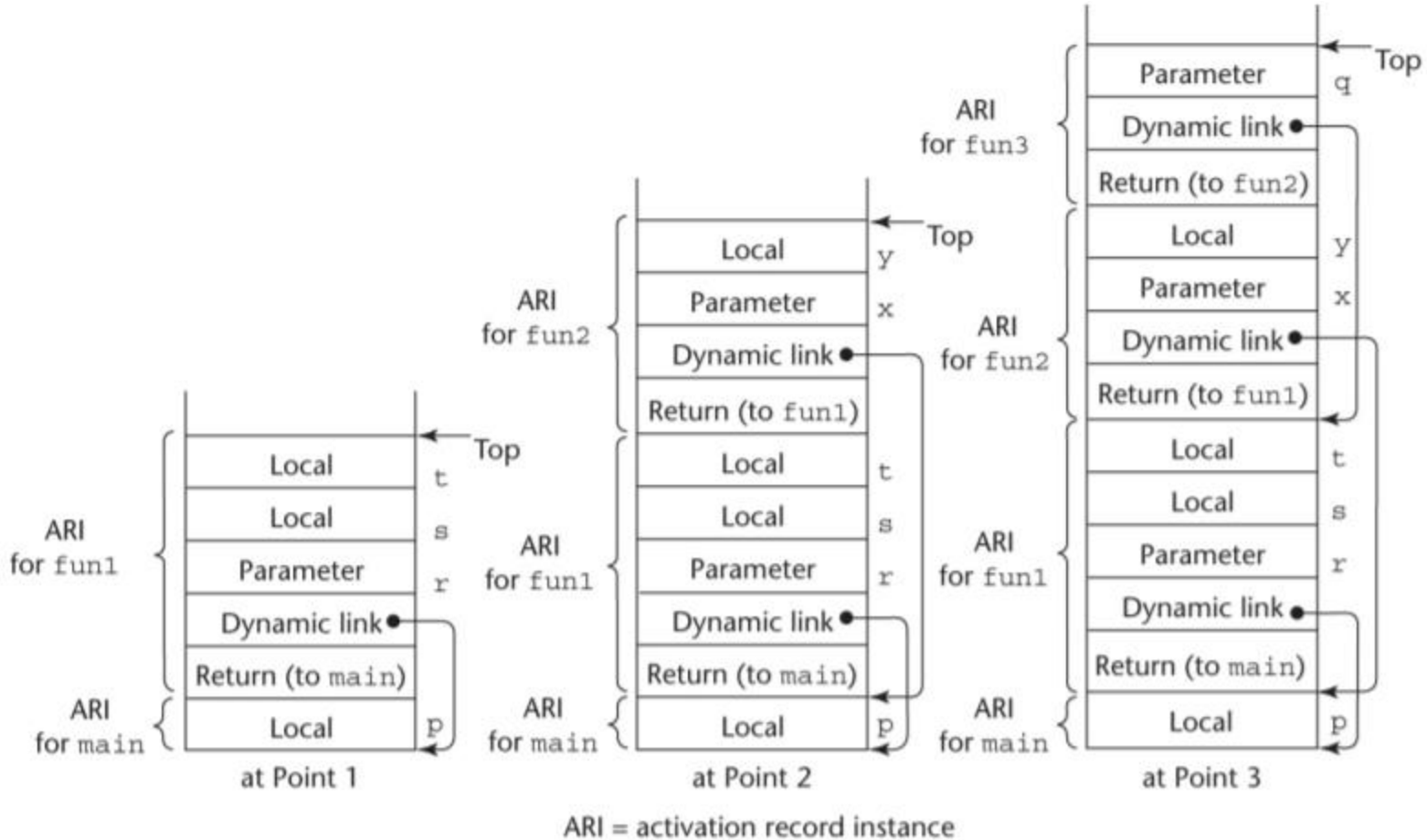
`rdArr()` is popped, `qsort(1,9)` is pushed

```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}
void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
void fun3(int q) {
    ...
}
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

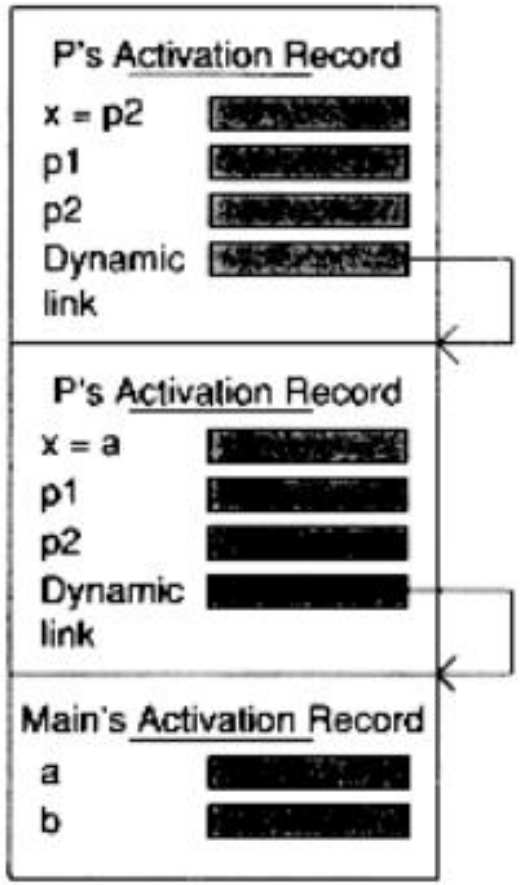
`main` **calls** `fun1`

`fun1` **calls** `fun2`

`fun2` **calls** `fun3`



```
PROGRAM Main
  LOCAL a,b
  PROCEDURE P(PARAMETER x)
    LOCAL p1,p2
  BEGIN {P}
    Call P(p2) ***
  END {P}
  BEGIN {Main}
    Call P(a)
  END {Main}
```

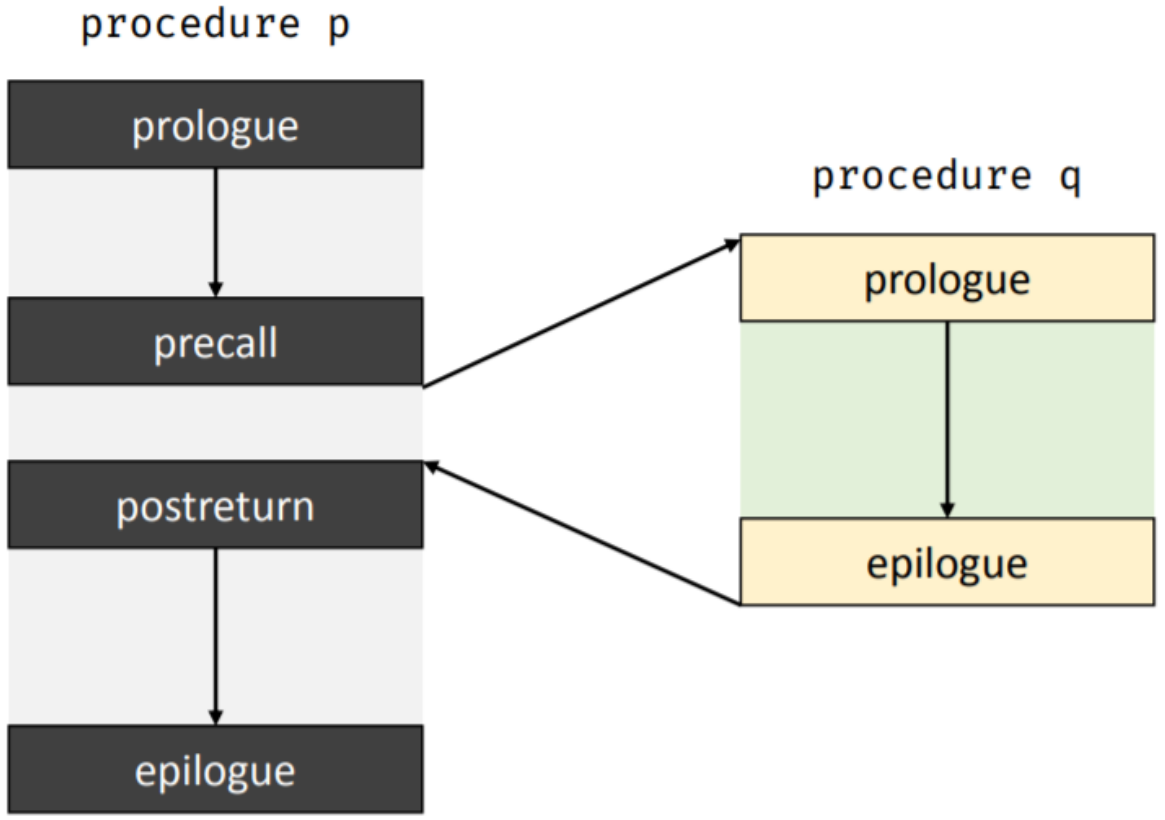


What is in G's Activation Record when F() calls G()?

- If a procedure F calls G, then G's activation record contains information about both F and G
- F is suspended until G completes, at which point F resumes
  - G's activation record contains information needed to resume execution of F
- G's activation record contains
  - G's return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G's local variables

A Standard Procedure Linkage

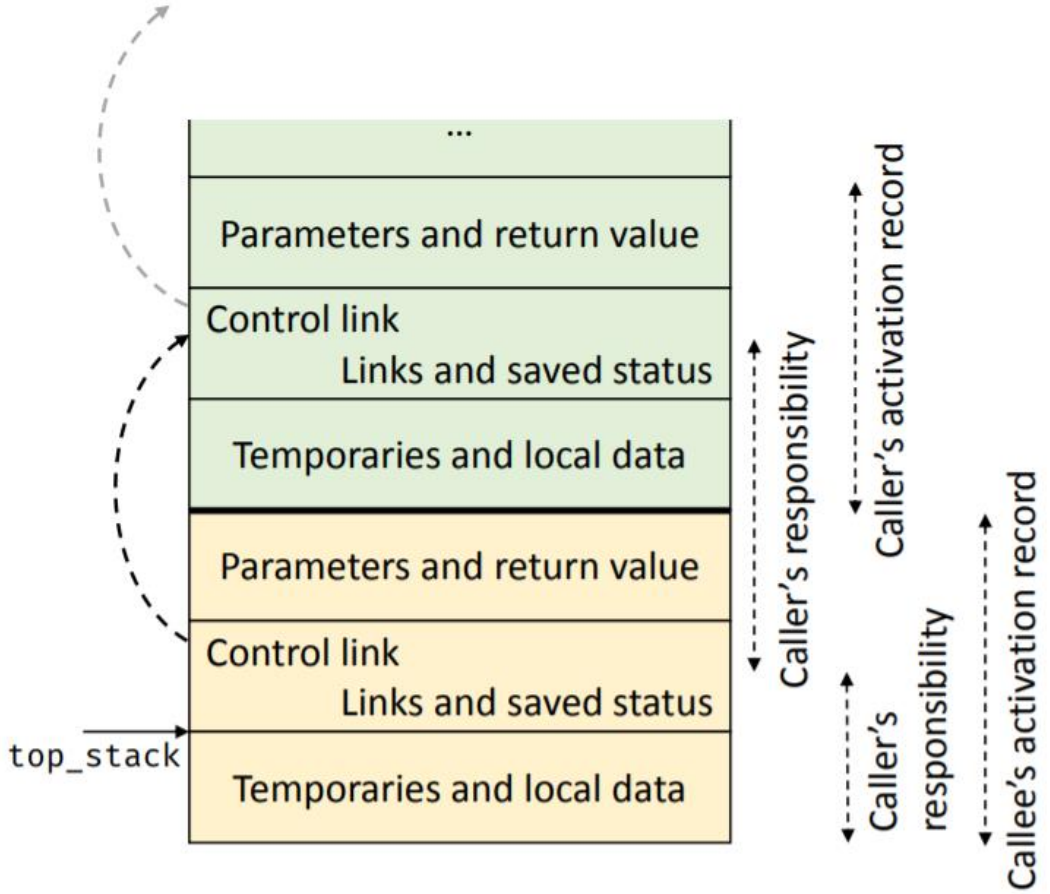
- Procedure linkage is a contract between the compiler, the OS, and the target machine
- Divides responsibility for naming, allocation of resources, addressability, and protection



Calling Sequence

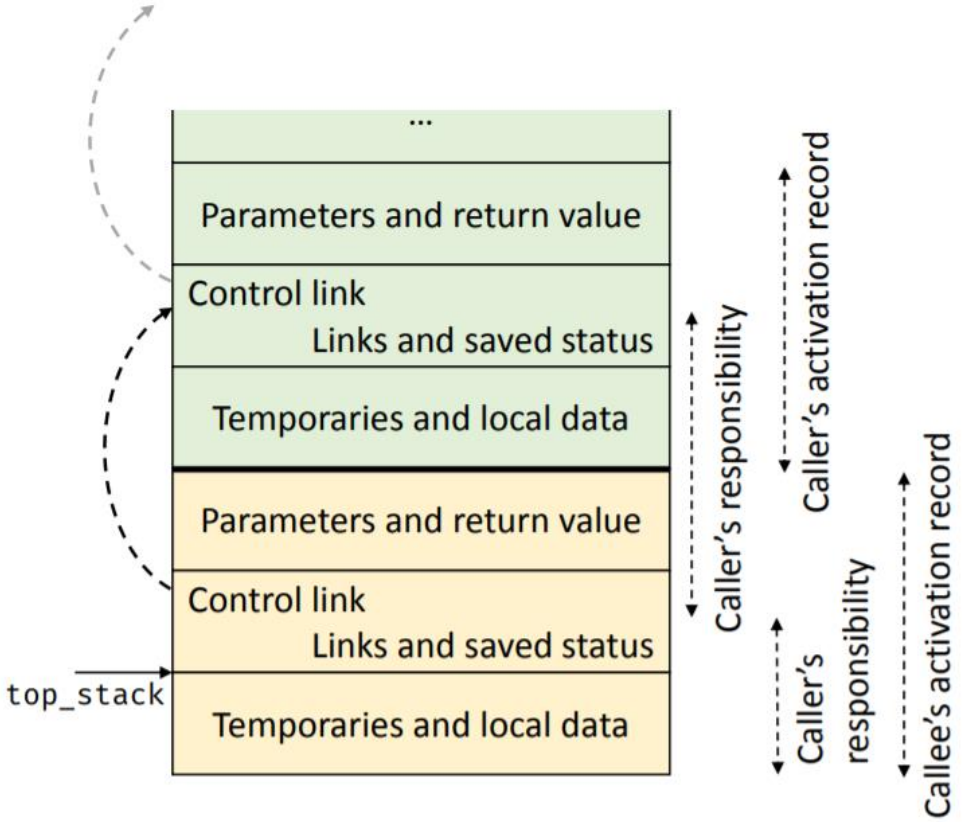
- Calling sequence allocates an activation record on the stack and enters information into its fields
  - Responsibility is shared between the caller and the callee
- Return sequence is code to restore the state of the machine so the calling procedure can continue its execution after the call

Division of Tasks Between Caller and Callee



Division of Tasks Between Caller and Callee

- Call sequence
  - **Caller** evaluates the **actual parameters**
  - **Caller** stores a **return address** and the old value of **top_stack into the callee's activation record**
  - **Caller** then **increments top_stack** past the caller's local data and temporaries and the callee's parameters and status fields
  - **Callee** saves the **register values** and other **status information**
  - **Callee** initializes its **local data** and begins **execution**



Division of Tasks Between Caller and Callee

- Return Sequence
  - **Callee** places the **return value** next to the parameters
  - **Callee** restores **top_stack** and **other registers**
  - **Callee branches** to the **return address** that the caller placed in the status field
  - **Caller copies return value** into its activation record