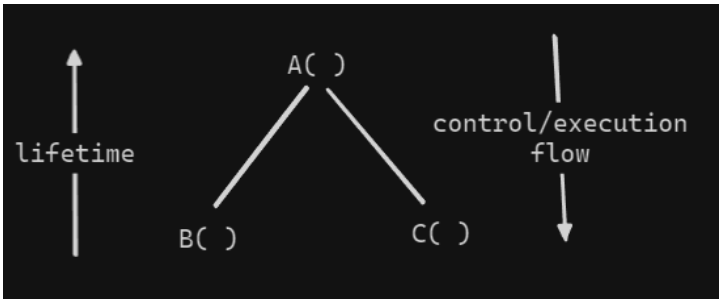
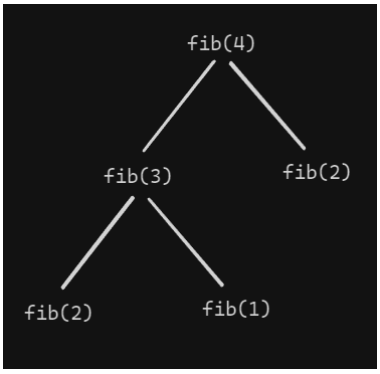
Runtime Environments:

- The time of execution of a computer program is known as the **Runtime.**
- For execution, a program is copied from secondary memory to main memory and the instructions from main memory are executed by CPU by referencing the variable names and values from main memory and CPU registers.
- The function calls / procedures are stored on a stack-based memory.
- The single execution of a function/procedure is known as an **Activation**. That is, when a procedure is called, we refer to it as “an activation has occurred”.
- Every procedure runs for a specific period of time on theCPU. The endtime of a procedure call is known as the **Lifetime** of the Procedure. (lifetime => the time at which the procedure execution is completed)
- A function/procedure can call any no.of procedures and so on. The Tree which portrays the execution flow of a procedures calls of a program is known as an **Activation tree.**

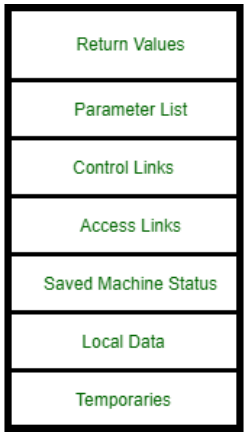


- Control flow is from A( ) to B( ) and C( ).
- Lifetime of A( ) occurs only after the lifetimes of B( ) and C( ).

- Ex: Consider a program of fibonacci series for (n = 4). The Activation tree would be:



- **Activation Record:** A Contiguous block of memory where all the information required to execute a procedure is stored, is known as an Activation record for that procedure.
- These activation records of the procedures of a program to be executed, are utilized by CPU while performing execution of respective procedures.
- These Activation Records are pushed into the stack memory, before starting the execution of the corresponding procedure call. And while executing, the status of procedure execution is stored into its respective activation record.
- An Activation Record of a procedure usually contains (7) fields of information, viz:



1. **Return Values:** This field holds the value that a function returns to the caller after its execution. It is used to pass the final result back to the calling function.
2. **Parameters List:** This field contains the actual parameters (arguments) passed to the function. These parameters are used by the callee to perform its operations.
3. **Control Link:** Also known as the dynamic link, this field points to the activation record of the caller. It helps in restoring the caller's state when the current function returns.
4. **Access Link:** Also known as the static link, this field points to the activation record of the nearest lexically enclosing function. It is used to access non-local variables in nested functions.
5. **Machine Status:** This field stores the state of the machine registers before the function call. It includes the program counter, base pointer, and other registers that need to be restored when the function returns.
6. **Local Data:** This field contains local variables declared within the function. These variables are used exclusively by the callee and are not accessible outside the function.
7. **Temporary Values:** The values of some sub-parts of expressions are known as temporary values. These are temporarily stored in a memory location while calculating a complex expression and then assessed when evaluating the final result of the expression.

Ex:

```c
int compute(int a, int b, int c) {
    int result;
    result = (a + b) * c;  // Complex expression
    return result;
}

int main() {
    int x = 2, y = 3, z = 4;
    int final_result;

    final_result = compute(x, y, z);
    printf("The result is %d\n", final_result);

    return 0;
}
```

Parameters List:

- `a = 2, b = 3, c = 4`

Local Data:

- `int result;`

Temporary Values:

- To compute `(a + b) * c`, we first need to evaluate `a + b` and temporarily store the result.
- Intermediate result of `a + b` (i.e., `2 + 3 = 5`) is stored as a temporary value.
- This temporary value (5) is then multiplied by c (i.e., `5 * 4 = 20`).

```
+-------------------+
| Return Value      |  <- Holds the result (20) to be returned to main
+-------------------+
| Parameters List   |
| a = 2             |
| b = 3             |
| c = 4             |
+-------------------+
| Control Link      |  <- Points to the caller's activation record (main)
+-------------------+
| Access Link       |  <- Points to the nearest enclosing function's activation record
+-------------------+
| Machine Status    |  <- Stores the state of machine registers
+-------------------+
| Local Data        |
| result = 20       |
+-------------------+
| Temporary Values  |
| temp1 = 5         |  <- Temporary storage for the intermediate result of a + b
+-------------------+
```

Division of tasks b/w Caller and Callee

Caller: This is the function or method that initiates the call to another function or method. It is responsible for passing the necessary arguments to the callee and handling any return values from it.
Callee: This is the function or method that is being called by the caller. It receives arguments from the caller, executes its code, and may return a value back to the caller.
Ex:

```javascript
function callee(a, b) {
    return a + b;
}

function caller() {
    const result = callee(3, 4); // caller is calling callee
    console.log(result); // prints 7
}

caller(); // Initiates the process
```

Caller Responsibilities:

1. **Prepare Arguments**:
    - The caller must prepare and pass the arguments required by the callee. This may involve placing arguments in specific registers or pushing them onto the stack.
2. **Save Caller-Saved Registers**:
    - The caller needs to save any caller-saved (volatile) registers that it wants to preserve across the function call. These are registers that the callee is allowed to modify.
3. **Call Instruction**:
    - The caller executes the instruction to call the function, often involving a jump to the callee's address.
4. **Adjust the Stack**:
    - The caller adjusts the stack pointer to make room for the arguments and return address if necessary.
5. **Receive Return Value**:
    - After the callee returns, the caller receives the return value, typically from a designated register.
6. **Restore Caller-Saved Registers**:
    - The caller restores the values of any caller-saved registers it saved before the call.
7. **Clean Up Stack**:
    - If necessary, the caller cleans up the stack by removing arguments pushed onto the stack.

Callee Responsibilities:

1. **Save Callee-Saved Registers**:
    - The callee must save any callee-saved (non-volatile) registers that it will use. These are registers that the caller expects to remain unchanged after the function call.
2. **Set Up Stack Frame**:
    - The callee sets up its stack frame, which typically includes saving the old base pointer (frame pointer) and setting the new base pointer.
3. **Function Body Execution**:
    - The callee executes its code, performing the required computations or tasks.
4. **Return Value Preparation**:
    - The callee prepares the return value and places it in a specific register designated for return values.
5. **Restore Callee-Saved Registers**:
    - The callee restores the values of any callee-saved registers it saved before executing its code.
6. **Tear Down Stack Frame**:
    - The callee tears down its stack frame by restoring the old base pointer and adjusting the stack pointer.
7. **Return Control to Caller**:
    - The callee executes the return instruction to pass control back to the caller, often using the return address stored on the stack.

Ex:

```c
#include <stdio.h>

// Callee function
int add(int a, int b) {
    int result;
    result = a + b;  // Perform the addition
    return result;   // Return the result
}

// Caller function
int main() {
    int x = 3;
    int y = 4;
    int sum;

    // Call the add function
    sum = add(x, y);

    // Print the result
    printf("The sum of %d and %d is %d\n", x, y, sum);

    return 0;
}
```

Access to non-local data on the stack:

Accessing non-local data involves understanding two distinct concepts: **lexical scope** and **dynamic scope**. These two approaches determine how variables are resolved and accessed in nested functions or procedures.

1. Lexical Scope

Lexical Scope, also known as static scope, resolves variable references based on the structure of the code. This means that a variable's scope is determined by its position in the source code and the nesting of functions or blocks. In lexical scoping, non-local variables are accessed using access links (static links) that point to the activation records of enclosing scopes.

Example in Lexical Scope:

Consider the following C-like pseudocode:

```c
void outer() {
    int a = 10;

    void inner() {
        printf("%d\n", a);  // Accessing non-local variable 'a'
    }

    inner();
}

int main() {
    outer();
    return 0;
}
```

Explanation:

- **Lexical Scoping** means that the reference to a in `inner` is resolved by looking at the structure of the code.
- The `inner` function can access a because a is in the lexical scope of `inner` due to its enclosing function `outer`.
- **Activation Record Setup:**
    - `outer`'s activation record contains the variable a.
    - `inner`'s activation record has an access link that points to `outer`'s activation record, allowing `inner` to access a.

2. Dynamic Scope

Dynamic Scope resolves variable references based on the calling context at runtime. This means that a variable's scope is determined by the order in which functions are called. In dynamic scoping, non-local variables are accessed based on the most recent binding in the call stack.

Example in Dynamic Scope:

Consider the following pseudocode with dynamic scoping:

```c
int a = 10;

void outer() {
    int a = 20;

    void inner() {
        printf("%d\n", a);  // Accessing non-local variable 'a'
    }

    inner();
}

int main() {
    outer();
    return 0;
}
```

Explanation:

- **Dynamic Scoping** means that the reference to `a` in `inner` is resolved based on the most recent binding of `a` in the call stack.
- When `inner` accesses `a`, it looks at the most recent binding of `a` in the current call stack, which is the `a` defined in `outer`, not the global `a`.
- **Activation Record Setup:**
    - The global `a` is defined in the global activation record.
    - `outer`'s activation record defines a new `a`, hiding the global `a`.
    - When `inner` is called, it looks up the call stack and finds `a` defined in `outer`'s activation record.

Summary of Differences:

- **Lexical Scope**:
    - Resolution is based on the code structure and nesting of scopes.
    - Variables are accessed using static links (access links) pointing to enclosing scopes.
    - Example Languages: C, C++, Java.
- **Dynamic Scope**:
    - Resolution is based on the calling context and the call stack at runtime.
    - Variables are accessed based on the most recent binding in the call stack.
    - Example Languages: Older languages like Perl, Emacs Lisp.

Stack for local scoping example

Stack for dynamic scoping example

Heap Management:

- While executing a program, the variables used in the program must be allocated a memory first and then the references to those memory locations are listed in the symbol table and then into registers.
- There are two types of memory allocations, based on the time of allocation, viz:
    1. Static Allocation (Compile Time Memory Allocation)
    2. Dynamic Allocation (Runtime Memory Allocation)
- The Static Allocation is useful when the variables sizes are fixed throughout the program. And it is done while compiling a program (or) before starting the execution of a program.
- The Dynamic Allocation is useful when the variables sizes are not fixed/keeps on changing, throughout the program. And it is done while executing the program.
- For Static Allocation, a stack based memory is used to push and pop the allocations.
- For Dynamic Allocation, a heap memory is used to be able to allocate and deallocate the memory for size varying variables whenever needed on runtime of a program.
- Hence, In Dynamic Allocation, there are two types of operations that can be performed on a memory, viz:
    1. Memory Allocation

2. Memory Deallocation