1. Discuss parameter passing techniques for procedure

When a procedure (or function) is called, parameters are passed from the caller to the callee.

several methods to handle this:

Types of Parameters:

Actual Parameters: The values or variables provided by the caller.

Formal Parameters: The variables defined by the function/procedure to receive the values.

The following are the primary parameter passing techniques

1. Call by Value:

In the call by value method, the actual parameter's value is copied into the formal parameter of the called procedure. The formal parameter acts as a local variable within the called procedure, and any modifications to the formal parameter do not affect the actual parameter in the calling procedure

Advantages:

- Safety: Since the actual parameter is not modified, it prevents unintended side effects in the calling procedure.
- Simplicity: The compiler can easily manage the copying of values, and no additional memory management is needed for references.
- Predictable Behavior: The caller is assured that the original data is unaffected.

Disadvantages:

- Overhead: Copying large data structures (e.g., arrays or structs) can be time consuming and memory-intensive.
- No Output Capability: Cannot be used to return values to the caller through Parameters

2. Call by Reference (Address/Location):

In the call by reference method (also known as call by address or call by location), the address of the actual parameter is passed to the called procedure. The formal parameter becomes an alias for the actual parameter, and any modifications to the formal parameter directly affect the actual parameter.

Advantages:

- Efficiency: Only the address is passed, avoiding the overhead of copying large data structures.
- Output Capability: Allows the called procedure to modify the caller's data, enabling multiple return values through parameters.
- Flexibility: Suitable for passing complex data types like arrays or structs

Disadvantages:

- Side Effects: Modifications to the formal parameter can unintentionally alter the caller's data, leading to bugs if not carefully managed.
- Complexity: The compiler must handle pointers or references, increasing the complexity of code generation.
- Aliasing Issues: Multiple references to the same memory location can cause unpredictable behavior

3. Copy-Restore (Copy-in Copy-out or Value-Result):

The copy-restore method is a hybrid of call by value and call by reference. The actual parameter's value is copied into the formal parameter at the start of the procedure (copy-in). Upon procedure exit, the final value of the formal parameter is copied back to the actual parameter (copy-out), provided the actual parameter has an L value (a memory location).

Advantages:

- Controlled Modification: Allows the called procedure to modify the caller's data without direct memory access during execution, reducing aliasing issues.
- Flexibility: Combines the safety of call by value with the output capability of call by reference.
- Useful in Distributed Systems: Suitable for remote procedure calls where data is copied across systems.

Disadvantages:

- Overhead: Requires two copy operations (copy-in and copy-out), increasing time and memory usage.
- Non-Intuitive Behavior: The actual parameter's value may change only after the procedure returns, which can be confusing.
- Dependency on L-value: Only works for parameters with a memory location, limiting its applicability

4. Call by Name:

In the call by name method, the actual parameter is treated as a macro or expression that is substituted into the function body wherever the formal parameter appears. The actual parameter is re-evaluated each time the formal parameter is referenced in the procedure.

Advantages:

- Flexibility: Allows complex expressions or computations as parameters, evaluated only when needed.
- Lazy Evaluation: The parameter is evaluated only when used, which can optimize performance in some cases.

Disadvantages:

- Overhead: Repeated evaluation of the actual parameter can be computationally expensive.
- Complexity: The compiler must manage the substitution and evaluation process, increasing code generation complexity.
- Side Effects: If the actual parameter involves variables that change between evaluations, the behavior can be unpredictable.
- Rarely Used: Less common in modern programming languages due to its complexity and potential for errors

2. What is Activation Record? Explain Stack Allocation of Activation Records with Example.

An activation record (also known as a **stack frame**) is a contiguous block of memory allocated on the runtime stack that stores **information about a single execution of a procedure (or function)**. When a function is called, an activation record is **pushed onto the call stack**, and when the function finishes, its activation record is **popped**.

It contains all the **runtime information** needed to manage a procedure call and its execution, such as Arguments passed to the function, Return address, Local variables, etc.

Structure of an Activation Record:

A typical activation record may include the following fields (from top to bottom in the stack):

1. **Temporaries** – For intermediate calculations
2. **Local Variables** – Variables declared within the function
3. **Saved Registers** – Registers that need to be restored after function call
4. **Control Link** – Pointer to the caller's activation record
5. **Access Link** – Points to the environment needed for non-local variable access (used in nested functions)
6. **Return Address** – Where to continue after function execution
7. **Actual Parameters (Arguments)** – Values passed to the function

Stack Allocation of Activation Records

Stack allocation is a storage allocation strategy where activation records are managed as a stack (last-in, first-out structure) in the runtime memory. This approach is ideal for languages that use procedures, functions, or methods, as it supports recursive calls and ensures that each procedure activation has its own isolated memory space. The stack grows when a procedure is called (pushing a new activation record) and shrinks when the procedure returns (popping the activation record).

Advantages of Stack Allocation
Dynamic Management: The stack allows dynamic allocation and deallocation of activation records as procedures are called and return.

Support for Recursion: Each recursive call creates a new activation record, preserving the state of previous calls.

Efficiency: Stack operations (push and pop) are fast, and memory is automatically reclaimed when a procedure exits

Locality: Local variables are stored close to the procedure's context, improving access speed.

Disadvantages of Stack Allocation

Fixed Size Limitation: The stack size is typically limited, and deep recursion can cause stack overflow.

Dangling References: Improper use of pointers can lead to references to deallocated memory, causing undefined behavior.

3.Describe Activation record and Activation tree

1. Activation Record.

2. Activation Tree

An activation tree is a tree-like representation of the sequence and nesting ofprocedure calls during program execution. Each node in the tree represents an activation (i.e., a single call to a procedure), and the edges represent the caller-callee relationship, showing which procedure called which other procedure. The activation tree captures the dynamic flow of control in the program.

Structure:

Root: The activation of the main program or the entry point of execution.

Nodes: Each node corresponds to an activation of a procedure, labeled with the procedure's name and possibly its parameters or instance number (for recursive calls).

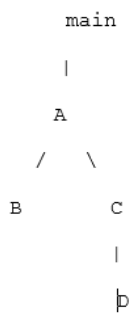
Edges: An edge from a parent node to a child node indicates that the parent procedure called the child procedure.

Leaves: Nodes representing activations that do not call other procedures (e.g., base cases in recursion or leaf procedures).

Order: The tree reflects the order of procedure calls, with children of a node representing sequential or nested calls made by the parent procedure.

Example :

```
void main() {
   A();
 }
 void A() {
   B();
   C();
 }
 void B() { }
 void C() {
   D();
 }
```



Purpose:

- Visualizes the dynamic execution of a program, showing how procedures are invoked and how control flows between them.
- Helps in understanding the runtime stack's behavior, as the activation tree corresponds to the sequence of activation records pushed and popped.
- Assists in analyzing recursive calls and nested procedures, making it easier to track the lifetime of each activation

4. Explain Static Storage Allocation Technique

Static storage allocation is a memory management strategy in which memory for all variables is allocated at compile time. Once allocated, the memory addresses for these variables remain fixed throughout program execution. This technique is typically used for global variables, static variables, and functions, which do not require dynamic allocation or recursion.

Key Characteristics of Static Storage Allocation:

1. **Compile-Time Allocation:**
    - The memory for variables is assigned during **compilation**, not during execution.
2. **Fixed Memory Location:**
    - Every variable is bound to a fixed memory address that does **not change** while the program runs.

3. **Efficient Access:**
    - Since addresses are known at compile time, access to variables is **very fast**.
4. **No Recursion Support:**
    - Recursive functions cannot be handled using static storage allocation because **each recursive call requires a new set of local variables**.
5. **Lifetime = Entire Program Duration:**
    - Memory stays allocated **for the whole life of the program**, even if the variable is only used briefly.

Memory Layout:

- The data segment stores initialized global and static variables (e.g., int x = 10;).
- The bss segment stores uninitialized global and static variables (e.g., int y;), which are initialized to zero by the operating system.
- The code segment stores the program's executable instructions.

How It Works:

- The compiler maintains a **symbol table** where it stores each variable's: Name, Type and Offset (or address)
- During code generation, actual memory addresses are assigned based on this table.

Advantages:

- Very fast access to memory since locations are known.
- No overhead for memory allocation/deallocation at runtime.
- Simple to implement.

Disadvantages:

- Not flexible – memory for unused variables is still allocated.
- Can't handle recursive functions or dynamic data structures like linked lists or trees.

5. Explain Activation Record Organization

An activation record (also known as a stack frame) is a block of memory created at runtime to store information necessary for managing a single execution of a function or procedure.

The activation record organization refers to the layout and structure of this block, including how various components such as parameters, local variables, return addresses, and temporary data are stored and accessed during function calls and returns.

Purpose:

- Manages the procedure's execution context, including local variables, parameters, and control flow.
- Supports recursion by creating a new activation record for each call, preserving the state of previous activations.
- Facilitates stack-based memory management, where records are pushed during calls and popped upon returns in a last-in, first-out (LIFO) manner.

Typical Fields in an Activation Record:

Here is the typical structure and organization of an activation record from top to bottom (as seen on a stack):

1. Temporary Variables:
    - Store intermediate values used during expression evaluation.
    - Compiler-generated; not visible in source code.
2. Local Variables:
    - Variables declared within the function.
    - Stored here for the duration of the function call.
3. Saved Machine Status:
    - Stores registers or other machine-specific information that needs to be restored after the function returns.
4. Access Link (Optional):
    - Points to an activation record of a function in the static nesting chain.
    - Used in nested or block-structured languages (like Pascal).
5. Control Link (Dynamic Link):
    - Points to the activation record of the caller function.
    - Helps restore the calling environment.
6. Return Address:
    - Tells where to return after the current function finishes.
7. Actual Parameters:
    - Stores the arguments passed to the function.

Activation Record Layout (Diagrammatically)

```
|----------------------| <-- Top of the Stack
| Temporary Variables  |
|----------------------|
| Local Variables      |
|----------------------|
| Saved Machine Status |
|----------------------|
| Access Link (optional) |
|----------------------|
| Control Link         |
|----------------------|
| Return Address       |
|----------------------|
| Actual Parameters    |
|----------------------| <-- Bottom of the Activation Record
```

Working Example:

Consider the following C code:

```
int sum(int a, int b) {
   int result;
   result = a + b;
   return result;
}

int main() {
   int x = sum(5, 10);
}
```

Activation Record for sum:

- Actual Parameters: a = 5, b = 10
- Return Address: Where to return after sum completes (inside main)
- Control Link: Points to activation record of main
- Local Variable: result
- Temporaries: If needed for intermediate expressions

Importance of Activation Record Organization:

1. **Supports Function Calls**:
    - Organizes runtime information for call and return.
2. **Manages Local Data**:
    - Ensures local variables are not overwritten between calls.
3. **Handles Recursion and Nesting**:
    - Each call maintains its context, allowing safe recursive calls.
4. **Efficient Execution**:
    - Stack-based structure allows for fast push/pop operations.

7. Dynamic Storage Allocation Techniques

Introduction:

In programming languages, **dynamic storage allocation** refers to the process of assigning memory to variables **during runtime** rather than compile time. This allows programs to handle **variable-sized data**, **recursive calls**, and **dynamic data structures** like linked lists, trees, etc.

There are **two main techniques** used for dynamic storage allocation:

1. **Explicit Allocation**
2. **Implicit Allocation**

1. Explicit Allocation

Definition:

In **explicit allocation**, the **programmer manually requests and releases memory**. This gives them direct control over when and how memory is managed.

This technique is used in languages like **C** and **C++**.

Key Features:

- Programmer uses **library functions** like malloc(), calloc(), free() in C, or new and delete in C++.
- The **burden of memory management** is on the programmer.
- Memory leaks or dangling pointers can occur if not managed properly.

Example in C:

```
int *ptr = (int*) malloc(sizeof(int) * 10); // Allocates memory for 10 integers
```

```
// Use the memory
ptr[0] = 5;


// Free the memory when done
free(ptr);
```

Advantages:

- More control and flexibility.
- Efficient in terms of speed when used correctly.

Disadvantages:

- Risk of memory leaks if free() is not called.
- Complex and error-prone.
- Bugs like dangling pointers and double frees can occur.

2. Implicit Allocation

Definition:

In **implicit allocation**, the **language runtime system automatically manages memory allocation and deallocation**. The programmer **does not explicitly allocate or free memory**.

Used in **high-level languages** like **Java, Python, Haskell, JavaScript**, etc.

Key Features:

- Memory is allocated as needed when variables are created or objects are instantiated.
- Memory is freed automatically by the **garbage collector**.
- Easier and safer for the programmer.

```
String name = new String("Alice");  // Memory is allocated
automatically
name = null;  // Memory will be garbage collected later
```

Advantages:

- Reduces programmer errors related to memory (e.g., no manual free() needed).
- Improves code readability and maintainability.
- Automatic garbage collection helps avoid memory leaks.

Disadvantages:

- Less control over performance and memory usage.
- Garbage collection may cause **unpredictable pauses** in execution.
- More runtime overhead compared to explicit allocation.

Example: Heap allocation is a dynamic storage allocation technique where memory is allocated and deallocated explicitly during runtime from a memory pool called the heap. The heap is a region of

memory used for objects with arbitrary lifetimes, such as dynamically sized arrays, objects in object-oriented languages, or data structures like linked lists and trees. Unlike stack allocation, heap memory is not tied to the LIFO structure and requires manual or automatic memory management

8. Discuss how Activation Record are used to access local and global variables

Accessing Local Variables Using Activation Records:

Local variables are variables that are declared within a function and only exist during the function's execution. They are **stored in the activation record** created when the function is called.

Access Process:

1. When a function is invoked, a new **activation record** is pushed onto the stack.
2. The **local variables** are placed inside the activation record, typically **above** the return address and arguments.
3. The **stack pointer** keeps track of the current position in memory, pointing to the most recently pushed activation record.
4. During execution, accessing a local variable is straightforward because it's **within the current activation record**, and the stack pointer provides direct access.

Example:

```
void func() {
    int localVar = 5;   // Local variable
    // localVar is stored in the activation record for 'func'
}
```

- The activation record for func() will contain localVar, and its address can be accessed by using the **offset** from the **stack pointer**.

Accessing Global Variables Using Activation Records:

Global variables are variables that are declared outside of any function and are **accessible throughout the program**. Since global variables are stored in **global memory**, they are not part of the activation record. Instead, they are referenced using their **global memory address**.

Access Process:

- **Global variables** are typically accessed directly from **global memory**, which is separate from the function stack.
- Since global variables have a **fixed memory address**, they can be accessed directly, irrespective of the **activation record** or the function call stack.
- When a function needs to access a global variable, the **activation record** does not need to store the variable. The **global memory** is searched to retrieve the value of the global variable.

Example:

```
int globalVar = 10;  // Global variable
```

```
void func() {
    globalVar = 20;  // Accesses global variable
}
```

- In this case, globalVar is stored in **global memory** and can be accessed directly by any function.
- **No special reference** is needed in the activation record to access globalVar as it is **globally accessible**.

9.Write a short note on Stack allocation.

Stack Allocation

Stack allocation is a dynamic storage allocation technique used in compiler design to manage memory for procedure calls by organizing it in a last-in, first-out (LIFO) structure called the runtime stack. It is primarily used to allocate activation records (stack frames) that store local variables, parameters, return addresses, and control information for each procedure invocation

Mechanism:

- Allocation: When a procedure is called, the compiler generates code to push an activation record onto the stack. The stack pointer (SP) is incremented to reserve space for local variables, parameters, return address, control link (pointer to the caller's record), and other fields. The frame pointer (FP) is set to access fields at fixed offsets.
- Deallocation: Upon procedure return, the activation record is popped by decrementing the stack pointer, restoring the caller's state using the control link, and branching to the return address.
- Management: The stack grows downward (toward lower addresses), and the compiler calculates the record's size at compile time, handling variable-length data via pointers.

Advantages:

- Efficiency: Fast push/pop operations via simple pointer adjustments.
- Automatic Memory Management: Memory is reclaimed automatically upon procedure return.
- Recursion Support: Each recursive call gets a new activation record, preserving prior states.
- Locality: Local variables are stored close to the procedure context, improving cache performance.

Disadvantages:

- Limited Stack Size: Deep recursion or large local variables can cause stack overflow.
- Short Lifetime: Variables are deallocated upon return, unsuitable for persistent data.
- Dangling References: Returning pointers to local variables can cause errors, as memory is freed.

Use Case: Ideal for local variables and activation records in languages like C, C++, and Java, especially for recursive procedures.

10. How is Task Divided Between Calling and Called Program for Stack Updating? In stack-based allocation, when a function (called program or callee) is invoked by another function (calling program or caller), both participate in creating and updating the stack. This is done through what's called the calling sequence and the return sequence.

Tasks of the Calling Program (Caller):

1. **Evaluate Actual Parameters**:
   The caller evaluates the actual arguments to be passed to the called function, which may involve calculating values or determining the memory addresses of variables.

2. **Create Space for Activation Record**:
   The caller allocates space on the stack for the callee's activation record, which holds information such as local variables, return address, and other necessary data.

3. **Pass Parameters**:
   The caller stores the evaluated actual parameters (either by value or by reference) into the allocated space in the activation record, preparing them to be accessed by the callee.

4. **Save Return Address**:
   The caller stores the address of the instruction that follows the function call in the activation record, so the callee knows where to return once it finishes executing.

5. **Control Transfer**:
   The caller transfers control to the callee by using a jump or call instruction, effectively handing over execution to the called function.

Tasks of the Called Program (Callee):

1. **Save Machine State**:
   The callee saves important registers such as the base pointer and return address into its activation record to preserve the state of the machine before it starts execution.

2. **Set Up Environment**:
   The callee sets up the required memory space for its local variables, temporaries, and other environment setup needed for the function to execute correctly.

3. **Initialize Local Variables**:
   The callee initializes any local variables declared within the function, ensuring that they have valid values before execution proceeds.

4. **Begin Execution**:
   The callee starts executing the function's code, using the passed parameters and working within the environment that has been set up.