C++ Concepts and Stack Implementations

# Stack Implementation Using Array

class StackArray {  
 int top;  
 int arr[100]; // Fixed size array  
public:  
 StackArray() { top = -1; }  
   
 void push(int x) {  
 if (top >= 99) {  
 std::cout << "Stack Overflow\n";  
 } else {  
 arr[++top] = x;  
 }  
 }  
  
 void pop() {  
 if (top == -1) {  
 std::cout << "Stack Underflow\n";  
 } else {  
 top--;  
 }  
 }  
  
 int peek() {  
 if (top != -1) return arr[top];  
 return -1; // Empty stack  
 }  
};

# Stack Implementation Using Dynamic Array

class StackDynamicArray {  
 int top;  
 int \*arr;  
 int capacity;  
public:  
 StackDynamicArray(int size = 10) {  
 top = -1;  
 capacity = size;  
 arr = new int[capacity];  
 }  
  
 void push(int x) {  
 if (top >= capacity - 1) {  
 // Resize the array dynamically  
 int \*newArr = new int[2 \* capacity];  
 for (int i = 0; i < capacity; i++) newArr[i] = arr[i];  
 delete[] arr;  
 arr = newArr;  
 capacity \*= 2;  
 }  
 arr[++top] = x;  
 }  
  
 void pop() {  
 if (top == -1) {  
 std::cout << "Stack Underflow\n";  
 } else {  
 top--;  
 }  
 }  
  
 int peek() {  
 if (top != -1) return arr[top];  
 return -1;  
 }  
};

# Stack Implementation Using Linked List

class StackLinkedList {  
 struct Node {  
 int data;  
 Node\* next;  
 Node(int d) : data(d), next(nullptr) {}  
 };  
 Node\* top;  
public:  
 StackLinkedList() { top = nullptr; }  
  
 void push(int x) {  
 Node\* newNode = new Node(x);  
 newNode->next = top;  
 top = newNode;  
 }  
  
 void pop() {  
 if (top == nullptr) {  
 std::cout << "Stack Underflow\n";  
 } else {  
 Node\* temp = top;  
 top = top->next;  
 delete temp;  
 }  
 }  
  
 int peek() {  
 if (top != nullptr) return top->data;  
 return -1;  
 }  
};

# Static Variables in C++

Static variables are variables that retain their value between function calls.   
They are initialized only once, and the last stored value is carried forward to the next function call.  
  
Example:  
void staticExample() {  
 static int count = 0; // static variable  
 count++;  
 std::cout << count << "\n";  
}  
  
int main() {  
 staticExample(); // prints 1  
 staticExample(); // prints 2  
}

# Internal and External Fragmentation

- Internal Fragmentation: This occurs when fixed-size memory blocks are allocated to processes, but the process does not need the entire block, leaving unused space within the block. This leads to memory waste.  
- External Fragmentation: This occurs when free memory is scattered in small blocks, making it difficult to allocate large contiguous blocks to a process, even though total free memory may be sufficient.

# Memory Allocation for Arrays and Linked Lists

- Arrays: Memory for arrays is allocated contiguously in a single block, meaning that all elements of the array are stored in consecutive memory locations.  
 Example: int arr[5] allocates memory for 5 integers consecutively.  
- Linked Lists: Memory for linked lists is allocated dynamically, one node at a time. Each node is scattered in memory and contains a pointer to the next node.  
 Example:  
 struct Node {  
 int data;  
 Node\* next;  
 };  
 Node\* head = new Node();

# Loader, Linker, and Compiler

- Compiler: Translates source code written in a high-level programming language (like C++) into machine code or intermediate code.   
- Linker: Combines multiple object files (compiled code) into a single executable. It resolves addresses and links library functions to the program.  
- Loader: Loads the executable file into memory and prepares it for execution by the CPU, resolving any runtime dependencies or library links.