Question

What is a static variable in C?

• Answer: A static variable in C is a variable that retains its value between function calls. It can be declared both inside and outside functions.

How is a static variable different from an automatic (local) variable?

• Answer: Automatic variables (local variables) are created and destroyed every time a function is called, while static variables persist throughout the program's execution.

What is the default initial value of a static variable?

• Answer: Static variables are automatically initialized to zero if no explicit initialization is provided.

Can a static variable be accessed outside the function where it is declared?

• Answer: No, if a static variable is declared within a function, it is limited to that function's scope and cannot be accessed outside of it.

Explain the difference between a static variable and a global variable.

• Answer: Both static and global variables have a lifetime that extends throughout the program, but global variables can be accessed from any part of the program, while static variables are limited to the scope in which they are declared.

When should you use a static variable?

• Answer: Use static variables when you need a variable to retain its value between function calls or when you want to limit its scope to a specific function or file.

What is the significance of the static keyword in a global variable declaration?

• Answer: The static keyword in a global variable declaration limits the variable's scope to the file where it is declared, making it accessible only within that file.

Can a static variable be modified by other functions?

• Answer: If a static variable is declared within a function, it cannot be directly accessed or modified by other functions. Its scope is limited to the function in which it is defined.

How does the use of static variables affect the efficiency of a program?

• Answer: Static variables can improve program efficiency by maintaining state information between function calls, reducing the need for repeated initializations.

Provide an example of a situation where using a static variable is beneficial.

• Answer: A common example is using a static variable to count the number of times a function is called, retaining the count between calls.

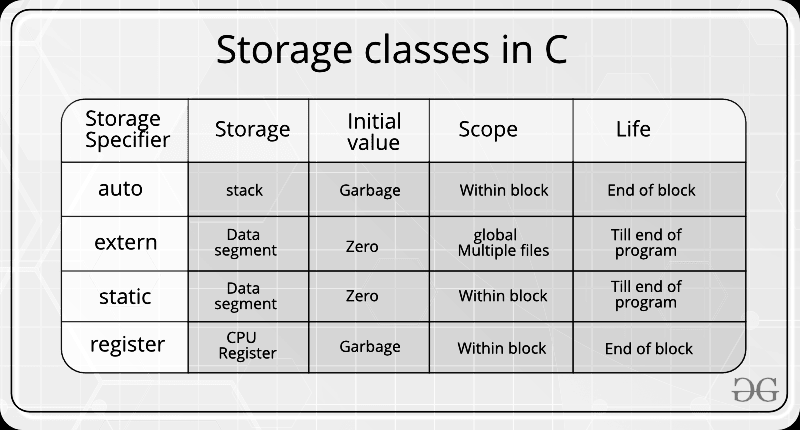
* How do you manage memory manually in C or C++? How do you avoid memory leaks and dangling pointers?
* How do you use pointers and references in C++ effectively? In what scenarios would you prefer references over pointers?
* How do you handle concurrency in C++? Can you explain how std::thread or other concurrency primitives like mutexes or condition variables are used?
* How do you implement polymorphism in C++? How does virtual function dispatch work under the hood?
* How do you manage object lifecycles in C++ (constructors/destructors)? What are some cases where copy constructors or assignment operators need to be custom-defined?
* How do you write and use templates in C++? Can you describe a use case for template metaprogramming or SFINAE?
* How do you use and manage dynamic memory using STL containers? What are the internal memory management differences between std::vector, std::list, and std::map?
* How do you debug segmentation faults or undefined behavior in C/C++ code? What tools do you typically use (e.g., valgrind, gdb)

Basics C/C++

1. **Storage Class in C**

It used to determine the lifetime, visibility, memory location, and initial value of a variable.

We can not use more than one storage specifier for the same variable.

****

1. auto :

Every local variable is automatic in C by default.

Any variable inside a function without any storage class specifier is by default auto.

1. extern :

Variables defined as extern are declared with an external linkage elsewhere in the program.

The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

1. static :

Initialized only once and instructs the compiler to keep the variable in existence till the termination of the program and retain its value between function calls

By Default initialized to zero.

static specifiers can hold their value between the multiple function calls.

1. resister :

Used to define local variables which require quick access such as counters.

It’s a request to the compiler

We can not dereference the register variables, i.e., we can not use &operator for the register variable.

1. **Inline Function**

Inline functions used to reduce the function call overhead.

When the inline function is called, the whole code of the inline function gets substituted at the point of the inline function call. it’s performed by the C++ compiler at compile time.

It’s a request to the compiler. The compiler can ignore the request for inlining if the function contains Loop, Static Variables, Recursion, switch case or goto calls.

All the functions defined inside the class are implicitly inline.

1. **typedef and #define**
2. typedef:

typedef keyword in C++ is used for aliasing existing data types, user-defined data types, and pointers.

1. #define:

#define in C is a directive which is used to define aliases.

typedef is limited to giving symbolic names to types only,

whereas #define can be used to define an alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.

typedef interpretation is performed by the compiler

whereas #define statement is performed by preprocessor.

typedef should be terminated with a semicolon

#define should not be terminated with a semicolon

typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there.

whereas In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

1. **Endianness**

It refers to the order in which bytes are arranged in memory.

It must be considered in various computing scenarios, particularly when systems with different byte orders need to communicate or share data.

1. Big-endian (BE):

Most significant byte (MSB) is stored at the lowest memory address.

Address: 00 01 02 03

Data: 12 34 56 78

1. Little-endian (LE): used by INTEL and ARM processors.

Least significant byte (LSB) is stored at the lowest memory address.

Address: 00 01 02 03

Data: 78 56 34 12

1. **Preprocessors in C**

Preprocessors are programs that process the source code before compilation.

1. #define : Used to define a macro

2. #undef : Used to undefine a macro

3. #include : Used to include a file in the source code program.

4. #ifdef : Used to include a section of code if a certain macro is defined by #define

5. #ifndef : Used to include a section of code if a certain macro is not defined by #define

6. #if : Check for the specified condition

7. #else : Alternate code that executes when #if fails

8. #elif : Combines else and if for another condition check

9. #endif : Used to mark the end of #if, #ifdef, and #ifndef

1. **Structure and Union**
2. Structure:

It can be defined as a user-defined data type which groups logically related items under one single unit. We can use all the different data items by accessing that single unit.

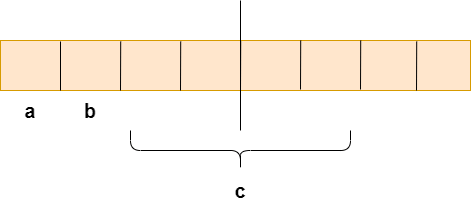
All the data items are stored in contiguous memory locations.

b. Union:

Union allows storing different data types in the same memory location.It provides an efficient way of using the same memory location for multiple purposes.

Structure Padding:

Reason : The processor does not read 1 byte at a time. It reads 1 word at a time.

Eg. 32-bit processor reads 4 bytes at a time, a 64-bit processor reads 8 bytes at a time.

struct student

{

char a; // 1 byte

char b; // 1 byte

int c; // 4 bytes

}

Problem : in one CPU cycle, one byte of char a, one byte of char b, and 2 bytes of int c can be accessed. will not face any problem while accessing the char a and char b as both the variables can be accessed in one CPU cycle.

But to access the int c variable 2 CPU cycles are required.

This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles.

The structure padding is done automatically by the compiler.

1. **NULL, Void, Wild and Dangling Pointer**

a. NULL Pointer:

Null pointer that points to nothing, when we don’t have a specific address to assign to a pointer use NULL to indicate that it’s not pointing to anything.

int \*ptr = NULL;

b. Void Pointer:

pointer that points to an unspecified data location in storage. Basically, the data that it points to can be anything.

It is not possible to dereference void pointers. However, it is possible to do so by typecasting the void pointer.

Due to the lack of a concrete value and thus size, pointer arithmetic is not possible on void pointers.

c. Wild Pointer:

A wild pointer is one that has not been initialized to anything (not even NULL). The pointer may be set to a non-NULL garbage value that is not a valid address.

int \*ptr;

d. Dangling Pointer:

A dangling pointer is a pointer that points to a deleted (or freed) memory location.

int \*ptr;

ptr = NULL; // Set to NULL to avoid dangling pointers

1. **Smart Pointers**

(*std::unique\_ptr*, *std::shared\_ptr*) introduced in C++11 to automatically manage memory. They deallocate memory when the pointer goes out of scope.

**std::unique\_ptr**:

Manages a single owner of the resource. The MyClass object is automatically deleted when uniquePtr goes out of scope.

Created with std::make\_unique (C++14+), which is safer and more concise.

Cannot be copied (ensures unique ownership) but can be moved.

**std::shared\_ptr**:

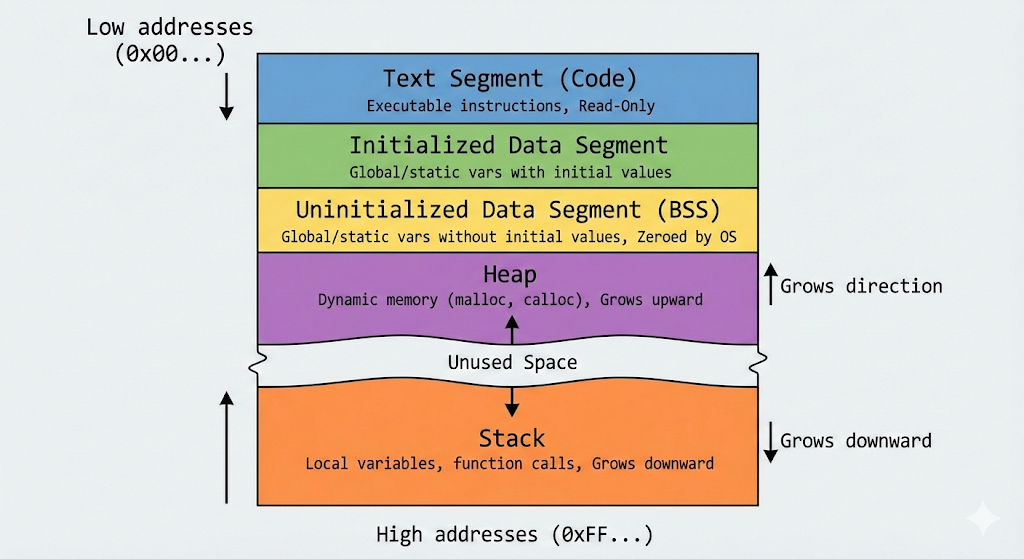
Allows multiple pointers to share ownership of the same resource. The resource is deleted only when the last shared\_ptr is destroyed.

Created with std::make\_shared (C++14+), which is efficient.

Tracks the number of references with use\_count().

1. **Memory**

Memory is smaller addressable units called bytes and divided in different sections.



**Compiler** → decides how much space each static variable needs

**Linker** → places them in Data/BSS segments

**Loader (OS)** → reserves that memory when program starts

**Compiler steps:**

1. Computes the size of global/static variables
2. Reserves addresses in Data/BSS segments
3. Generates symbol table (e.g., global\_var → addr 0x601040)

### **1. Text Segment (Code Segment) :** Whenever the program is executed it will be brought into the main memory. This program will get stored under the code section. Based upon the program it will decide whether to utilize the stack or heap sections. It’s often a read-only memory.

* **Location:** Lowest addresses.
* **Contents:** Contains the compiled binary instructions (machine code) of your program.
* **Characteristics:**
  + **Read-Only:** This area is marked read-only to prevent the program from accidentally modifying its own instructions.
  + **Sharable:** If you run multiple instances of the same program (e.g., three terminal windows), they often share this single Text segment to save memory.

### **2. Initialized Data Segment (Data Segment) :** A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. Initialized Data Memory: static int i = 10 and global int j = 10 will be stored in this segment.

* **Location:** Above the Text segment.
* **Contents:** Stores **global** and **static** variables that are explicitly initialized by the programmer to a non-zero value.
* **Lifetime:** Variables here exist for the entire duration of the program.

### **3. Uninitialized Data Segment (BSS) :** Uninitialized Data Memory: Data in this segment is initialized by the compiler to arithmetic 0 before the program starts executing uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. static int i; and global int j; will be stored in this segment.

* **Location:** Above Initialized Data.
* **Name Origin:** "Block Started by Symbol" (an old assembler pseudo-op).
* **Contents:** Stores **global** and **static** variables that are *not* explicitly initialized or are initialized to zero.
* **Key Efficiency:** Unlike the Data Segment, the BSS does not take up actual space in the executable file on your disk. The OS simply notes how much memory is needed and initializes this entire block to zero when the program starts.

### **4. Heap :** It is a part of the main memory. It is unorganized and treated as a resource when you require the use of it if not released. Heap memory can’t be used directly with the help of a pointer.

* **Location:** Above the BSS.
* **Contents:** Memory dynamically allocated by the programmer during runtime using malloc(), calloc(), realloc(), or new (in C++).
* **Direction:** Grows **upward** (towards higher addresses).
* **Management:** This memory must be managed manually. If you allocate it, you must free() it, otherwise, you get memory leaks. It is shared by all threads in a process.

### **5. Stack :** It stores temporary variables created by a function. In stack, variables are declared, stored, and initialized during runtime. It follows the First in last out method that means whatever element is going to store last is going to delete first when it’s not in use.

* **Location:** Highest addresses (starts high and grows down).
* **Contents:** Used for function execution control. It stores:
  + **Local variables:** Variables declared inside functions (automatic variables).
  + **Function parameters:** Arguments passed to functions.
  + **Return addresses:** Where the code should return after a function finishes.
* **Direction:** Grows **downward** (towards lower addresses/Heap).
* **Management:** Automatic. When a function is called, a "stack frame" is pushed; when it returns, the frame is popped (deleted).

1. **Static and Dynamic Memory**

a. Static Memory Allocation:

It involves reserving a fixed amount of memory for a variable at the time of program compilation. The memory is allocated at compile time and remains fixed throughout the life of the program. It is commonly used for global variables, static variables, and arrays.

**Key Features:**

Allocation and deallocation are done by the compiler.

It uses stack data structure for static memory allocation.

Variables get allocated permanently.

Execution is faster than dynamic memory allocation.

Memory is allocated before runtime.

It is less efficient.

**Disadvantage:**

Memory wastage problem.

Exact memory requirements must be known.

Memory can’t be resized once after initialization.

b. Dynamic Memory Allocation:

Memory is allocated and deallocated as required during the program execution. It is commonly used for creating data structures such as linked lists, trees, and dynamic arrays.

It is managed and served with pointers that point to the newly allocated memory space in the heap memory area. Heap memory is unorganized and it is treated as a resource when you require the use of it if not release it.

**Key Features:**

We can also reallocate memory size if needed.

Dynamic Allocation is done at run time.

No memory wastage.

**Disadvantage:**

Slower Access: Accessing dynamic memory is slower compared to static memory because the memory address is not known at compile time.

Memory Leaks: Dynamic memory allocation can result in memory leaks if memory is not deallocated properly. This can cause the program to crash or slow down.

Fragmentation: Dynamic memory allocation can result in memory fragmentation if the memory is not allocated and deallocated properly. Memory fragmentation occurs when there are small unused gaps between allocated memory blocks.

1. **Malloc, Calloc, Realloc and free**

Dynamic memory allocation in c language is possible by 4 functions of the stdlib.h header file.

a. Malloc:

allocates a single block of requested memory and if memory is allocated it returns the address of the first byte of memory otherwise NULL if memory is not sufficient.

it has garbage value initially.

*int \*ptr = (int\*)malloc(No of values\*size(int));*

b. Calloc:

allocates multiple blocks of memory of a given size and if memory is allocated it returns the address of the first byte of memory otherwise NULL if memory is not sufficient.

Initialized with all the bytes as zero.

*int \*ptr = (int\*)calloc(Number of data items, sizeof(int));*

c. Realloc:

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by this function.

*int \*nptr = (typecast) realloc (ptr, new number of elements \* sizeof(int));*

d. Free:

memory occupied by malloc() or calloc() functions must be released by calling free() functions.

heap memory will be automatically released when the program ends but is always better to explicitly release using *free(ptr).*

A memory leak occurs when memory is allocated dynamically and reference to it is not retained, due to which it is unable to release the memory.

1. **Memory Leak**

A memory leak occurs when a dynamically allocated memory is not freed.

When the track of pointers that references the allocated memory is lost then it may happen that memory is not freed up.

To avoid memory leaks, memory allocated on the heap should always be freed when no longer needed.

1. **Typecasting :**

Process of converting one data type to another data type by using the casting operator.

a. Implicit Typecasting:

It doesn't require any special syntax because the compiler takes care of it.

b. Explicit Typecasting:

Using the cast operator explicitly changing one data type to another. It needs explicit instructions in the code.

1. **Basics :**

Four pillars of OOPs: **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**

a. Encapsulation :

Encapsulation bundles data and methods into a class, restricting direct access to some components (using private or protected) to ensure data integrity and controlled access via public methods.

b. Inheritance :

Inheritance allows a derived (child) class to inherit attributes and methods from a base (parent) class, enabling code reuse and hierarchical relationships.

c. Polymorphism :

Polymorphism allows objects of different classes to be treated as objects of a common base class, typically through virtual functions and method overriding, enabling flexible behavior.

Compile-Time Polymorphism (Static Polymorphism) : Operator or Function overloading with the same name but different parameters (number, types, or order).

Run-Time Polymorphism (Dynamic Polymorphism) : It is achieved through function overriding and virtual functions, typically using a base class pointer or reference to call derived class methods.

Virtual Functions: Functions declared with the virtual keyword in the base class, overridden in derived classes.

Dynamic Dispatch: The decision about which function to call is made at runtime using a virtual table (vtable).

d. Abstraction :

Abstraction hides implementation details and exposes only essential functionality, often through abstract classes with pure virtual functions that derived classes must implement.

1. **Abstract Class and Pure Virtual Function**

Abstract class is a class that cannot be instantiated directly and is designed to serve as a base class for derived classes.

An abstract class contains at least one pure virtual function, which is declared with = 0 in its declaration. It forces the derived classes to implement the function.

Derived classes must implement all pure virtual functions to become concrete (non-abstract) classes, or they remain abstract.

Abstract classes can have regular member functions, data members, and constructors.

It is used to define a common interface or blueprint for a group of related classes, enforcing that derived classes implement specific methods.

Abstract classes enable runtime polymorphism through virtual functions and pointers/references to base class types.

The compiler creates a hidden **vtable (virtual table)** for each class with virtual functions. It holds pointers to virtual functions. Exists once per class with virtual functions.

Each object of such a class contains a hidden pointer called **vptr (virtual pointer)**, which points to the class's vtable and helps in runtime resolution of virtual calls.

1. **Passing Function Parameters**

Function parameters can be passed in different ways: **call by object**, **call by pointer**, **call by value**, and **call by reference**

a. call by object :

passing an object of a class or struct to a function. This is similar to call by value for objects, where a copy of the object is passed to the function.

b. call by pointer :

a pointer to the variable’s memory address is passed to the function. Changes to the dereferenced pointer inside the function modify the original variable.

c. call by value :

a copy of the argument’s value is passed to the function. Changes to the parameter inside the function do not affect the original variable.

d. call by reference :

a reference (alias) to the original variable is passed to the function. Changes to the parameter directly modify the original variable. This is achieved using the & operator in the function parameter.

1. **Basics :**

a. Virtual Inheritance and Virtual Class : used to address the diamond problem in multiple inheritance scenarios. It ensures that a base class is inherited only once when a derived class inherits it through multiple paths, preventing ambiguity and duplicate base class instances.

A

/ \

B C

\ /

D

b. Scope Resolution Operator : used to define or access members (variables, functions, or nested types) of a specific class, namespace, or global scope.

**Accessing Global Variables or Functions**: To access a global variable or function when a local variable or function with the same name exists.

**Defining Class Members Outside the Class**: To define member functions or static members outside the class declaration.

**Accessing Namespace Members**: To access identifiers defined in a specific namespace.

**Resolving Inheritance Ambiguity**: To specify which base class’s member to use in multiple inheritance scenarios.

c. Templates : it enables generic programming, allowing you to write code that works with any data type. Templates are a key component of compile-time polymorphism

**Function Templates**: Functions that work with generic types.

**Class Templates**: Classes that work with generic types.

**Alias Templates** : Type aliases for templates.

d. Friend Functions : In C++ is a function that is not a member of a class but has access to the class's private and protected members. It is declared inside the class with the friend keyword, granting it special access privileges.

Friend functions are useful when you need an external function to manipulate or access a class’s internal data without being a member of that class.

e. Static Linking and Dynamic Linking :

**Static Linking**: involves combining all necessary library code into the executable file at compile time. The resulting executable is self-contained, including all dependencies.

* Happens during compilation.
* Increases executable size.
* No external dependencies at runtime.
* Faster execution (no runtime loading).
* Less flexible (updates to libraries require recompilation).

**Dynamic linking** loads libraries at runtime, using shared libraries (e.g., .dll on Windows, .so on Linux) that are separate from the executable.

* Happens at runtime.
* Smaller executable size.
* Libraries can be updated without recompiling the program.
* Slower startup due to runtime linking.
* Requires libraries to be present on the system.

1. **SOLID principles**

Set of five design principles in object-oriented programming (OOP) that promote maintainable, scalable, and robust software design.

**SRP**: One class, one responsibility.

**OCP**: Extend behavior without modifying existing code.

**LSP**: Subclasses should work seamlessly as their base class.

**ISP**: Use specific interfaces, not general ones.

**DIP**: Depend on abstractions, not concrete implementations.

**S - Single Responsibility Principle (SRP)**

A class should have only one reason to change, meaning it should have a single responsibility or purpose. This reduces coupling and improves maintainability.

Eg : class handling user data shouldn’t handle file I/O

**O - Open/Closed Principle (OCP)**

Classes should be open for extension but closed for modification.

Eg : You should extend a class’s behavior (e.g., via inheritance or polymorphism) without modifying its source code. This is often achieved using virtual functions or interfaces.

**L - Liskov Substitution Principle (LSP)**

Derived Class must be substitutable for their base types without altering the correctness of the program.

**I - Interface Segregation Principle (ISP)**

Clients should not be forced to depend on interfaces they don’t use.

**D - Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details.

| **Container** | **Underlying Structure** | **Key Characteristics** |
| --- | --- | --- |
| std::vector | Dynamic array | Fast random access,amortized O(1)  push\_back, slow middle insert/erase |
| std::deque | Chunked arrays  (block map) | Fast insert/remove at both ends,  random access slower than vector |
| std::list | Doubly-linked list | O(1) insert/erase anywhere,  no random access |
| std::forward\_list | Singly-linked list | Lightweight, forward-only iteration |
| std::array | Static array | Fixed size at compile time, stack-allocated |
| std::map | Red-Black Tree  (self-balancing BST) | Sorted unique keys, O(log n) operations |
| std::multimap | Red-Black Tree | Sorted keys, allows duplicates |
| std::set | Red-Black Tree | Sorted unique keys, no values |
| std::multiset | Red-Black Tree | Sorted keys with duplicates allowed |
| std::unordered\_map | Hash table | O(1) average-case access, unordered,  unique keys |
| std::unordered\_multimap | Hash table | Allows duplicate keys, unordered |
| std::unordered\_set | Hash table | Unordered unique keys |
| std::unordered\_multiset | Hash table | Unordered, allows duplicate keys |
| std::stack | Adaptor over deque or  vector | LIFO, only access top element |
| std::queue | Adaptor over deque | FIFO, access front/back only |
| std::priority\_queue | Binary heap  (on top of vector) | Max-heap by default, O(log n) insert/delete |

STL containers are **not thread-safe** by default for concurrent write operations.

1. **STL Containers**
2. **Vector (std::vector)**

A dynamic array that can resize itself. It provides fast random access.

1D Initialization: *vector<int> v;*

2D Initialization: *vector<vector<int>> v2(rows, vector<int>(cols, 0));*

3D Initialization: *vector<vector<vector<int>>> v3(x, vector<vector<int>>(y, vector<int>(z, 0)));*

1. **Deque (std::deque)**

Double-Ended Queue is a sequence container that allows fast insertion and deletion at both its beginning and its end.

1. **List (std::list)**

Doubly-linked list. Fast insertion/deletion at any point, but no random access like list[x].

1. **Array (std::array)**

It is fixed-size. Unlike std::vector, its size is determined at **compile-time**.

1. **Map (std::map)**

Sorted associative container that contains key-value pairs with unique keys.

1. **Multimap (std::multimap)**

Sorted associative container that contain key-value pairs like Map, but it can have duplicates elements the share a key, equal\_range is the most important tool for multimap.

*lower\_bound(k)*: Returns an iterator to the first element $\ge k$.

*upper\_bound(k)*: Returns an iterator to the first element $> k$.

*equal\_range(k)*: Effectively returns std::make\_pair(lower\_bound(k), upper\_bound(k))

1. **Set (std::set)**

Sorted (default ascending) associative container that stores unique elements. for (descending)

1. **Multiset (std::multiset)**

Sorted associative container that can store duplicate elements.

1. **Unordered\_map (std::unordered\_map)**

Unordered associative container that contains key-value pairs with unique keys

1. **Unordered\_multimap (std::unordered\_multimap)**

Unordered associative container that supports equivalent keys (multiple values for the same key)

1. **Unordered\_set (std::unordered\_set)**

Unordered associative container that stores unique elements

1. **Unordered\_multiset (std::unordered\_multiset)**

Unordered associative container that can store duplicate elements

1. **Stack (std::stack)**

A container adapter that follows the LIFO (Last In, First Out) principle. It acts as a wrapper around an underlying sequence container (by default *std::deque*).

1. **Queue (std::queue)**

A container adapter that follows the FIFO (First In, First Out) principle. It acts as a wrapper around a sequence container (by default std::deque).

1. **Priority\_Queue (std::priority\_queue)**

A container adapter that provides constant time O(1) access to the "largest" (or highest priority) element. The element with the highest value is at the top. for (lowest value at top).

1. **Bitset (std::bitset)**

a special container designed to store and manipulate a fixed-size sequence of bits (0s and 1s). Unlike a *vector<bool>*, the size of a bitset must be known at compile-time.



1. **Pointer Output :**
2. Question-1

#include<stdio.h>

int main()

{

int x = 20, \*y, \*z;

// Assume address of x is 500 and

// integer is 4 byte size

y = &x;

z = y;

\*y++;

\*z++;

x++;

printf("x = %d, y = %d, z = %d \n", x, y, z);

return 0;

}

Output : x=21 y=504 z=504

the address of x is assigned to y and then y to z, it makes y and z similar. when the pointer variables are incremented their value is added with the size of the variable, in this case, y and z are incremented by 4.

1. Question-2

#include <stdio.h>

int main()

{

int a = 36;

int\* ptr;

ptr = &a;

printf("%u %u", \*&ptr, &\*ptr);

return 0;

}

Output : Address Address

& and \* canceled each other and display the address stored in a pointer variable ptr i.e) the address of a

1. Question-3

#include<stdio.h>

char \*getString()

{

char str[] = "Will I be printed?";

return str;

}

int main()

{

printf("%s", getString());

getchar();

}

Output : Garbage

array variables are stored in the Stack Section. So, when getString returns, values at str are deleted and str becomes a dangling pointer.

1. Question-4

#include<stdio.h>

char \*getString()

{

char str[] = "Will I be printed?";

return str;

}

int main()

{

printf("%s", getString());

getchar();

}

Output : Garbage

array variables are stored in the Stack Section. So, when getString returns, values at str are deleted and str becomes a dangling pointer.

1. **Linked List :**
2. Middle of Linked List

*int getMiddle(Node\* head) {*

*Node\* slow\_ptr = head;*

*Node\* fast\_ptr = head;*

*while (fast\_ptr != NULL && fast\_ptr->next != NULL) {*

*fast\_ptr = fast\_ptr->next->next;*

*slow\_ptr = slow\_ptr->next;*

*}*

*return slow\_ptr->data;*

*}*

1. Reverse the Linked List

*Node\* reverseList(Node\* head) {*

*Node \*curr = head, \*prev = nullptr, \*next;*

*while (curr != nullptr) {*

*next = curr->next;*

*curr->next = prev;*

*prev = curr;*

*curr = next;*

*}*

*return prev;*

*}*

c. Reverse the Linked List in k-group

*ListNode\* reverseKGroup(ListNode\* head, int k) {*

*ListNode\* temp = head;*

*ListNode\* prevLast = NULL;*

*while(temp != NULL){*

*ListNode\* kThNode = getkthnode(temp, k);*

*if(kThNode == NULL){*

*if(prevLast) {prevLast -> next = temp;}*

*break;*

*}*

*ListNode\* nextNode = kThNode -> next;*

*kThNode -> next = NULL;*

*reverse(temp);*

*if(temp == head){*

*head = kThNode;*

*}*

*else{*

*prevLast -> next = kThNode;*

*}*

*prevLast = temp;*

*temp = nextNode;*

*}*

*return head;*

*}*

1. Nth node from end

*int nthFromEnd(Node \*head, int N) {*

*Node\* main\_ptr = head;*

*Node\* ref\_ptr = head;*

*for (int i = 1; i < N; i++) {*

*ref\_ptr = ref\_ptr->next;*

*if (ref\_ptr == NULL) {*

*return -1;*

*}*

*}*

*while (ref\_ptr->next != NULL) {*

*ref\_ptr = ref\_ptr->next;*

*main\_ptr = main\_ptr->next;*

*}*

*return main\_ptr->data;*

*}*

1. Detect Loop in the Linked List

*int detectLoop(Node\* head) {*

*Node \*slow = head, \*fast = head;*

*while (slow && fast && fast->next) {*

*slow = slow->next;*

*fast = fast->next->next;*

*if (slow == fast) {*

*return 1;*

*}*

*}*

*return 0;*

*}*

1. Delete every Kth Node in the Linked List

*Node\* deleteK(Node\* head, int k) {*

*if (head == nullptr || k <= 0)*

*return head;*

*Node\* curr = head;*

*Node\* prev = nullptr;*

*int count = 0;*

*while (curr != nullptr) {*

*count++;*

*if (count % k == 0) {*

*if (prev != nullptr) {*

*prev->next = curr->next;*

*}*

*else {*

*head = curr->next;*

*}*

*}*

*else {*

*prev = curr;*

*}*

*curr = curr->next;*

*}*

*return head;*

*}*

1. Merge two sorted Linked List

*Node\* mergeSortedList(Node\* a, Node\* b) {*

*Node\* result = NULL;*

*if (a == NULL)*

*return (b);*

*else if (b == NULL) {return (a);}*

*if (a->data <= b->data) {*

*result = a;*

*result->next = mergeSortedList(a->next, b);*

*}*

*else {*

*result = b;*

*result->next = mergeSortedList(a, b->next);*

*}*

*return (result);*

*}*

ENUM: in C (enumerations) are user-defined types that consist of named integer constants. They are useful for representing a collection of related values with meaningful names, improving code readability and maintenance. You cannot declare the same enumerator names twice in the same scope, even inside different enums.

Format Specifiers in C

**Type Format Specifier Description**

int %d Decimal signed Integer

int %o Octal Integer

int %x Hexadecimal Integer (lower case)

int %X Hexadecimal Integer (upper case)

int %i Integer, octal or hexadecimal

int %u Unsigned Integer

int %h Short Integer

float %f Floating Point value without exponent

float %e Floating Point value with exponent

char %c Char

char %s String

double %lf Long float

long int %ld Long Integer

Timer in Cpp : using chrono library, it’s helpful to check how much time function or program took for execution

*#include <iostream>*

*#include <chrono> // Essential for timing operations*

*#include <algorithm> // Used for std::fill in the example*

*int main() {*

*// --- 1. Define the starting point ---*

*// Use high\_resolution\_clock for the most precise time measurement.*

*auto start = std::chrono::high\_resolution\_clock::now();*

*// --- 2. CODE BLOCK TO BE TIMED —--*

*const int SIZE = 1000000;*

*int\* array = new int[SIZE];*

*std::fill(array, array + SIZE, 42);*

*long long sum = 0;*

*for (int i = 0; i < SIZE; ++i) {*

*sum += array[i];*

*}*

*delete[] array;*

*// --- 3. Define the ending point ---*

*auto end = std::chrono::high\_resolution\_clock::now();*

*// --- 4. Calculate the duration ---*

*// std::chrono::duration calculates the difference between the two time points.*

*// We specify <double> to get the result in a floating-point format*

*// and <std::milli> to get the result in milliseconds.*

*std::chrono::duration<double, std::milli> duration\_ms = end - start;*

*// Alternatively, to get the time in seconds:*

*// std::chrono::duration<double> duration\_s = end - start;*

*// --- 5. Print the result ---*

*std::cout << "\n----------------------------------------\n";*

*std::cout << "\*\*Total Execution Time:\*\*\n";*

*std::cout << duration\_ms.count() << " milliseconds\n";*

*std::cout << "----------------------------------------\n";*

*return 0;*

*}*

File Handling in cpp : reading, writing and updating file

*#include <iostream> #include <fstream> #include <string> #include <sstream>*

*void updateFileContent(const std::string& filename) {*

*std::string content;*

// 1. Read the entire file content

*{*

*std::ifstream inputFile(filename);* //creates an input file stream object named inputFile and attempts to open the file specified by the filename string for reading.

*if (!inputFile.is\_open()) {std::cerr << "Error: Could not open \n”; return; }*

*std::stringstream buffer;* // creates an object of type std::stringstream named buffer. This object acts like a dynamic string buffer in memory to treat a string like an I/O stream

*buffer << inputFile.rdbuf();* //This is the core reading operation. inputFile.rdbuf() returns a pointer to the file stream's internal buffer object (streambuf). The stream extraction operator (<<) then efficiently copies all data from the file's buffer directly into the stringstream buffer until the end of the file is reached.

*content = buffer.str();*

*inputFile.close();*

*}*

*if (content.empty()) {std::cout << "File is empty.\n"; return;}*

// 2. Update "My" to "Our"

*size\_t pos\_my = content.find("My");* // gives the first occurrence of "My"

*if (pos\_my != std::string::npos) { content.replace(pos\_my, 2, "Our"); }*

*/*/ 3. Remove " name is "

*size\_t pos\_name\_is = content.find(" name is ");*

*if (pos\_name\_is != std::string::npos) {*

*content.erase(pos\_name\_is, 9); // Find " name is " and remove it. The length of " name is " is 9.*

*content.insert(pos\_name\_is, " ");* // can be used content.replace(pos\_name\_is, 9, " ");

*}*

// 4. Insert "ironman " before "Aditya"

*size\_t pos\_aditya = content.find("Aditya");*

*if (pos\_aditya != std::string::npos) {content.insert(pos\_aditya, "ironman "); }*

*else {std::cerr << "Error: 'Aditya' not found.\n"; return; }*

// 5. Append " Optimus" at the end

*content += " Optimus";*

// 6. Write the modified content back to the file

*{*

*std::ofstream outputFile(filename);* //line creates an output file stream object named outputFile. It attempts to open the file specified by filename for writing. By default, if the file already exists, it is truncated (all its previous content is deleted). If the file doesn't exist, it is created.

*if (!outputFile.is\_open()) {std::cerr << "Error: Could not open \n"; return; }*

*outputFile << content;*

*outputFile.close();*

*}*

*std::cout << "\nSuccess: File \"" << filename << "\" updated and saved.\n";*

*}*

*int main() {*

*const std::string filename = "file.txt";*

// Setup: Create the initial file with "My name is Aditya"

*{*

*std::ofstream setupFile(filename);*

*if (setupFile.is\_open()) {*

*setupFile << "My name is Aditya";*

*setupFile.close();*

*std::cout << "Setup: Created " << filename << " with initial content.\n";*

*} else {*

*std::cerr << "FATAL ERROR: Could not create initial file.\n";*

*return 1;*

*}*

*}*

*std::cout << "---------------------------------------\n";*

// ---Replace (My->Our) , Addition at the end (Optimus), Addition in between, before Aditya (ironman), Remove "name is"

*updateFileContent(filename);* //”Our ironman Aditya Optimus”

*return 0;*

*}*

**20. Code Optimization Techniques in cpp**

1. Speed up I/O (Input/Output):
   1. using *std::ios::sync\_with\_stdio(false);*

The standard C++ I/O streams (std::cin, std::cout, etc.) are by default synchronized with the standard C I/O functions (scanf,printf etc.).

**Default State (Synchronized)**: When I/O streams are synchronized, the C++ stream object and the C stream object can share buffers. This ensures that you can mix C and C++ I/O calls (e.g., using both cin and scanf) without worrying about the order of operations getting mixed up.

**Effect of** sync\_with\_stdio(false);: Setting this to false (or 0) disables this synchronization.

**Benefit:** C++ streams can now use faster, independent buffers, leading to a massive speed increase, especially when reading/writing large amounts of data.

**Caution:** After calling this, you should not mix C++ I/O (cin /cout) with C I/O (scanf/printf). Stick to using only cin and cout.

* 1. using *std::cin.tie(nullptr);*

By default, std::cin is "tied" to std::cout. This means that before a program attempts to read input using std::cin, it automatically calls std::cout.flush() to ensure any output is displayed immediately. This is a helpful feature for interactive programs (where the user needs to see a prompt before typing).

**Default State (Tied):** cin flushes cout before reading.

**Effect of** cin.tie(nullptr);**:** Setting this to nullptr (or 0) unties cin from cout.

**Benefit:** cin no longer forces a flush of cout. This prevents unnecessary flushing overhead, especially in programs where input and output operations are interleaved many times, further improving performance.

Code without Optimization

Code with Optimization

Bit Manipulation

& is Bitwise Operator and && is logical operator

1. **Compute XOR from 1 to n**

Input : n

Output : 1^2^3^4^........^n

*int computeXOR(int n)*

*{*

*if (n % 4 == 0)*

*return n;*

*if (n % 4 == 1)*

*return 1;*

*if (n % 4 == 2)*

*return n + 1;*

*else*

*return 0;*

*}*

1. **How to know if a number is a power of 2?**

Input : n

Output : true/false

*bool isPowerOfTwo(int n)*

*{*

*return n && (!(n & (n - 1)));*

*}*

1. **Find XOR of all subsets of a set**

The answer is always 0 if the given set has more than one element. For sets with a single element, the answer is the value of the single element.

1. **The Quickest way to swap two numbers:**

*a = a ^ b;*

*b = b ^ a;*

*a = a ^ b;*

1. **Position of the most significant set bit (MSB) Leftmost:**

Input: 10 (01010)

Output: 4

Explanation:Greatest number which is a Power of 2 less than 10 is 8

Binary representation of 10 is 1010

The most significant bit corresponds to position 4.

1. **Position of least significant set Bit (LSB) Rightmost**

Input : n = 20 (10100)

Output : 3

Approach : log of bitwise and 10100 & ~(10011) => 10100 & 01100 => 00100

log2(00100) + 1 => 3

*int clear\_bit(int n, int i)*

*{*

*return log2(n & ~(n-1)) + 1 ;*

*}*

1. **Multiply by 2 (Left shift)**

Input : 8

Output : 8 \* 2 \* 2 \* 2 = 64 1000 << 3 = 1000000

*int left\_shift(int n)*

*{ return (n << 3); }*

1. **Divide by 2 (Right shift)**

Input : 64

Output : (((64 / 2) / 2) / 2) = 8 1000000 >> 3 = 1000

*int right\_shift(int n)*

*{ return (n >> 3); }*

1. **Check iTH bit is set(1) or not**

Input : n = 21 (10101), i = 3

Output : True

Approach : masking by k = (1 << (i-1)) k = 100

*bool check\_setbit(int n, int i)*

*{*

*int k = (1 << (i-1));*

*return (n & k);*

*}*

1. **Clear the iTH bit of Number**

Input : n = 21 (10101), i = 3

Output : 17(10001)

Approach : bitwise and 10101 & ~(00100) => 10101 & 11011 => 10001

*int clear\_bit(int n, int i)*

*{*

*return (n & ~(1 << (i-1)); }*

1. **Remove the last set bit of Number(Rightmost)**

Input : n = 20 (10100)

Output : 16(10000)

Approach : bitwise and with (n-1) 10100 & (10011) => 10000

*int clear\_bit(int n)*

*{ return (n & (n-1)); }*

1. **Number is even or odd**

Input : n = 20 n = 21

Output : True False

1. Using Mod(%) operator:

*bool check\_even(int n)*

*{*

*if(n%2 == 0){return true;}*

*return false;*

*}*

1. Using Division(/) operator:

*bool check\_even(int n)*

*{*

*if((n/2)\*2 == n){return true;}*

*return false;*

*}*

1. Using Bitwise and(&) operator:

Approach : bitwise and with (1) 10100 & (00001) => 00000 (true)

10101 & (00001) => 00001 (false)

*bool check\_even(int n)*

*{*

*if((n & 1) == 0){return true;}*

*return false;*

*}*

1. Using shift(<< , >>) operator:

Approach : (10100 << 1) >> 1 => 101000 >> 1 => 10100 (true)

*bool check\_even(int n)*

*{*

*if(((n << 1) >> 1) == n){return true;}*

*return false;*

*}*

1. **Toggle iTH bit of a Number**

Input : n = 20 (10100) , i = 3 n = 17 (10001) , i = 4

Output : 16(10000) 25 (11001)

Approach : bitwise xor with (1 << (i-1)) 10100 ^(00100) => 10000

*int toggle\_bit(int n)*

*{ return (n ^ (1<<(i-1))); }*

1. **Count Number of set(1) bit (Brian Kernighan's algo)**

Input : n = 20 (10100) n = 19 (10011)

Output : 2 3

Approach : bitwise and with (n-1) until n becomes zero

*int count\_set\_bit(int n)*

*{*

*count = 0*

*while(n > 0){*

*n = n & (n - 1); # Clear the least significant set bit*

*count += 1; # Increment the count of set bits*

*}*

*return count;*

*}*

1. **Uppercase alphabet to Lowercase**

Input : c = A

Output : a

Approach : bitwise OR with space(‘ ‘)

*char lowercase(char c)*

*{ return c | ‘ ‘; }*

1. **Lowercase alphabet to Uppercase**

Input : c = a

Output : A

Approach : bitwise AND with underscore(\_)

*char uppercase(char c)*

*{ return c & ‘\_’; }*

1. **Invert alphabet from Uppercase to Lower or Lowercase to Upper**

Input : c = a c = A

Output : A a

Approach : bitwise XOR with space(‘ ‘)

*char uppercase(char c)*

*{ return c ^ ‘ ’; }*

1. **Find the element which occurs once in an array (remaining twice)**

Input : n = {1,2,3,4,3,4,2,1,5,8,5}

Output : 8

Approach : bitwise XOR of all elements

*int find\_single(vector<int>& arr)*

*{*

*int result = 0;*

*for (int num : arr) {*

*result ^= num; // XOR with each element*

*}*

*return result;*

*}*

1. **No. of bits need to flip for converting A to B**

Input : A = 37 (100101) B = 20 (010100)

Output : 3

Approach : bitwise XOR of A and B the count No. of set Bit using Brian Kernighan's algo

*int convert(int A, int B)*

*{*

*int val = A ^ B;*

*return count\_set\_bit(val);*

*}*

Puzzle

1. **Riddles:**

**You have a 5-liter jug and a 9-liter jug. Your mother asks you to use these two jugs to measure one liter of water. How would you do it?**

First fill the 5-liter jug and then pour all the water in the 9-liter jug.

refill the 5-liter jug and fill up the remaining portion of the 9-liter jug.

This way, I would have 1-liter water left in the 5-liter jug.

1. **Pay an employee using a gold rod of 7 units ?:**

**An employee works for an employer for 7 days. The employer has a gold rod of 7 units. How does the employer pay to the employee, so that the number of employee’s rod units increases by one at the end of each day? The employer can make at most 2 cuts in the rod.**

The employer can pay for seven days by making 2 cuts in a way that he has 3 rods of size 1, 2 and 4.

1st Day: Employer gives 1 unit cut.

2nd day: Takes back 1 unit cut from the employee given on the first day and gives 2 unit cut.

3rd Day: Gives 1 unit and then the employer is left with 4 unit rod lengths.

4th Day: Takes back cuts of 1 and 2 units. Gives the cut of 4 units.

5th Day: Gives cut of 1 unit to the employee.

6th Day: Takes back cut of 1 unit and gives a cut of 2 units.

7th Day: Gives cut of 1 unit to the employee.

OS

### **Synchronization Mechanisms in Operating Systems**

Synchronization mechanisms in operating systems ensure safe and predictable interaction between multiple processes or threads, especially when they share resources. Here's a closer look at mutexes, semaphores, and monitors, with their differences and use cases.

### **1. Mutex (Mutual Exclusion Lock)**

* **Purpose**: Ensures exclusive access to a shared resource by one thread or process at a time.
* **Key Features:**
  + Binary state: Locked or Unlocked.
  + Only the thread/process that locks the mutex can unlock it.
  + Typically lightweight and fast.
* Use Case: Critical sections where only one thread should modify a shared resource.
* **Advantages**:
  + Simplicity and efficiency for mutual exclusion.
  + Prevents race conditions in shared resources.
* **Disadvantages**:
  + Deadlock risk if locks are not released properly.
  + No signaling mechanism (can't count threads).

### **2. Semaphore**

* **Purpose**: Used to manage access to a finite number of resources or to synchronize threads.
* **Key Features:**
  + - Can be binary (similar to a mutex) or counting (allows multiple threads).
    - Any thread can signal (V operation) and wait (P operation).
* **Operations**:
  + - Wait (P): Decreases the semaphore value. If the value is negative, the thread is blocked.
    - Signal (V): Increases the semaphore value. Unblocks waiting threads if value is ≤ 0.
* **Use Case:**
  + - Managing a pool of resources (e.g., thread pools, database connections).
    - Synchronizing producer-consumer problems.
* **Advantages**:
  + - Counting semaphores allow multiple threads to access resources up to a limit.
    - Can handle more complex synchronization.
* **Disadvantages**:
  + - Harder to implement correctly.
    - Risk of deadlocks and priority inversion.

### **3. Monitor**

* **Purpose**: High-level abstraction that encapsulates mutual exclusion and synchronization within a single construct.
* **Key Features:**
  + - Combines locks (mutexes) and condition variables.
    - Only one thread can execute within the monitor at a time.
    - Built-in mechanisms for condition signaling.
* **Use Case:** Structured synchronization in high-level languages like Java (e.g., synchronized methods).
* **Advantages**:
  + - Simplifies synchronization by integrating locking and signaling.
    - Reduces boilerplate code.
* **Disadvantages**:
  + - Less flexible than semaphores for certain problems.
    - Performance overhead due to abstraction.

**4. Spinlock**

* **Purpose:** A low-level synchronization mechanism where a waiting thread repeatedly checks (spins in a loop) to see if the lock is available, rather than sleeping.
* **Key Features:**
  + **Busy Waiting:** The waiting thread remains active on the CPU, consuming cycles while checking the lock status.
  + **No Context Switch:** Unlike a mutex, it avoids the operating system overhead of putting a thread to sleep and waking it up.
  + **Binary State:** Locked or Unlocked.
* **Use Case:**
  + Extremely short critical sections where the time to execute the code is shorter than the time it takes to switch threads.
  + Multi-core systems where one thread can work while another spins.
* **Advantages:**
  + **Lowest Latency:** Very fast to acquire if the lock is held for a very short time because there is no OS scheduling overhead.
  + **Simple:** Conceptually simple for short, non-blocking operations.
* **Disadvantages:**
  + **High CPU Waste:** If the lock is held for a long time, the waiting thread burns 100% CPU doing nothing useful.
  + **Priority Inversion:** A high-priority thread might spin forever waiting for a low-priority thread that has been preempted.
  + **Not Resource Friendly:** Can degrade overall system performance if not used carefully (e.g., without pause instructions).

### **4. Atomic (Atomic Operations)**

* **Purpose:** Allows read-modify-write operations on shared variables to happen as a single, indivisible hardware instruction, ensuring data consistency without using locks.
* **Key Features:**
  + **Indivisible:** The hardware guarantees the operation completes entirely or not at all; no other thread can interrupt it halfway.
  + **Lock-Free:** Does not use OS-level blocking or locking mechanisms.
  + **Memory Ordering:** Provides options for synchronization strength, from "relaxed" (no order guarantees) to "sequentially consistent" (total order).
* **Use Case:**
  + Simple counters (e.g., reference counting).
  + Status flags.
  + Building block for complex "lock-free" data structures (like queues).
* **Advantages:**
  + **Fastest Performance:** For simple counters, it is faster than mutexes or semaphores because it eliminates locking overhead.
  + **Deadlock Free:** Since there are no locks to acquire, a single atomic variable cannot cause a deadlock.
  + **Non-Blocking:** A thread is never put to sleep by the OS.
* **Disadvantages:**
  + **Complexity:** Difficult to implement correct logic for complex data structures (issues like the ABA problem).
  + **Limited Scope:** Can typically only protect a single variable (int or pointer), not a whole block of code.

### **Comparison: Mutex vs Semaphore vs Monitor**

| **Feature** | **Mutex** | **Semaphore** | **Monitor** | **SpinLock** | **Atomic** |
| --- | --- | --- | --- | --- | --- |
| Definition | Binary lock for mutual exclusion. | Counter for resource management or synchronization. | Abstraction with lock and signaling. | uses busy-waiting loops | Hardware instruction for single variables. |
| Threads Allowed | 1 thread at a time. | Multiple (depending on value). | 1 thread at a time inside. | 1 at a time. | N/A (Operation is instantaneous) |
| Waiting Behavior | Sleeps (Blocks). | Sleeps (Blocks). | Sleeps (Blocks). | Spins (Busy Wait) | **None** (Does not wait) |
| Performance | Fast (if low contention). | Moderate. | (Slower) High-level, structured. | **Fastest** (for short waits) | **Fastest** (for single ops). |
| Best Use Case | Complex critical sections | Resource pools / Producer-Consumer. | High-level thread synchronization. | Tiny critical sections. | Counters & Flags |

### 

### **Thread-Safe Counter Implementations**

* **Mutex:** Uses a lock (pthread\_mutex\_lock). Safest for complex logic but has overhead. (Mutual Exclusion) ensures that only one thread can execute the critical section (incrementing count) at a time.
* **Semaphore:** Uses a signaling mechanism (sem\_wait/sem\_post). This is functionally correct but computationally "overkill" for a simple counter.
* **Monitor:** Complex synchronization requiring signaling (waiting for a condition). C does not have a native "Monitor" keyword like Java. In C, a Monitor is built using a **struct** containing a Mutex (for locking) and a Condition Variable (for signaling) .
* **Spinlock:** Extremely short critical sections where you don't want the thread to sleep. *Note: Spinlocks keep the thread active on the CPU ("busy waiting") rather than blocking.*
* **Atomic:** Uses hardware instructions (atomic\_fetch\_add). This is the fastest method for simple counters. It uses hardware instructions (like LOCK XADD on x86) to perform the read-modify-write operation indivisibly, without the overhead of a full OS lock.

### **4. Process:** is an instance of a program in execution. **Processes** are isolated execution units that includes:

### **Code**: The program’s instructions.

### **Data**: Variables and memory allocated for the program.

### **State**: The current execution state (e.g., CPU registers, program counter).

### **Resources**: Memory, file handles, and other system resources.

### 

### Each process has its own **address space**, meaning its memory is isolated from other processes. Processes communicate via **inter-process communication (IPC)** mechanisms like pipes, message queues, or shared memory, which are typically slower due to the need for system calls.

### **5. Threads:** Threads are light-weight processes that can run concurrently within a single program. A thread is the smallest unit of execution within a process. A process can have one or more threads, and all threads within a process share the same memory and resources (e.g., global variables, file descriptors). Threads are lightweight compared to processes because they don’t require separate address spaces. but they are fragile; a bug (like a segfault) in one thread crashes the entire process.

### Threads are ideal for tasks that can be parallelized, like handling multiple user requests in a server.

**Single threaded Program:** One sequence of instruction executes at a time

**Multi threaded Program:** Multiple threads execute different parts of the program simultaneously.

* Efficiency: Thread uses less memory and resources then a full fledged process.
* Parallelism: Tasks like sorting big database and calculations which can be done faster using threads
* Responsiveness: In GUI applications, threads can keep the interface responsive while performing background tasks

**6. Process Control Block (PCB):** The PCB is a kernel data structure that contains all the information about a process required for its execution and management. When a process is created, the operating system allocates a PCB for it. The PCB is updated as the process transitions through states (e.g., ready, running, waiting, terminated).

**The PCB is critical for**:

Process Management: Storing process metadata (e.g., process ID, state).

Context Switching: Saving and restoring a process’s state when it’s preempted or resumed.

Scheduling: Providing information for the OS scheduler to prioritize processes.

A typical PCB contains the following information:

1. **Process ID (PID)**: A unique identifier for the process.
2. **Process State**: The current state of the process (e.g., New, Ready, Running, Waiting, Terminated).
3. **Program Counter**: The address of the next instruction to be executed.
4. **CPU Registers**: The values of CPU registers (e.g., general-purpose registers, stack pointer) to save the process’s context during context switching.
5. **CPU Scheduling Information**: Priority, scheduling queue pointers, or other scheduling parameters.
6. **Memory Management Information**: Details about the process’s memory allocation, such as page tables, base and limit registers, or segment information.
7. **Accounting Information**: CPU time used, elapsed time, or other resource usage metrics.
8. **I/O Status Information**: List of I/O devices allocated to the process, open files, etc.
9. **Parent/Child Process Information**: Links to parent process or child processes (if any).
10. **Other Resources**: File descriptors, network sockets, or other resources assigned to the process.

**Storage**: PCBs are stored in the kernel’s memory (protected from user access).

**Performance Impact**: Context switching involves saving and loading PCBs, which can be costly if many processes are running.

**Security**: PCBs ensure process isolation by tracking separate memory and resource allocations.

**Relation to Multithreading**: In a multithreaded process, the PCB tracks shared resources, while TCBs handle thread-specific execution states.

**7. Data Plane Development Kit (DPDK):**

**Kernel-Bypass Techniques for High-Speed Networking**

#### **1. The Problem with Traditional Linux Networking**

As network speeds have increased from 1 Gbps to 10 Gbps, 40 Gbps, 100 Gbps, and beyond, the traditional Linux kernel networking stack has become a significant bottleneck. The standard path for a network packet involves many steps that introduce latency and consume CPU cycles unrelated to the actual packet data processing.

**Key Bottlenecks in the Traditional Stack:**

* **Interrupt Storms:** For every incoming packet (or batch of packets), the Network Interface Card (NIC) generates a hardware interrupt to signal the CPU. The CPU must stop what it's doing, save its context, jump to the interrupt handler, process the packet, and then restore context. At high packet rates, this constant interruption, known as an "interrupt storm," overwhelms the CPU, leading to practically zero productive work.
* **Context Switching:** Switching between user space (where the application runs) and kernel space (where the network stack runs) is an expensive operation requiring the CPU to flush caches and TLBs.
* **Data Copying:** Packet data is often copied multiple times: from the NIC to kernel memory, from one kernel buffer to another, and finally from kernel space to the user-space application's buffer. Each copy wastes memory bandwidth and CPU cycles.
* **Complex Protocol Stack:** The general-purpose Linux network stack is feature-rich (iptables, routing, etc.), but this complexity adds overhead for applications that just want raw packet access.

#### **2. The Solution: Kernel Bypass Networking**

Kernel bypass is a technique that allows a user-space application to take direct control of the network hardware, completely side-stepping the operating system's kernel for data plane operations.

**How it Works:** The control path (device configuration, setup) may still involve the kernel, but the data path—sending and receiving actual network packets—is established directly between the application and the NIC. This eliminates interrupt handling, context switches, and in-kernel data copying for the critical fast path.

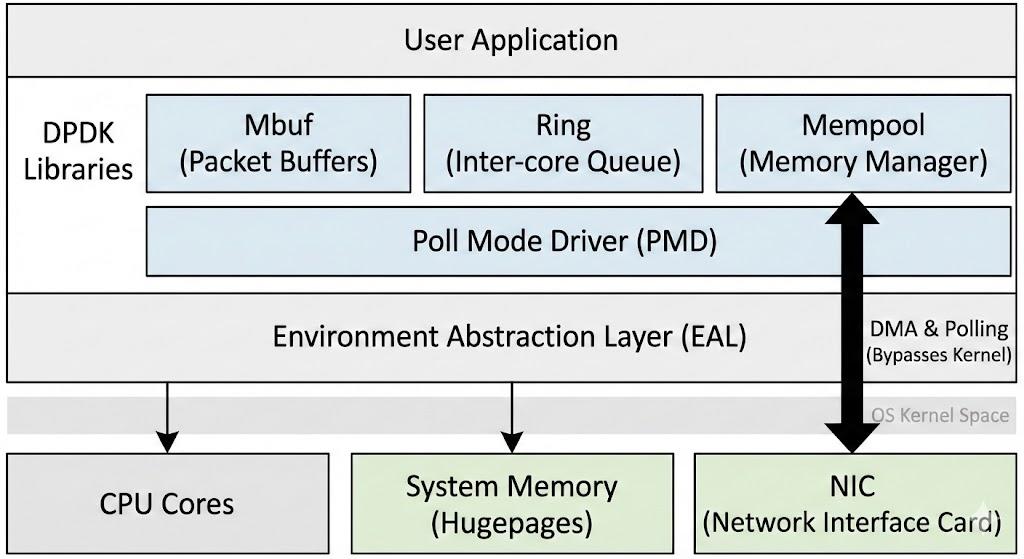
**Benefits:**

* **Ultra-Low Latency:** By removing kernel overhead, packet processing times are drastically reduced, often down to the microsecond level.
* **High Throughput:** The CPU can spend nearly all its cycles on packet processing, enabling line-rate performance even on high-speed links.
* **Deterministic Performance:** Eliminating the unpredictability of the kernel scheduler and interrupt handling leads to more consistent and reliable performance.

Below is a visual comparison between the traditional Linux network stack and a kernel-bypass architecture like DPDK.

### **3. Deep Dive into DPDK (Data Plane Development Kit)**

DPDK is not just a single tool but a comprehensive set of data plane libraries and network interface controller drivers for fast packet processing. It is designed to run in user space and provides a framework for building high-performance networking applications. The architecture is depicted in the diagram below.



#### **Core Components of DPDK:**

* **Environment Abstraction Layer (EAL):** This is the foundation of DPDK. It abstracts the specifics of the underlying operating system (e.g., Linux, FreeBSD) and provides a uniform interface for the upper layers to access hardware resources. The EAL is responsible for:
  + Scanning and probing the PCI bus for network devices.
  + Reserving memory (hugepages) and managing memory mappings.
  + Initializing CPU cores and managing thread affinity.
  + Providing atomic operations and other low-level primitives.
* **Poll Mode Driver (PMD):** This is the most critical component for performance. Unlike traditional drivers that rely on interrupts, a PMD operates by continuously polling the NIC's receive queue for new packets.
  + **How it works:** A dedicated CPU core runs an infinite loop, repeatedly asking the NIC, "Do you have any new packets?"
  + **Trade-off:** This eliminates the massive overhead of interrupt handling and context switching, but it means the CPU core runs at 100% utilization, even when there is no network traffic. This is a deliberate design choice for maximum performance..
* **Memory Management (Mempool & Mbuf):** Efficient memory management is paramount in high-speed networking.
  + **Hugepages:** Standard 4KB memory pages can lead to a high number of Translation Lookaside Buffer (TLB) misses when processing large amounts of data. DPDK uses hugepages (e.g., 2MB or 1GB) to drastically reduce the number of pages and, consequently, TLB misses, improving memory access speed.
  + **Mbuf (Memory Buffer):** This is the data structure DPDK uses to hold a network packet. It's analogous to the Linux kernel's sk\_buff but is designed to be lighter and more efficient. It contains metadata about the packet (length, offload flags, etc.) and a pointer to the actual packet data.
  + **Mempool:** To avoid the expensive overhead of dynamic memory allocation (malloc/free) for every single packet, DPDK pre-allocates a fixed-size pool of packet buffers (mbufs) at startup. When a packet arrives, an mbuf is quickly taken from the pool. When processing is complete, it's returned to the pool. This is a lockless, fast operation.
* **Ring Buffers (rte\_ring):** In a multi-core environment, packets often need to be passed from one core to another (e.g., one core does RX, another does protocol processing). DPDK provides a highly optimized, fixed-size, lockless FIFO ring buffer for this purpose. It allows for extremely fast, thread-safe communication between cores without the contention of traditional locks.

### **4. Key Techniques for High Performance**

DPDK achieves its high performance through a combination of several key techniques:

1. **Kernel Bypass:** As discussed, this is the fundamental principle. By moving the data path to user space, DPDK avoids the system call, context switch, and interrupt overheads of the kernel.
2. **Polling instead of Interrupts:** This is the core mechanism of the PMD. It trades CPU efficiency (in terms of idle time) for raw packet processing speed and deterministic latency.
3. **Zero-Copy:** DPDK is designed to minimize data movement. Packets are DMA'd by the NIC directly into pre-allocated hugepage memory in user space. The application processes the packet right there in the buffer, without copying it to another location.
4. **CPU Pinning and Affinity:** To prevent the OS scheduler from moving a packet-processing thread from one CPU core to another (which would invalidate CPU caches and cause performance jitter), DPDK threads are typically "pinned" to specific logical cores.
5. **NUMA (Non-Uniform Memory Access) Awareness:** On multi-socket server systems, accessing memory attached to a different CPU socket is slower than accessing local memory. DPDK is NUMA-aware, meaning it allocates memory for packet buffers and runs packet processing threads on the same NUMA node where the NIC is physically connected. This minimizes memory access latency.
6. **Batch Processing:** Instead of processing one packet at a time, DPDK functions are designed to handle batches of packets (e.g., 32 at once). This amortizes the overhead of function calls and allows for better utilization of CPU instruction caches and SIMD (Single Instruction, Multiple Data) instructions.

### **5. DPDK Packet Processing Models**

DPDK applications typically follow one of two main processing models:

* **Run-to-Completion:** A single CPU core handles the entire lifecycle of a packet. It polls the RX queue, receives a burst of packets, processes them (e.g., performs parsing, lookup, modification), and then transmits them out on a TX queue. This model is simple to implement and avoids inter-core communication overhead but is limited by the processing power of a single core.
* **Pipeline Model:** Packet processing is broken down into stages, and each stage is assigned to a different CPU core.
  + **Core A (RX Core):** Polls the NIC and places received packets into a software ring.
  + **Core B (Worker Core):** Dequeues packets from the ring, performs the heavy processing (e.g., deep packet inspection, complex routing), and enqueues them into another ring.
  + **Core C (TX Core):** Dequeues packets from the second ring and transmits them out the NIC.

Code Example without DPDK

Code Example with DPDK

SystemDesign

1. **UML Behavioral Diagrams**

Unified Modeling Language (UML) diagrams focus on illustrating the dynamic aspects of a software system, showcasing how it behaves, responds to events, and undergoes state changes during runtime.

1. State Machine Diagrams:

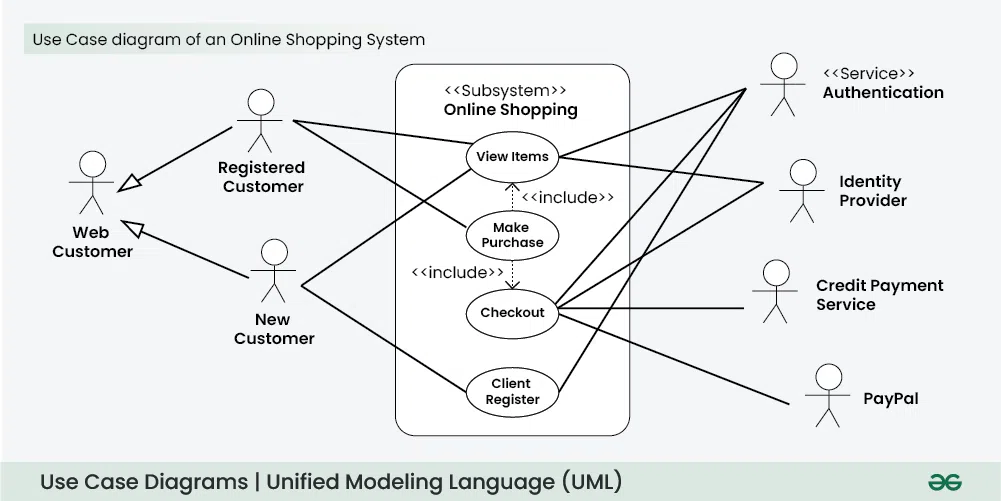
used to represent the condition of the system or part of the system at finite instances of time. It’s a behavioral diagram and it represents the behavior using finite state transitions.



1. Use Case Diagram:

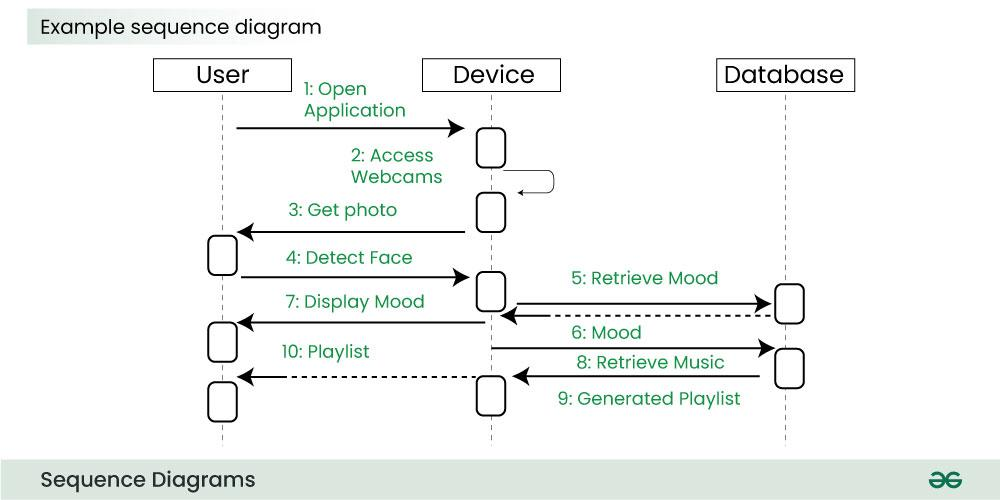
relationships play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios.

need to gather and clarify user requirements, use case diagrams help visualize how different users interact with the system.



c. Sequence Diagrams:

diagram that visually represents the interactions between objects or components in a system over time. They focus on the order and timing of messages or events exchanged between different system elements.



**Hash Tables for Caching**: A hash table can be used in a web server to store frequently requested web pages in cache. The server uses a hash of the page URL to check the cache when a user requests a page. The page is provided rapidly if it is in the cache, eliminating the need to regenerate the full page.

**Graphs for Social Networks:** The network of friends on a social media site such as Facebook can be shown as a graph. Finding connections between users or proposing new connections can be accomplished by using algorithms such as depth-first search and breadth-first search.

**Trie for Auto-Complete:** Tries are used in search engine or messaging app auto-complete functionalities. Based on the input prefix, the trie assists the user in predicting and suggesting the most likely words or sentences as they type.

**Priority Queues for Task Scheduling:** Tasks in an operating system can be scheduled using a priority queue. To ensure that crucial activities are completed on time, higher-priority tasks are completed before lower-priority ones.

**Dijkstra's Algorithm for Routing:** Dijkstra's algorithm is used in GPS navigation systems to determine the shortest path between two points. It assists users in taking the most effective route to their destination.

**Binary Search in Databases:** In a database system, when searching for a specific record based on a unique identifier, binary search can be applied. This is especially useful in scenarios where the dataset is large, ensuring a quick retrieval of the desired information.

1. **Fundamental Data Structures and Algorithms for System Design**
2. Arrays: An array is a collection of elements stored in contiguous memory locations. It provides fast and constant-time access to elements using an index.

**Advantage**

* 1. Random Access : i-th item can be accessed in O(1)
  2. Cache Friendliness : Since items / references are stored at contiguous locations, we get the advantage of locality of reference.
  3. Fundamental and linear data structure using which we build other data structures like Stack Queue, Deque, Graph, Hash Table, etc.

**Disadvantage**

* 1. Memory Allocation Issues: Allocating large arrays can cause memory exhaustion, leading to crashes, especially on systems with limited resources.
  2. Insertion and Deletion Challenges: Adding or removing elements requires shifting subsequent elements, making these operations inefficient.
  3. Lack of Flexibility: Fixed size and limited type support make arrays less adaptable than structures like linked lists or trees.

1. Linked List: linear data structure where elements are stored in nodes, and each node points to the next one in the sequence.

**Advantage**

* 1. process scheduling (for example circular linked list for round robin scheduling)
  2. Algorithms that need to frequently insert or delete items from large collections of data.
  3. implementations of Queue and Deque data structures because of fast deletions (or insertions) from the front of the linked lists.

**Disadvantage**

* 1. Slow Access Time: Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for
  2. Linked lists have a higher overhead compared to arrays, as each node in a linked list requires extra memory to store the reference to the next node.
  3. Cache Inefficiency: Linked lists are cache-inefficient because the memory is not contiguous.

1. Stack: linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.(LIFO)

**Advantage**

* 1. Function calls: Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
  2. Recursion: Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.
  3. Expression evaluation: Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).
  4. Syntax parsing: Stacks are used to check the validity of syntax in programming languages and other formal languages.

**Disadvantage**

* 1. Limited access: Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
  2. Not suitable for random access: Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
  3. Potential for overflow: If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.

1. Queue: linear data structure in which the insertion of a new element from the end and removal of an existing element takes place from the front.(FIFO)

**Advantage**

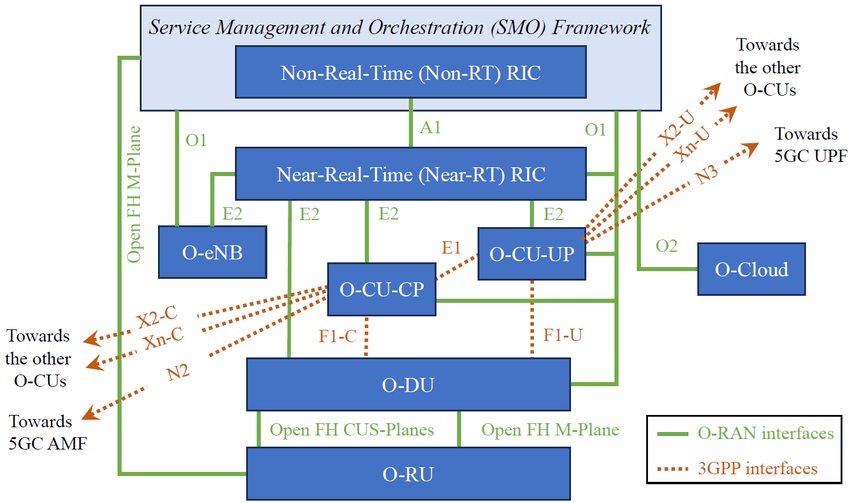
* 1. Multi programming: when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
  2. Network: In a network, a queue is used in devices such as a router or a switch. Another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
  3. CPU Job Scheduling: The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
  4. Shared resources: Queues are used as waiting lists for a single shared resource.

**Disadvantage**

* 1. Operations such as insertion and deletion of elements from the middle are time consuming.

Searching for an element takes O(N) time.

5G Basics



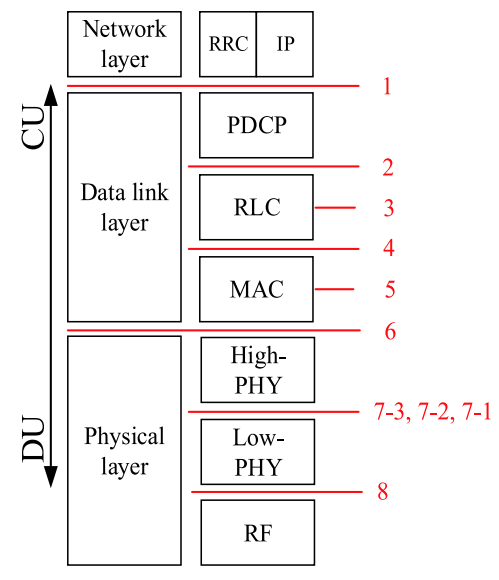
### **1. RAN Nodes**

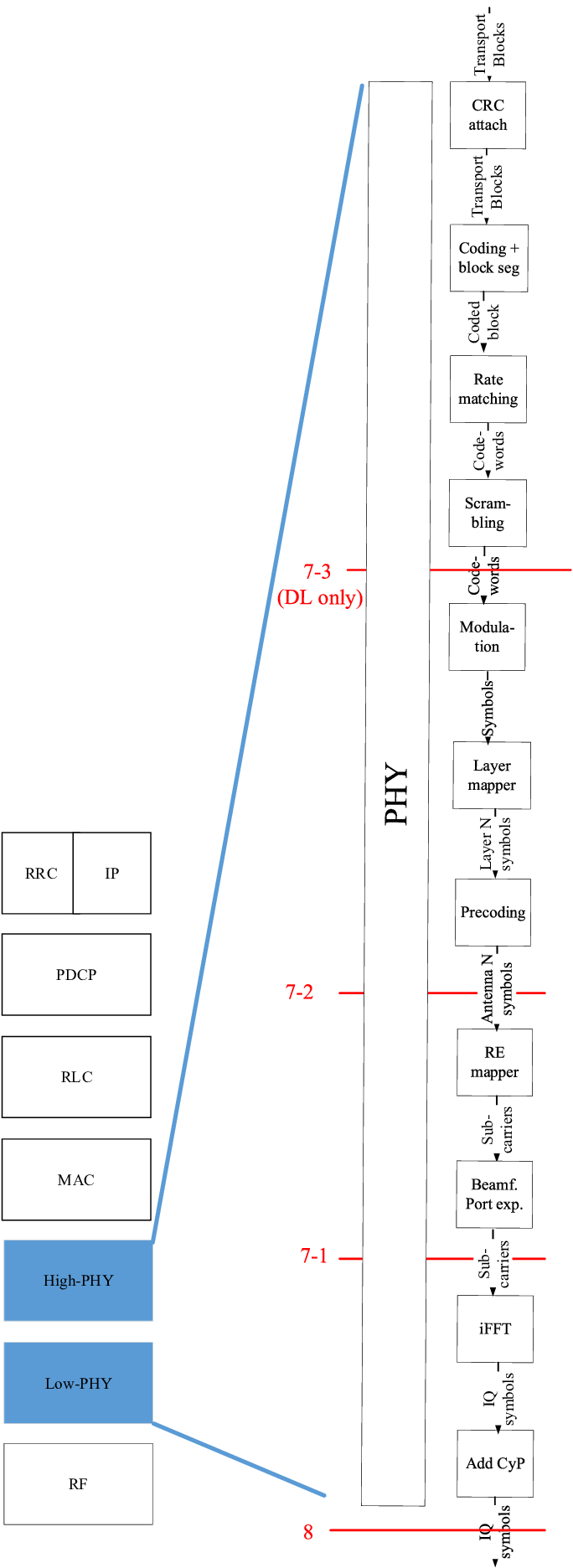
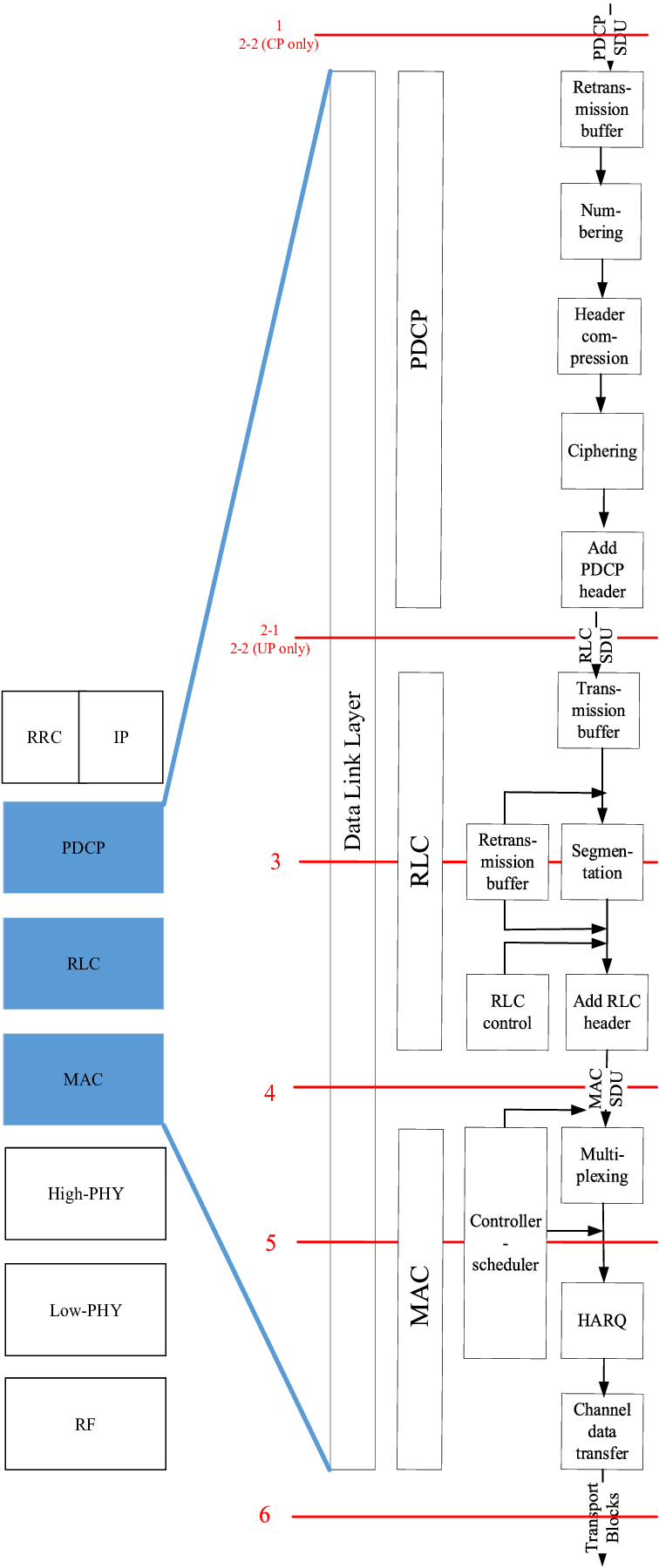
* **O-Cloud:** a Cloud Computing platform comprising physical infrastructure nodes to host O-RAN functions, like near RT-RIC, O-DU, etc.; supporting software components (e.g. OS, VM monitoring, container runtime), management, and orchestration functions.
* **O-RU (O-RAN Radio Unit)**: a logical node hosting a low-PHY layer (e.g. FFT/IFFT, PRACH) and RF-based on LLS (Lower-Layer Split);
* **O-DU (O-RAN Distributed Unit)**: a logical node hosting RLC (Radio Link Control) / MAC (Medium Access Control) / high-PHY layers based on LLS;
* **O-CU-CP (O-RAN Central Unit-Control Plane)**: a logical node hosting RRC (Radio Resource Control) and CP (Control Plane) part of PDCP (Packet Data Convergence Protocol);
* **O-CU-UP (O-RAN Central Unit-User Plane)**: a logical node hosting SDAP (Service Data Adaptation Protocol) and UP (User Plane) part of PDCP;
* **near-RT RIC (near Real-Time RAN Intelligent Controller or nRT RIC)**: a logical node enabling near-RT control/optimization of RAN elements and resources via fine-grained data collection and actions over E2. nRT RIC may include AI/ML workflow.
* **Non-RT RIC (Non-Real-Time RAN Intelligent Controller or NRT RIC)**: a logical node enabling Non-RT control/optimization of RAN elements and resources, capturing AI/ML workflow, and policy-based guidance of applications/features in nRT RIC;
* **xApp**: an application designed to run on nRT RIC, likely to consist of one or more microservices, that identifies data to consume and provide. xApp is independent of nRT RIC and may be provided by a third party.
* **SMO (Service and Management Orchestration)**: a system supporting orchestration of O-RAN components that includes NRT RIC.

**2. RAN Interfaces**

* **NG Interface**: Connects the gNodeB (gNB) to the 5G Core Network (5GC), specifically to the Access and Mobility Management Function (AMF) for control plane using (NG-C) and User Plane Function (UPF) for user plane using (NG-U). Uses NGAP (NG Application Protocol) for control plane and GTP-U (GPRS Tunneling Protocol) for user plane over IP.
* **Xn Interface**: Connects two gNodeBs (gNBs) for inter-gNB communication to support handovers, dual connectivity, and coordination for load balancing or interference management. Uses XnAP (Xn Application Protocol) for control plane and GTP-U for user plane.
* **F1 Interface**: Connects the Centralized Unit (CU) to the Distributed Unit (DU) within a gNB. It separates higher-layer (CU) and lower-layer (DU) processing for flexible deployment (e.g., CU in the cloud, DU at the edge). It uses F1AP (F1 Application Protocol) for control planes and GTP-U for user planes.
* **E1 Interface**: Connects the CU-Control Plane (CU-CP) to the CU-User Plane (CU-UP) within a gNB to enable separation of control and user plane processing for scalability and optimization. It uses E1AP (E1 Application Protocol).
* **Uu Interface**: The air interface between the User Equipment (UE) and the gNB. To handle radio communication, including physical layer, MAC, RLC, PDCP, and SDAP protocols.
* **FH Interface**: Connects the DU to the RU, often referred to as the fronthaul in 3GPP functional split (e.g., Option 7-2x). It carries digitized radio signals (IQ samples) between RU and DU. Typically uses eCPRI (enhanced Common Public Radio Interface) or CPRI for low-latency, high-bandwidth communication.
* **A1 interface:** defined between NRT RIC and nRT RIC, through which NRT RIC provides nRT RIC with policies, enrichment info, and ML model updates, while from the other hand, nRT RIC provides back the policy feedback (i.e. how the policy set by NRT RIC works)
* **E2 interface:** actually touches and gets into the specific entities within the base station, i.e. O-RU, O-DU, O-CU. Therefore from one side we can control what is happening within that BaseStation, using monitor, suspend, override, control messages, and execute actions coming from xApps/nRT RIC, and get data collection and feedback from those entities.
* **O1 and Open-Fronthaul M-plane interface**: a regular FCAPS (Fault, Configuration, Accounting, Performance, Security) interface with configuration, reconfiguration, registration, security, performance, monitoring aspects exchange with individual nodes, like O-CU-UP, O-CU-CP, O-DU, O-RU, as well as nRT RIC.
* **O2 interface**: to manage the platform resources and workload (like resource scaling and FCAPS).

**3. RAN splits**

****

****

### **User Plane Protocol Stack:**

### 

### **SDAP (Service Data Adaptation Protocol):** This is a new layer in 5G NR. You should know its primary function: mapping QoS (Quality of Service) flows to DRBs (Data Radio Bearers) and marking packets with the QFI (QoS Flow ID). This is a critical concept for meeting the diverse requirements of 5G use cases (e.g., eMBB, URLLC, mMTC).

### **PDCP (Packet Data Convergence Protocol):** Key functions include header compression (RoHC), ciphering and deciphering, and handling of in-sequence delivery and reordering. You should understand how it's used for both user data and control plane signaling.

### **RLC (Radio Link Control):** This layer provides reliable data transfer over the radio link. Be prepared to discuss the different RLC modes (Transparent Mode - TM, Unacknowledged Mode - UM, Acknowledged Mode - AM) and their respective use cases. Functions include segmentation, reassembly, and error correction via ARQ (Automatic Repeat reQuest).

### **MAC (Medium Access Control):** The MAC layer is responsible for multiplexing/demultiplexing data, scheduling information reporting, and error correction via HARQ (Hybrid Automatic Repeat reQuest). Be prepared to discuss scheduling (dynamic and proactive), logical channels to transport channels mapping, and the Random Access procedure.

### **Control Plane Protocol Stack:**

* **RRC (Radio Resource Control):** This is the heart of L3. The RRC layer manages all aspects of the radio connection between the UE and the gNB. You should be able to discuss:
  + **RRC States:** RRC\_IDLE, RRC\_INACTIVE, and RRC\_CONNECTED. Know the transitions between these states and the conditions that trigger them.
  + **RRC Procedures:** System Information broadcast (MIB, SIBs), connection establishment, re-establishment, release, mobility management (handover), and security functions.
  + **Radio Bearers:** The difference between Signaling Radio Bearers (SRBs) and Data Radio Bearers (DRBs) and their respective RLC modes.

**NAS (Non-Access Stratum):** While not strictly part of the RAN, you should understand the role of NAS messages (e.g., for registration, authentication) and how they are transparently carried by the RRC layer.

## **5G NR Random Access (RACH) Procedure**

RACH allows UE to establish uplink synchronization and request resources from the gNB when no dedicated uplink resources are available.

**A. Contention-Based RACH (CB-RACH) 4-Step Procedure:** have higher latency and be used in initial setup or RRC re-establishment.

1. **Msg1:** PRACH Preamble transmission
2. **Msg2:** Random Access Response (RAR)
3. **Msg3:** Scheduled transmission (contains UE ID)
4. **Msg4:** Contention Resolution

**UE gNB**

**│ │**

**│ ──── Msg1: PRACH ────────► │ (Random preamble selection)**

**│ Preamble │**

**│ │**

**│ ◄──── Msg2: RAR ────────── │ (UL grant + timing advance)**

**│ (DL-SCH on PDSCH) │**

**│ │**

**│ ──── Msg3: Scheduled ────► │ (RRC Connection Request +**

**│ transmission │ UE identity on UL-SCH)**

**│ (UL-SCH on PUSCH) │**

**│ │**

**│ ◄──── Msg4: Contention ─── │ (Contention resolution +**

**│ Resolution │ UE identity confirmation)**

**│ (DL-SCH on PDSCH) │**

**│ │**

**UE gNB-DU gNB-CU-CP gNB-CU-UP**

**│ │ │ │**

**│ ── Msg1: PRACH ─► │ │ │ (Random preamble)**

**│ Preamble │ │ │**

**│ │── RACH Indication ─►│ │ (F1-C: UL RRC Message Transfer)**

**│ │ (F1-C Interface) │ │ (Contains preamble info, timing)**

**│ │ │ │**

**│ │◄─── RAR Decision ───│ │ (F1-C: DL RRC Message Transfer)**

**│ │ (UL grant + TA) │ │ (RAR content, timing advance)**

**│ │ │ │**

**│ ◄─ Msg2: RAR ─── │ │ │ (UL grant + timing advance)**

**│ (DL-SCH/PDSCH) │ │ │**

**│ │ │ │**

**│ ── Msg3: RRC ───► │ │ │ (RRC Connection Request +**

**│ Connection │ ── UL RRC Msg ────► │ │ UE identity on UL-SCH)**

**│ Request │ Transfer (F1-C) │ │**

**│ (UL-SCH/PUSCH) │ │ │**

**│ │ │── RRC Processing──│**

**│ │ │ (Authentication,│**

**│ │ │ Security setup)│**

**│ │ │ │**

**│ │ ◄─── Contention ────│ │ (F1-C: DL RRC Message Transfer)**

**│ │ Resolution Msg │ │ (RRC Setup/Reject message)**

**│ │ (F1-C Interface) │ │**

**│ │ │ │**

**│ ◄─ Msg4: RRC ─── │ │ │ (Contention resolution +**

**│ Setup │ │ │ RRC Setup Complete)**

**│ (DL-SCH/PDSCH) │ │ │**

**│ │ │ │**

**│ ── RRC Setup ───► │ │ │ (RRC Setup Complete)**

**│ Complete │ ── Setup Complete ─►│ │**

**│ │ (F1-C) │── Bearer Setup ──►│ (E1: Bearer Context Setup)**

**│ │ │ Request │ (DRB establishment)**

**│ │ │◄─── Bearer Setup──│ (E1: Bearer Context Setup**

**│ │ │ Response │ Response)**

**│ │ ◄─── UE Context ────│ │ (F1-C: UE Context Setup)**

**│ │ Setup (F1-C) │ │ (DRB configuration)**

**B. Contention-Free RACH (CF-RACH) 2-Step Procedure:** have lower latency and be used in Handover, beam failure recovery

1. **Msg1:** Dedicated PRACH Preamble + Data
2. **Msg2:** Random Access Response

**UE gNB**

**│ │**

**│ ──── Dedicated Preamble ──►│ (gNB assigns dedicated preamble)**

**│ Assignment │**

**│ │**

**│ ──── Msg1: PRACH + ──────► │ (Dedicated preamble + data)**

**│ Data transmission │ (PRACH + PUSCH)**

**│ │**

**│ ◄──── Msg2: Success ────── │ (ACK/NACK + UL grant if needed)**

**│ Indication │ (DL-SCH on PDSCH)**

**│ │**

**UE gNB-DU gNB-CU-CP gNB-CU-UP**

**│ │ │ │**

**│ │ ◄─── Dedicated ──── │ │ (F1-C: RACH Configuration)**

**│ │ Preamble │ │ (During handover preparation**

**│ │ Assignment │ │ or beam failure recovery)**

**│ │ (F1-C) │ │**

**│ │ │ │**

**│ ── Dedicated ─── │ │ │ (gNB assigns dedicated preamble)**

**│ Preamble │ │ │**

**│ Assignment │ │ │**

**│ │ │ │**

**│── Msg1: PRACH ─► │ │ │ (Dedicated preamble + data)**

**│ + Data │ ── RACH Success ──► │ │ (F1-C: UL RRC Message Transfer)**

**│ (PRACH+PUSCH) │ Indication │ │ (Contains preamble ID, data)**

**│ │ (F1-C) │ │**

**│ │ │──RRC Processing── │**

**│ │ │ (Handover │**

**│ │ │ completion/ │**

**│ │ │ beam recovery) │**

**│ │ │ │**

**│ │ ◄─── Success ────── │ │ (F1-C: DL RRC Message Transfer)**

**│ │ Response │ │ (ACK + UL grant if needed)**

**│ │ (F1-C) │ │**

**│ │ │ │**

**│ ◄─Msg2: Success ─│ │ │ (ACK/NACK + UL grant)**

**│ Indication │ │ │ (DL-SCH on PDSCH)**

**│ (DL-SCH/PDSCH)│ │ │**

**│ │ │ │**

**│ │ │──Context Update─► │ (E1: Bearer Context**

**│ │ │ (if needed) │ Modification - if required)**

**│ │ ◄─── UE Context ────│ │ (F1-C: UE Context**

**│ │ Update (F1-C) │ │ Modification)**

**│ │ │ │**

## **MAC Scheduler**

The gNB scheduler decides which UEs get resources, when, and how much based on QoS requirements, channel conditions, and fairness considerations.

The MAC scheduler is responsible for several critical tasks:

**Resource Allocation:** It allocates radio resources—specifically, time and frequency blocks—to different users. In 5G, this is done on a per-slot basis. The scheduler determines which user equipment (UE) gets to transmit or receive data in which part of the available spectrum.

**Quality of Service (QoS) Management:** Different applications have different requirements. A video call needs low latency and consistent bandwidth, while a simple email download can tolerate a longer delay. The MAC scheduler uses QoS information (e.g., from the RRC layer) to prioritize traffic, ensuring that real-time services meet their stringent requirements.

**Error Correction:** The MAC layer is involved in the HARQ (Hybrid Automatic Repeat reQuest) process. If a packet is received with errors, the MAC scheduler manages the retransmission, prioritizing it over new data transmissions to ensure data integrity.

**Multiplexing and Demultiplexing:** The scheduler aggregates data from multiple logical channels (e.g., different services for a single user) into a single transport block for efficient transmission. On the receiving end, it does the reverse, demultiplexing the data back to the correct logical channels.

Uplink Scheduling:

**UE gNB Scheduler**

**│ │**

**│ ──── BSR (Buffer Status) ──► │ (UE reports buffer status)**

**│ │**

**│ ──── CSI (Channel State) ──► │ (Channel quality info)**

**│ │**

**│ │ ── Scheduling Decision ──**

**│ │ (Based on QoS, fairness,**

**│ │ channel conditions)**

**│ │**

**│ ◄──── DCI (Downlink ────── │ (Resource allocation info)**

**│ Control Information) │ (PDCCH)**

**│ │**

**│ ──── Data transmission ───► │ (PUSCH/PDSCH based on grant)**

**│ │**

**│ ◄──── HARQ-ACK/NACK ─── │ (Acknowledgment)**

Downlink Scheduling:

**gNB Scheduler UE**

**│ │**

**│ ── DL Data Arrival ── │ (From core network/buffer)**

**│ & Scheduling Decision │**

**│ │**

**│ ── DCI 1\_0/1\_1 ──────────► │ (DL assignment on PDCCH)**

**│ (Format indicates │ (Contains: MCS, RB allocation,**

**│ DL grant) │ HARQ process ID, NDI, RV)**

**│ │**

**│ ── DL Data ──────────────► │ (PDSCH transmission in slot n)**

**│ (PDSCH in slot n) │ (Based on K0 offset from DCI)**

**│ │**

**│ │ ── Data Reception & ──**

**│ │ CRC Check**

**│ │**

**│ ◄── HARQ-ACK/NACK ──────── │ (PUCCH/PUSCH in slot n+k1)**

**│ (slot n+k1) │ (k1 configured by RRC)**

**│ │**

**│ ── Retransmission ────────► │ (If NACK received)**

**│ (if needed) │ (Adaptive/non-adaptive)**

**│ │**

Docker Kubernetes

### **1. Containers and VMs**

**Containers**: Containers are lightweight, portable units that package an application and its dependencies (libraries, binaries, configuration files) into a single executable package. They share the host operating system’s kernel, which makes them highly efficient in terms of resource usage (CPU, memory, disk).

**Virtual Machines (VMs)**: VMs emulate an entire operating system, including the kernel, on top of a hypervisor (e.g., VMware, VirtualBox). Each VM requires its own OS, making it heavier in resource consumption compared to containers.

| **Feature** | **Containers** | **Virtual Machines (VMs)** |
| --- | --- | --- |
| **Isolation Level** | Process-level | Hardware-level (full OS) |
| **Size** | Lightweight (MBs) | Heavy (GBs) |
| **Startup Time** | Fast (milliseconds to seconds) | Slow (seconds to minutes) |
| **Resource Usage** | Shares host OS kernel | Requires full OS for each VM |
| **Performance** | Near-native | Slight overhead due to virtualization |
| **Portability** | High (across environments) | Medium |
| **Tools** | Docker, Podman | VMware, VirtualBox, KVM |

Containers are ideal for microservices, CI/CD pipelines, and cloud-native apps, while VMs are used for legacy apps or when complete OS isolation is needed.

### **2. Docker**

### Docker operates on a client-server architecture. means there's a client that sends commands, and a server (daemon) that executes them.

### Here's a breakdown of the key components:

* Docker Daemon (dockerd):

This is the core server component of Docker, often referred to simply as dockerd.

It runs continuously in the background on your host operating system and its primary responsibility is to listen for Docker API requests from the Docker CLI or other clients and manage Docker objects.

The dockerd handles all the heavy lifting: building, running, and distributing your Docker containers.

It manages images, containers, networks, and volumes. When you run a docker run command, it's the dockerd that pulls the image, creates the container, allocates resources, and starts the process.

* Docker Engine:

is a broader term that includes the Docker Daemon, the Docker CLI, and the REST API.

It's the complete set of tools that allows us to build, ship, and run applications using Docker.

Essentially, it's the runtime environment for Docker.

* Docker Objects:

**Images:** image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

Think of an image as a blueprint or a template for creating containers.

Images are immutable once created.

They are built from a Dockerfile. Eg: ubuntu:latest, nginx:stable, python:3.9-slim.

**Containers:** container is a runnable instance of an image.

When you run a Docker image, it becomes a container.

Containers are isolated environments that share the host OS kernel but have their own filesystem, processes, and network interfaces.

They are ephemeral by design, meaning any changes made inside a container are lost when it's stopped unless explicitly persisted (e.g., using volumes).

**Networks:** Docker networks allow containers to communicate with each other and with the outside world.

Docker provides several networking drivers (e.g., bridge, host, overlay) to facilitate different communication patterns.

We can create custom networks to isolate applications or services.

**Volumes:** are the preferred mechanism for persisting data generated by and used by Docker containers.

They are managed by Docker and are stored on the host filesystem, outside the container's writable layer.

This ensures that data persists even if the container is removed or recreated.

Volumes are crucial for stateful applications. Eg. database dependent applications

* REST API:

The Docker Daemon exposes a RESTful API.

This API allows various clients (like the Docker CLI, other tools, or even custom scripts) to communicate with and control the Docker Daemon programmatically.

When you type a command like docker run or docker build, the Docker CLI translates that into an API request and sends it to the dockerd.

* Docker CLI:

It is the primary way users interact with Docker.

**Manage images:** docker pull (download), docker build (create), docker push (upload), docker images (list).

**Manage containers:** docker run (create and start), docker ps (list running), docker stop (stop), docker start (start), docker rm (remove), docker exec (run command inside).

**Manage networks:** docker network create, docker network ls, docker network connect.

**Manage volumes:** docker volume create, docker volume ls, docker volume inspect.

**Inspect Docker objects:** docker inspect (get detailed information about an image, container, etc.).

**View logs:** docker logs

### **3. Dockerfile**

### A text file that contains a set of instructions for building a Docker image. It's essentially a script that automates the process of creating a custom image, defining everything from the base operating system to the application's code, dependencies, and runtime environment. **Instructions**: Each line in a Dockerfile is an instruction, typically starting with a keyword (e.g., FROM, RUN, COPY, CMD) followed by Arguments (e.g. apt-get update).

### **Layers**: Each instruction creates a new, read-only layer on top of the previous one. When you make changes to a Dockerfile, only the changed layers and subsequent layers are rebuilt, leading to faster build times.

### **Context**: When you run docker build, Docker sends the contents of the build context (the directory where your Dockerfile resides, and any files/folders within it that you reference) to the Docker daemon.

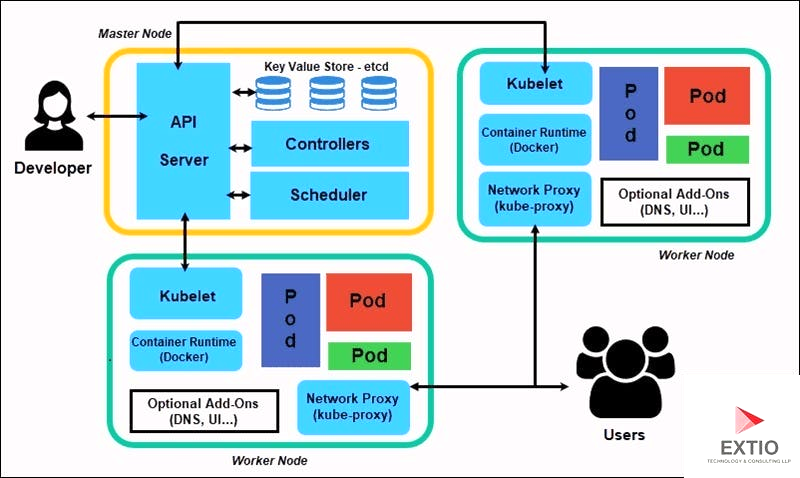
### some of the most frequently used instructions

1. **FROM** <image>[:<tag>]:
   1. Purpose: Specifies the base image for your build. This is almost always the first instruction in a Dockerfile.
   2. Content: The name of an existing Docker image (e.g., ubuntu, node, python) and an optional tag (e.g., latest, 18.04, 16-alpine).
   3. Example: FROM ubuntu:22.04 (Starts with Ubuntu 22.04 as the base OS).
2. **WORKDIR** /path/to/workdir:
   1. Purpose: Sets the working directory inside the container for any subsequent RUN, CMD, ENTRYPOINT, COPY, or ADD instructions.
   2. Content: An absolute path within the container.
   3. Example: WORKDIR /app (All following commands will operate relative to /app).
3. **COPY** <source> <destination>:
   1. Purpose: Copies files or directories from your build context (your local machine) into the image at the specified destination.
   2. Content: <source>: Path(s) to files or directories relative to the build context. <destination>: Absolute path inside the container.
   3. Example: COPY . . (Copies everything from the current directory of the build context into the /app directory inside the image).
4. **RUN** <command>:
5. Purpose: Executes commands during the image build process. This is typically used to install packages, set up configurations, compile code, etc.
6. Content: A shell command that will be executed inside the image during the build. Each RUN instruction creates a new layer.
7. Example: RUN apt-get update && apt-get install -y nodejs (Updates package lists and installs Node.js).
8. **EXPOSE** <port>:
   1. Purpose: Informs Docker that the container listens on the specified network ports at runtime. It's a documentation instruction and doesn't actually publish the port. You need to use the -p flag with docker run to publish the port.
   2. Content: The port number(s).
   3. Example: EXPOSE 80 (Indicates the application inside listens on port 80).
9. **ENV** <key>=<value>:
   1. Purpose: Sets environment variables inside the image. These variables will be available to the running container.
   2. Content: Key-value pairs for environment variables.
   3. Example: ENV NODE\_ENV=production (Sets a NODE\_ENV environment variable).
10. **CMD** ["executable", "param1", "param2"] (exec form) or CMD command param1 param2 (shell form):
    1. Purpose: Provides default commands or arguments for a running container. There can only be one CMD instruction in a Dockerfile. If multiple are present, only the last one takes effect.
    2. Example: CMD ["node", "server.js"] (When the container starts, it will run node server.js).
11. **ENTRYPOINT** ["executable", "param1", "param2"]:
    1. Purpose: Configures a container that will run as an executable. Unlike CMD, the ENTRYPOINT command and parameters are always executed when the container starts. Any CMD instructions or command-line arguments to docker run are appended as arguments to the ENTRYPOINT.
    2. Content: Similar to CMD (exec form is preferred).
    3. Example: ENTRYPOINT ["/usr/bin/supervisord"] (Always starts supervisord, and any CMD or docker run arguments become arguments to supervisord).

To build the own image from Dockerfile : docker build Dockerfile -t aditya/myapp

### **3. Kubernetes**

### Kubernetes operates as a distributed system with a Control Plane that manages the cluster's state and a set of Worker Nodes that run the actual workloads (applications). The Control Plane makes global decisions about the cluster (e.g., scheduling, scaling), while Worker Nodes execute the workloads in the form of containers within Pods.



Workflow: Deploying an Application

1. A user runs *kubectl apply -f app.yaml* to deploy an application.
2. The **kube-apiserver** validates the YAML, stores the Deployment in **etcd**, and notifies the **kube-controller-manager**.
3. The Deployment controller creates a ReplicaSet, which in turn creates Pods.
4. The **kube-scheduler** assigns the Pods to Worker Nodes based on resource availability and constraints.
5. On each Worker Node, the **kubelet** pulls the container images using the Container Runtime and starts the Pods.
6. The **kube-proxy** sets up networking rules to route traffic to the Pods via a Service.
7. If a Pod fails, the **kube-controller-manager** detects the failure and creates a replacement Pod, which the scheduler assigns to a node.

### Master Node(Control Plane)

Responsible for managing the cluster, maintaining its desired state, and responding to cluster events. The Control Plane consists of several components that work together to manage the cluster. These components can run on a single master node in smaller setups or be distributed across multiple nodes for high availability in production environments.

1. **kube-apiserver :** It is the front-end of the Kubernetes Control Plane, acting as the primary interface for all cluster operations. It exposes the Kubernetes API, which is used by users, CLI tools (like kubectl), and other components to interact with the cluster.
   * Processes RESTful API requests (GET, POST, PUT, DELETE) to create, update, or delete resources like Pods, Services, Deployments, etc. Authenticates and authorizes requests using mechanisms like RBAC (Role-Based Access Control). Validates and mutates resource configurations before storing them in etcd. Acts as a communication hub between all Control Plane components, Worker Nodes, and external clients.
   * *kubectl apply -f deployment.yaml*, the request is sent to the kube-apiserver, which validates the YAML, stores the resource in etcd, and triggers the necessary actions (e.g., scheduling Pods).
2. **etcd :** It is a distributed, consistent key-value store that serves as the single source of truth for the cluster’s state.
   * Stores all Kubernetes objects (e.g., Pods, Services, ConfigMaps, Deployments) and their metadata.
   * Provides a highly available and consistent database for cluster configuration and state.
   * Uses a watch mechanism to notify components of changes to resources (e.g., when a Pod is created or deleted).
3. **kube-scheduler :** It is responsible for assigning Pods to Worker Nodes based on resource requirements, policies, and constraints.
   * Filtering: Identifies nodes that meet the Pod’s requirements (e.g., sufficient CPU/memory).
   * Scoring: Ranks the filtered nodes based on optimization criteria (e.g., least resource usage, balanced distribution).
   * Places the Pod on the selected node by updating its specification in etcd via the kube-apiserver

**4. kube-controller-manager :** It Runs multiple independent controllers, each responsible for a specific resource or process

* + **Node Controller**: Monitors node health and manages node lifecycle (e.g., marking nodes as unhealthy if they fail).
  + **Replication Controller**: Ensures the desired number of replicas for a ReplicationController or ReplicaSet is running.
  + **Endpoints Controller**: Populates Endpoint objects for Services to map them to Pod IPs.
  + **Service Account & Token Controller**: Creates default ServiceAccounts and API tokens for Pods.
  + Other controllers handle Deployments, StatefulSets, Jobs, etc.

### Worker Node

Responsible for managing the cluster, maintaining its desired state, and responding to cluster events. The Control Plane consists of several components that work together to manage the cluster. These components can run on a single master node in smaller setups or be distributed across multiple nodes for high availability in production environments.

1. **kubelet :** It is an agent running on each Worker Node, responsible for managing Pods and their containers.
   * Communicates with the kube-apiserver to receive Pod specifications.
   * Ensures that the containers in assigned Pods are running and healthy.
   * Interacts with the Container Runtime to start, stop, and manage containers.
   * Reports node and Pod status (e.g., CPU/memory usage, Pod health) to the kube-apiserver.
   * Executes health checks (liveness and readiness probes) defined in Pod specifications.
   * Manages Pod lifecycle, including mounting volumes and setting up container networking.
2. **kube-proxy:** It is a network proxy that runs on each Worker Node, managing network connectivity for Services. Supports multiple modes for traffic routing
   * Userspace: Proxies traffic through a userspace process (older, less efficient).
   * iptables: Uses Linux iptables to configure NAT and load balancing (default in older versions).
   * IPVS: Uses Linux IP Virtual Server for high-performance load balancing.
   * For a Service with three Pods, kube-proxy configures iptables or IPVS rules to distribute incoming traffic across the Pods’ IP addresses.
3. **Container Runtime:** It is the software responsible for running containers on each Worker Node.
   * Pulls container images from registries (e.g., Docker Hub, private registries).
   * Creates, starts, stops, and deletes containers as directed by the kubelet.
   * Manages container lifecycle, including networking and storage setup.
   * **containerd**: A lightweight, high-performance runtime used in modern Kubernetes clusters.
   * **CRI-O**: A runtime designed specifically for Kubernetes, supporting the Container Runtime Interface (CRI).
   * **Docker**: Historically used, but now less common due to its replacement by containerd in many setups.

## **I. Docker In-Depth**

### **A. Core Concepts (Deep Dive)**

* **What is a Container?**
  + Isolation: Linux namespaces (PID, NET, UTS, MNT, IPC, USER)
  + Resource Management: cgroups (CPU, memory, I/O)
  + Union File Systems: OverlayFS, AUFS – how layers work, copy-on-write
* **Docker Engine Architecture:**
  + Daemon (dockerd)
  + CLI (docker client)
  + REST API
  + Containerd, runc
* **Images & Dockerfiles:**
  + **Dockerfile best practices:** Multi-stage builds, .dockerignore, choosing optimal base images (Alpine vs. Ubuntu), layer caching, reducing image size, security considerations (non-root users, least privilege).
  + Understanding image layers and their immutability.
  + Building efficient and secure Docker images.
  + Image tagging and versioning.
* **Containers:**
  + Lifecycle: Create, Start, Stop, Pause, Unpause, Restart, Remove.
  + Interacting with containers (exec, logs, inspect).
  + Container resource limits (CPU, memory).
* **Networking:**
  + **Container Network Model (CNM):** Bridge, Host, None, Overlay, Macvlan.
  + Docker's default bridge network.
  + User-defined networks.
  + Port mapping and exposing ports.
  + Container-to-container communication.
* **Storage:**
  + **Volumes:** Named volumes, anonymous volumes, bind mounts.
  + Managing data persistence.
  + Tmpfs mounts.
* **Docker Compose:**
  + Defining multi-container applications.
  + Service dependencies, networking, and volume management with Compose.
  + Scaling services with Compose.
* **Docker Swarm (for foundational understanding, though Kubernetes is dominant for orchestration):**
  + Swarm architecture (managers, workers).
  + Services, tasks, and scaling.
  + Load balancing in Swarm.

### **B. Advanced Docker Concepts & Best Practices**

* **Security:**
  + Running containers as non-root.
  + Image scanning and vulnerability management.
  + Content Trust.
  + Limiting capabilities (Linux capabilities).
  + Seccomp and AppArmor profiles.
  + Protecting the Docker daemon socket.
* **Optimization:**
  + Minimizing image size.
  + Efficient caching in Dockerfiles.
  + Container health checks.
* **Troubleshooting:**
  + Using docker logs, docker inspect, docker stats.
  + Debugging containerized applications.

## **II. Kubernetes In-Depth**

Kubernetes is the standard for container orchestration. Mastering it is crucial for deploying and managing applications at scale.

### **A. Core Concepts (Deep Dive)**

* **Kubernetes Architecture:**
  + **Control Plane (Master Node components):**
    - kube-apiserver: The frontend for the Kubernetes control plane.
    - etcd: Distributed key-value store for cluster state.
    - kube-scheduler: Assigns Pods to Nodes.
    - kube-controller-manager: Runs various controllers (Node, Replication, Endpoints, Service Account & Token Controllers).
    - cloud-controller-manager (if applicable).
  + **Worker Node components:**
    - kubelet: Agent that runs on each node in the cluster.
    - kube-proxy: Network proxy for Services.
    - Container Runtime (e.g., containerd, Docker).
* **Objects & Resources:**
  + **Pods:** The smallest deployable unit. Understanding multi-container Pods (sidecar, init containers).
  + **Controllers:**
    - **Deployments:** Declarative updates for Pods and ReplicaSets. Rolling updates, rollbacks.
    - **ReplicaSets:** Ensures a specified number of Pod replicas are running.
    - **StatefulSets:** For stateful applications, stable network identities, ordered deployment/scaling.
    - **DaemonSets:** Ensures a Pod runs on all (or selected) nodes.
    - **Jobs/CronJobs:** For batch processing.
* **Networking:**
  + **CNI (Container Network Interface):** How Pods get IPs and communicate.
  + **Pod-to-Pod communication:** Flat network model.
  + **Services:** Abstraction for Pods.
    - ClusterIP: Internal service.
    - NodePort: Exposes service on each Node's IP at a static port.
    - LoadBalancer: Integrates with cloud provider load balancers.
    - ExternalName: Maps a service to a DNS name.
  + **Ingress:** External access to services via HTTP/HTTPS routing. Ingress controllers (Nginx, Traefik).
  + **Network Policies:** Controlling Pod-to-Pod communication based on labels and namespaces.
* **Storage:**
  + **Volumes:** Ephemeral vs. persistent.
  + **Persistent Volumes (PV):** Cluster-wide storage resources.
  + **Persistent Volume Claims (PVC):** Requests for PVs by Pods.
  + **Storage Classes:** Dynamic provisioning of PVs.
* **Configuration:**
  + **ConfigMaps:** Storing non-confidential configuration data.
  + **Secrets:** Storing sensitive information.
* **Labels, Selectors, Annotations:**
  + Organizing and selecting Kubernetes objects.

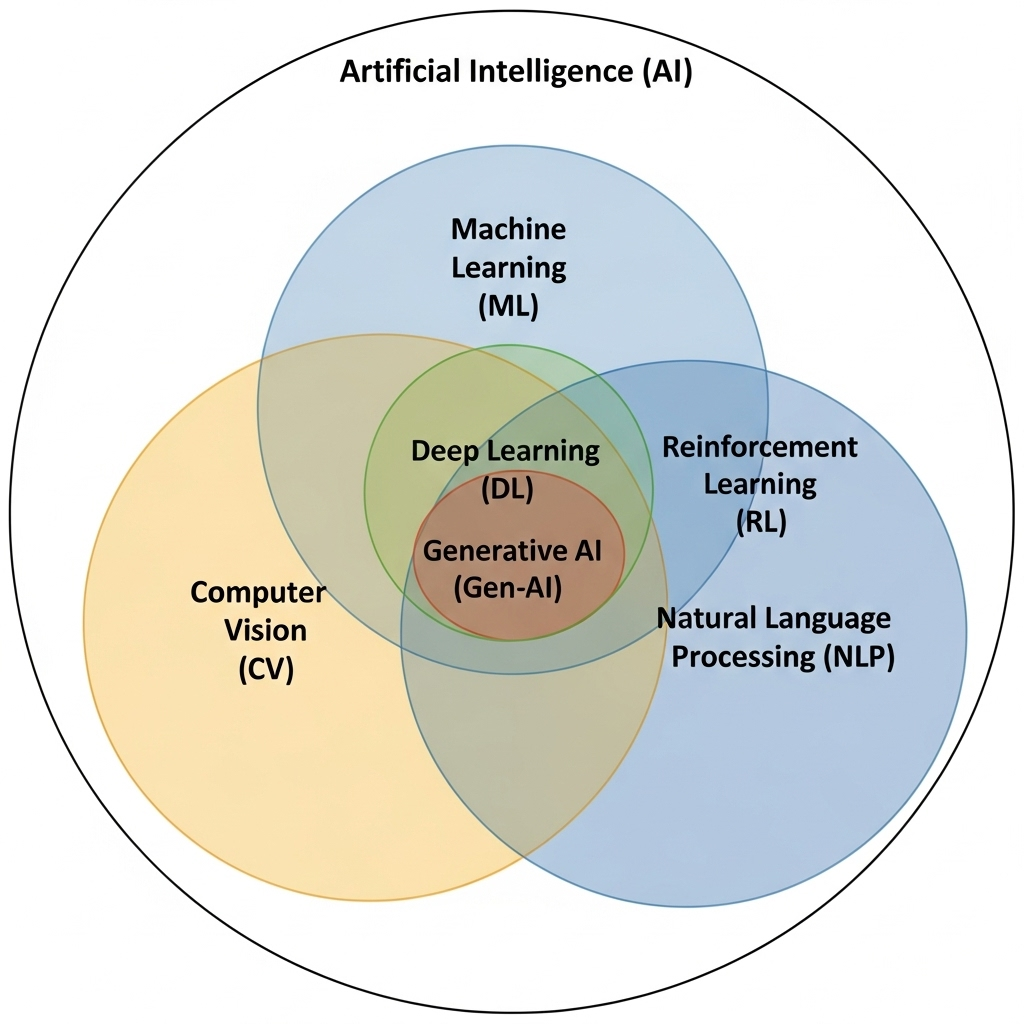
### **B. Advanced Kubernetes Concepts & Best Practices**

* **Security:**
  + **RBAC (Role-Based Access Control):** Defining user/service permissions.
  + **Pod Security Standards (or Pod Security Policies if older cluster):** Enforcing security configurations for Pods.
  + Network Policies for granular traffic control.
  + Image security: Scanning, trusted registries.
  + Managing Secrets securely (e.g., Sealed Secrets, Vault integration, CSI Secrets Store Driver).
* **Resource Management:**
  + **Resource Requests & Limits:** Ensuring efficient resource allocation and preventing resource starvation.
  + **Quality of Service (QoS) classes:** Guaranteed, Burstable, BestEffort.
  + **Horizontal Pod Autoscaler (HPA):** Scaling Pods based on CPU/memory utilization or custom metrics.
  + **Cluster Autoscaler:** Automatically adjusting cluster size.
* **Scheduling:**
  + Node Selectors, Node Affinity/Anti-affinity.
  + Taints and Tolerations.
  + Resource Quotas and Limit Ranges.
* **Helm:**
  + Package manager for Kubernetes.
  + Creating, managing, and deploying Helm charts.
* **Observability:**
  + **Logging:** Centralized logging (Elastic Stack, Loki).
  + **Monitoring:** Prometheus, Grafana for metrics collection and visualization.
  + **Tracing:** Jaeger, Zipkin for distributed tracing.
  + **Health Checks:** Liveness and Readiness probes.
* **Troubleshooting:**
  + Using kubectl describe, kubectl get events, kubectl logs.
  + Debugging Pods and services.
* **Advanced Deployments:**
  + Canary deployments, Blue/Green deployments (often with Ingress controllers).
  + Custom Resource Definitions (CRDs) and Operators (for extending Kubernetes)

Cloud

AI ML

### **Artificial Intelligence**

**Artificial Intelligence (AI)**: Broad field of creating systems that mimic human intelligence. Example: A chatbot that understands and responds to queries.

**Machine Learning (ML):** Subset of AI where systems learn from data to make predictions or decisions. Example: Spam email filter learning from labeled emails.

**Natural Language Processing (NLP):** AI subfield focused on understanding and processing human language. Example: Voice assistants like Siri interpreting spoken commands.

**Deep Learning (DL):** A subset of ML using neural networks with multiple layers. Example: Image recognition systems.

**Computer Vision (CV):** AI field focused on enabling machines to interpret visual data. Example: Facial recognition technology.

**Reinforcement Learning (RL):** ML technique where agents learn by interacting with an environment. Example: Game-playing AI like AlphaGo.

**Generative AI (Gen-AI):** AI that creates new content, often using ML and NLP techniques. Example: GPT generating human-like text or DALL-E creating images from prompts.

### **1. Supervised Learning**

The model is trained on labeled data, the dataset contains both input features (X) and the corresponding correct outputs (Y).. The goal is to learn a mapping from input to output.

**Process**

1. **Data Collection:** You gather a dataset where each piece of data is paired with a corresponding label (the "correct answer").
2. **Model Training:** The labeled data is fed into an algorithm. The algorithm's job is to build a model by finding a pattern or function that maps the input data “train data” to its correct label. The model makes predictions, and its errors are calculated and used to adjust the model's parameters, iteratively improving its accuracy.
3. **Prediction & Evaluation**: Once trained, the model is given new, unlabeled data to make predictions. Its performance is then evaluated against a separate "test data" set of labeled data to ensure it can generalize well to new scenarios.

Supervised learning problems are divided into two main categories.

1. [**Classification (Discrete outputs)**](https://colab.research.google.com/github/rsys-adsingh/Classification/blob/main/ClassificationAlgo.ipynb)**:** used when the output is a discrete, categorical value. The model predicts which class or category an input belongs to.

Algorithms: Logistic Regression, Decision Trees, (k-NN) k-Nearest Neighbors, (SVM) Support Vector Machines, Naive Bayes, Random Forest, Neural Networks.

Example: A spam filter. The model is trained on emails labeled as "spam" or "not spam".

Image recognition (identifying if an image is a cat, dog, or bird).

1. **Regression (Continuous outputs):** used when the output is a continuous numerical value. The model predicts a specific number or quantity.

Algorithms: Linear Regression, Polynomial Regression, Support Vector Regression, Random Forest Regression, Neural Networks.

Example: Predicting a house price. The model is trained on a dataset of houses with features like square footage, number of bedrooms, and location, all paired with their corresponding sale price. It then uses this learned relationship to predict the price of a new house.

### **2. Unsupervised Learning**

The model is trained on unlabeled data, the dataset contains only the input features (X). The algorithm must discover hidden patterns, structures, or relationships in the data without supervision (labels). The goal is not prediction of a known outcome but the pattern discovery.

**Process**

1. **Data Collection:** Unstructured dataset that has no labels or predefined categories.
2. **Model Training:** The unlabeled data is fed into an unsupervised learning algorithm. The algorithm's job is to analyze the data and identify inherent patterns, similarities, or relationships. The model essentially tries to organize the data into meaningful groups or representations. There is no error calculation or correction against a known output, as there is none. The algorithm "learns" by identifying the underlying structure.
3. **Insight Generation:** After the model is trained, the output reveals the patterns it has found. The results must then be interpreted by a human or a downstream application to make sense of the new structure. The model provides insights.

Unsupervised learning problems are divided into two main categories.

1. **Clustering:** Grouping data points based on similarity.

Algorithms: K-Means Clustering, Hierarchical Clustering, DBSCAN (Density-Based Spatial Clustering)

Example: Grouping news articles by topic, Detecting similar images

1. **Dimensionality Reduction:** Reducing the number of features while retaining essential information. Used to visualize high-dimensional data or remove noise.

Algorithms: PCA (Principal Component Analysis), t-SNE (t-distributed Stochastic Neighbor Embedding), Autoencoders (Neural Networks for dimensionality reduction)

Example: Compressing images, Reducing features before feeding into another ML model, Visualization (e.g., showing 100-D data in 2-D)

1. **Association Rule Learning:** Discovering relationships/rules between variables in datasets.

Algorithms: Apriori Algorithm, Eclat Algorithm

Example: Market Basket Analysis → “People who buy bread often also buy butter.”

1. **Anomaly Detection:** Identifying unusual data points. Examples: Fraud detection, fault detection in machines, network intrusion detection.

**Classification (Discrete outputs):**

* 1. Collect and preprocess data (handle missing values, normalize features).
  2. Split data into training and testing sets (e.g., 80/20 split).
  3. Choose and train a model.
  4. Evaluate on test data.
  5. Tune hyperparameters (e.g., via cross-validation) and iterate.

Examples will use simple datasets like the Iris dataset (predict flower species: setosa, versicolor, virginica)

1. **Logistic Regression:** Uses a mathematical function (sigmoid) to map any input to a value between 0 and 1, representing probability.

Best for: Binary classification, when you need probability estimates, linearly separable data.

* 1. Sigmoid function: transforms linear combination into probability
  2. Decision boundary: typically 0.5 probability threshold
  3. Maximum likelihood estimation for parameter learning
  4. **Hyperparameters**: Learning rate, regularization strength (C), max iterations.

1. **Decision Trees:** Builds a tree-like model where each internal node represents a feature split (e.g., "petal length > 2.5?"), branches are outcomes, and leaves are class predictions. It recursively splits data to maximize purity.

Best for: Binary classification, when you need probability estimates, linearly separable data.

* 1. **Training**: Greedy algorithm selects best split at each step
  2. Interpretable, handles non-linear data, no need for feature scaling but Prone to Overfitting.
  3. **Hyperparameters**: Max depth, min samples per leaf, splitter (Gini/entropy).

1. **k-Nearest Neighbors (k-NN):** Instance-based (lazy) learning: No explicit training model. For a new input, find k closest training points (using distance like Euclidean: , then predict the majority class among them. 
   1. Slow for large datasets
   2. **Hyperparameters**: k (e.g., 3-5; too small = noisy, too large = smooths boundaries), distance metric.
2. **Support Vector Machines (SVM):** Finds a hyperplane that best separates classes with maximum margin (distance to nearest points, called support vectors). For non-linear data, use kernels (e.g., RBF) to map to higher dimensions.

Objective: Minimize classification error + maximize margin.

* 1. **Training**: Quadratic optimization; soft margins allow some misclassifications (via C parameter).
  2. **Pros**: Effective in high dimensions, robust to overfitting with kernels, good for small/medium datasets.
  3. **Cons**: Computationally intensive for large data, hard to interpret, sensitive to kernel choice and parameters.
  4. **Hyperparameters**: Kernel (linear/poly/rbf), C (error penalty), gamma (for RBF: low = smooth boundary).

1. **Naive Bayes:** Probabilistic classifier based on Bayes' theorem

Best for: Binary classification, when you need probability estimates, linearly separable data.

* 1. **Training**: Compute class priors and feature likelihoods (e.g., mean/variance per class).
  2. **Pros**: Fast, simple, works well with high dimensions, good for text/categorical data, handles missing values.
  3. **Cons**: Independence assumption often unrealistic (features in Iris correlate), zero-probability issue (use smoothing).
  4. **Hyperparameters**: Smoothing parameter (alpha).

1. **Random Forest:** Ensemble of decision trees (bagging): Build multiple trees on bootstrapped subsets of data, with random feature subsets at each split.
   1. **Training**: Average predictions reduce variance.
   2. **Pros**: Reduces overfitting, handles non-linearity/missing values, feature importance scores, robust.
   3. **Cons**: Less interpretable than single tree, slower to train/predict, memory-intensive.
   4. **Hyperparameters**: Number of trees (n\_estimators), max depth, min samples.
2. **Neural Networks:** Inspired by brains; layers of nodes (input, hidden, output) connected by weights. For classification, the output layer uses softmax for probabilities. Train via backpropagation: forward pass computes output, backward adjusts weights to minimize loss (cross-entropy).
   1. **Training**: Gradient descent (e.g., Adam optimizer).
   2. **Pros**: Handles complex non-linear patterns, scalable with data, state-of-the-art for large datasets.
   3. **Cons**: Black-box (hard to interpret), requires lots of data/tuning, prone to overfitting (use dropout), computationally expensive.
   4. **Hyperparameters**: Layers/nodes, activation (ReLU), epochs, learning rate, batch size.

Coding

### **1. Dynamic Programming (DP)**

Dynamic Programming involves breaking problems into overlapping subproblems and solving them using memoization or tabulation.

1. **Knapsack-Based Problems**:

**Description**: Solve optimization problems where items have weights and values, often with a capacity constraint.

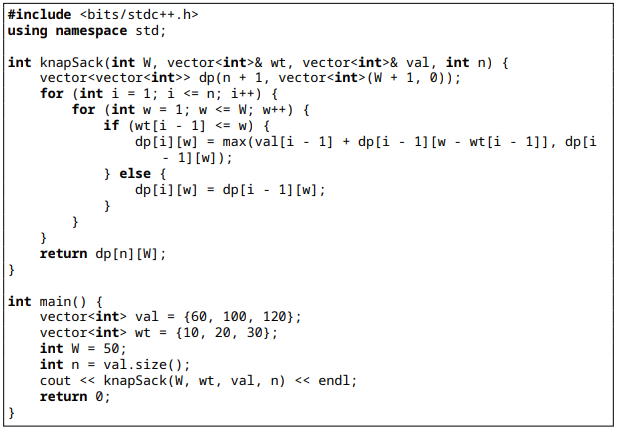
**Examples**:

* 0/1 Knapsack: Maximize value without exceeding weight limit.
* Subset Sum: Find if a subset sums to a target.
* Coin Change: Find minimum coins for a given amount.

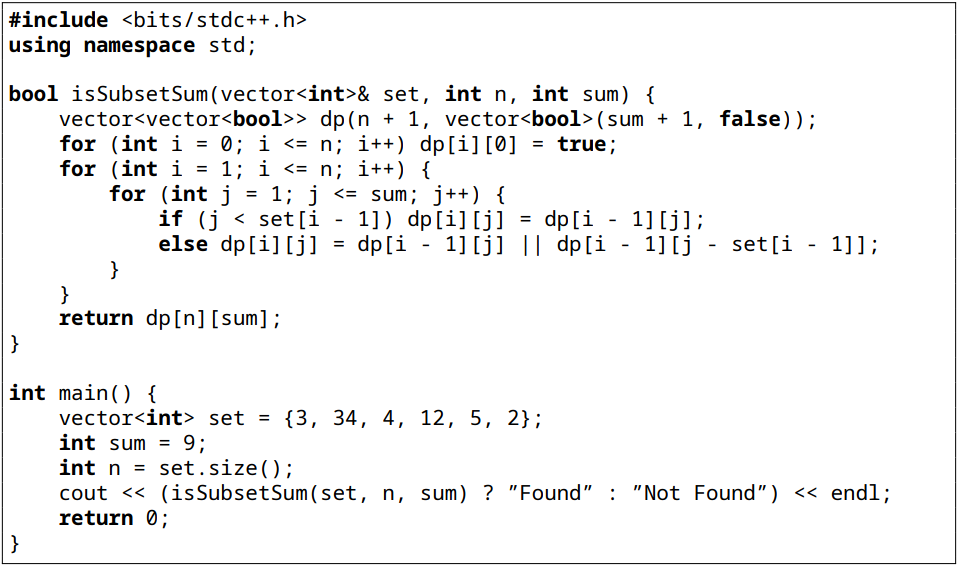
**Approach**: Use a 2D DP table (item vs. weight) or 1D table for optimization.

Time: O(n\*W), Space: O(n\*W) or O(W).

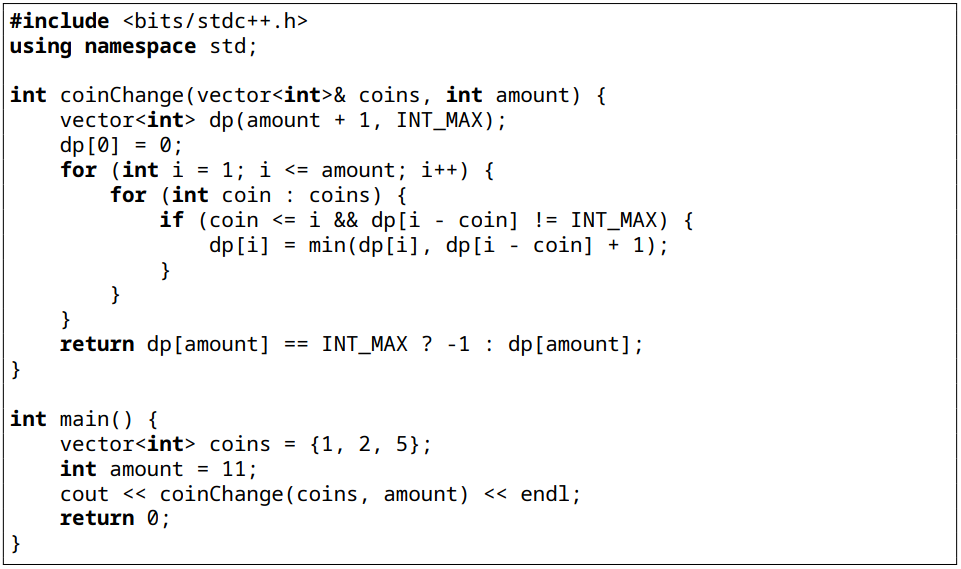
**0/1 Knapsack**: Maximize value without exceeding weight limit.



**Subset Sum:** Find if a subset sums to a target.



**Coin Change:** Find minimum coins for a given amount.



1. **Longest Common Subsequence (LCS)/Substring**:

**Description**: Find sequences or substrings common to two strings or arrays.

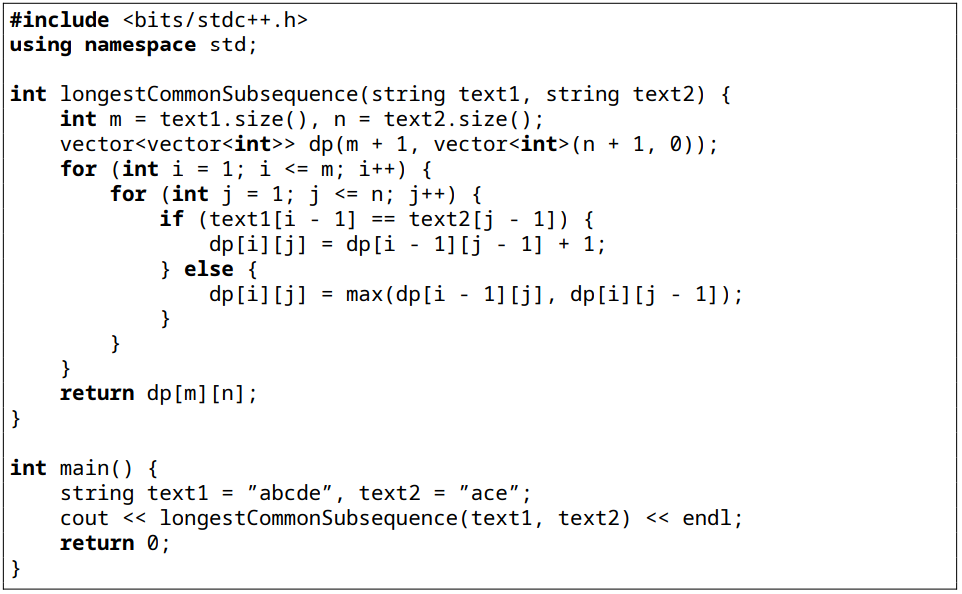
**Examples**:

* Longest Common Subsequence.
* Edit Distance (Levenshtein Distance).
* Longest Palindromic Substring.

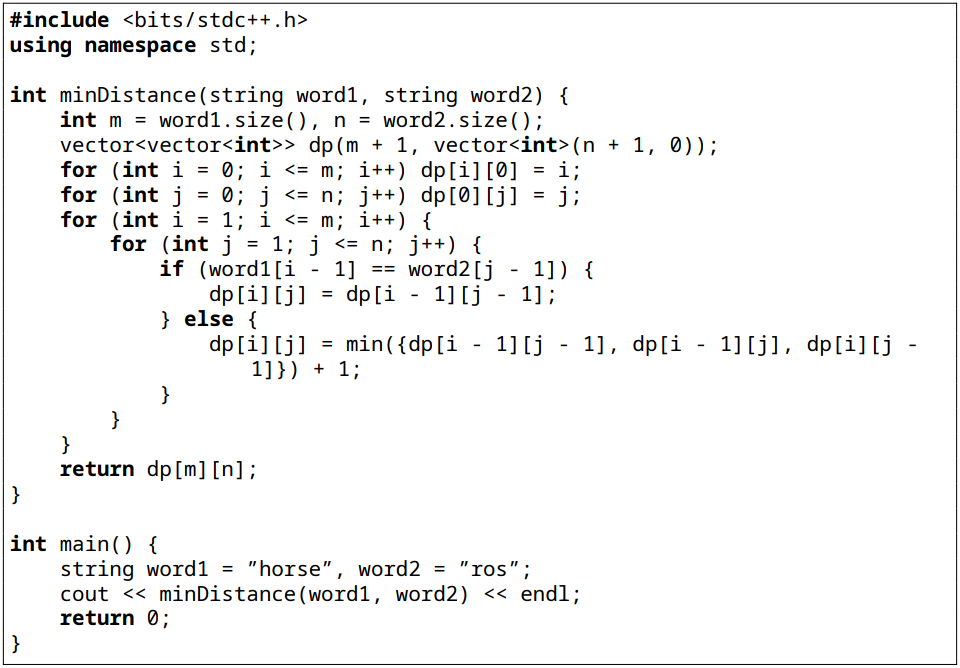
**Approach**: Use a 2D DP table to track matches.

Time: O(m\*n), Space: O(m\*n) or O(min(m,n)).

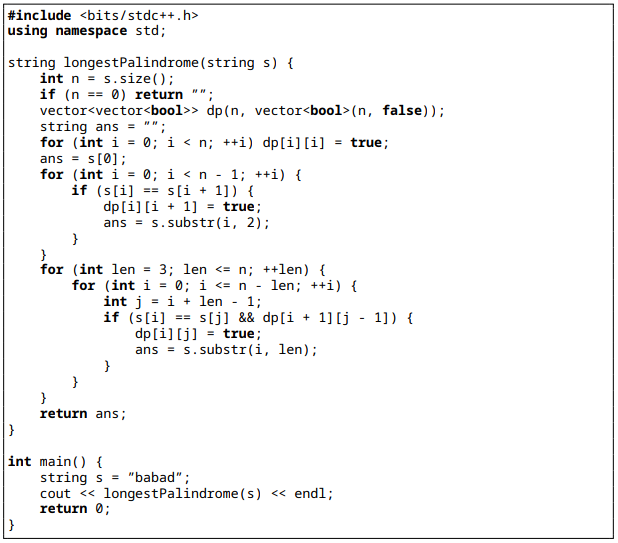
**Longest Common Subsequence:**



**Edit Distance (Levenshtein Distance).**



**Longest Palindromic Substring**



1. **Path-Based DP:**

**Description**: Find optimal paths in grids or sequences.

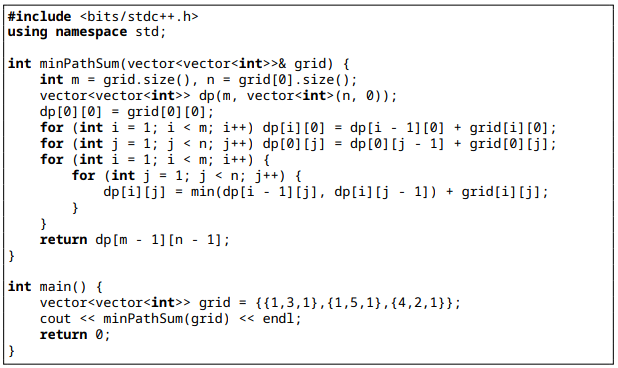
**Examples**:

* Minimum Path Sum in a grid.
* Unique Paths with obstacles.
* Longest Increasing Subsequence (LIS).

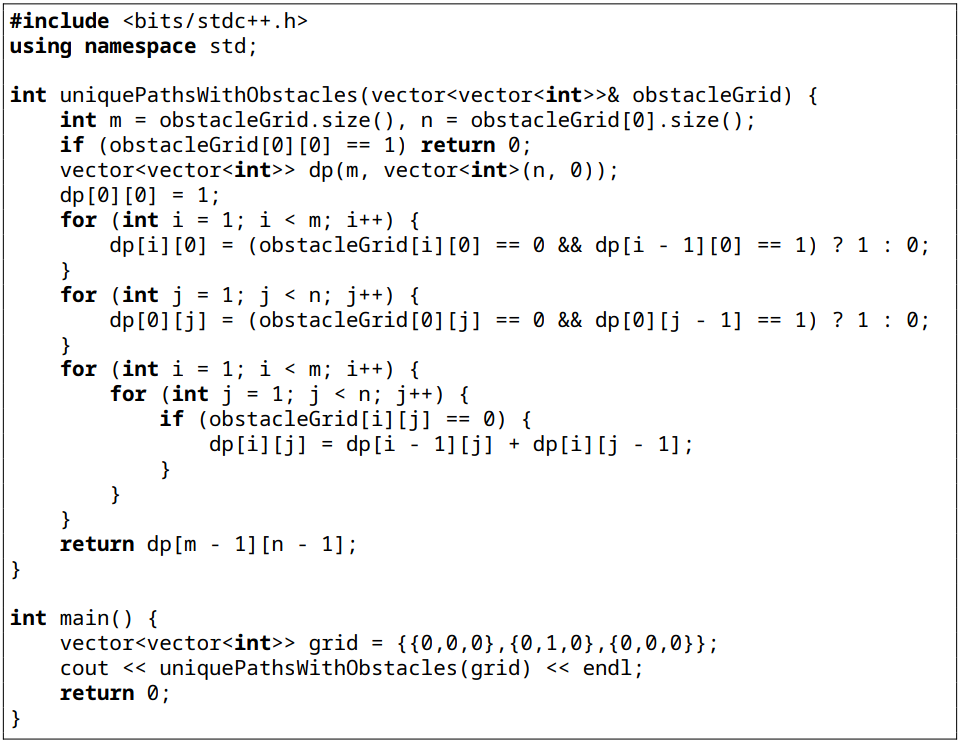
**Approach**: Use DP table to store minimum/maximum costs or counts.

Time: O(m\*n), Space: O(m\*n) or O(n).

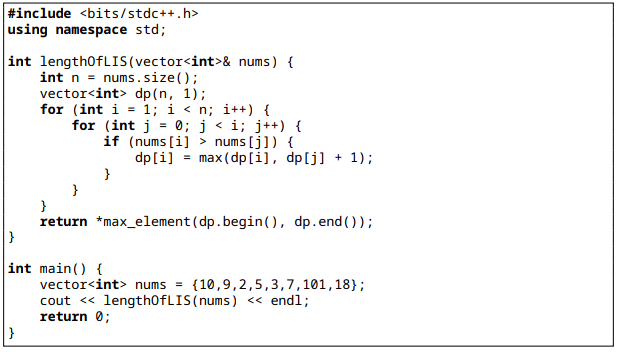
**Minimum Path Sum in a grid**

****

**Unique Paths with obstacles.**



**Longest Increasing Subsequence (LIS)**



1. **Partition Problems:**

**Description**: Divide arrays or sets into equal or specific partitions.

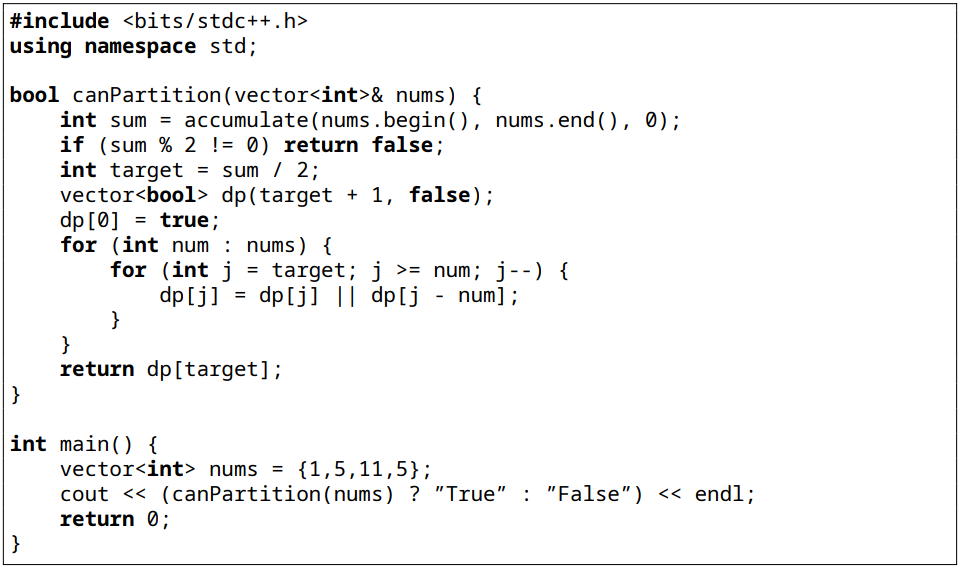
**Examples**:

* Partition Equal Subset Sum.
* Word Break
* Palindrome Partitioning.

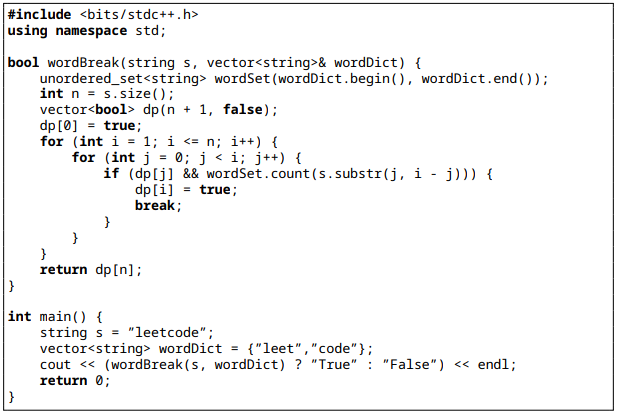
**Approach**: Use DP to track valid partitions or use backtracking. Time: O(n\*sum) or O(n2 ), Space : O(n).

Time: O(m\*n), Space: O(m\*n) or O(n).

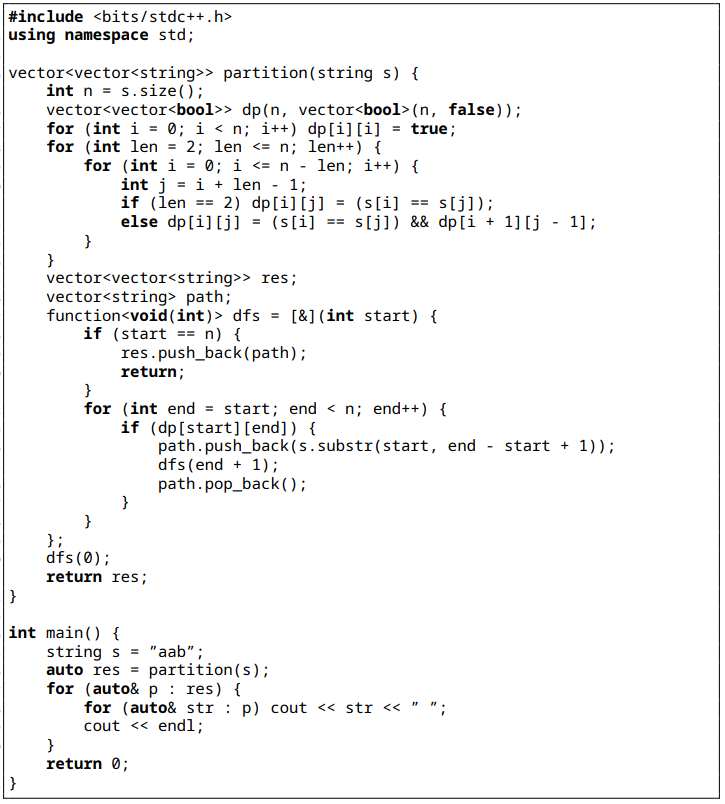
**Partition Equal Subset Sum**



**Word Break**



**Palindrome Partitioning**



### **2. Graphs**

Graph problems involve nodes and edges, solved using traversal or specialized algorithms.

1. **Graph Traversal (DFS/BFS)**

**Description**: Explore nodes using Depth-First Search (DFS) or Breadth-First Search (BFS).

**Examples**:

* Find connected components.
* Detect cycle in a graph.
* Shortest path in an unweighted graph.

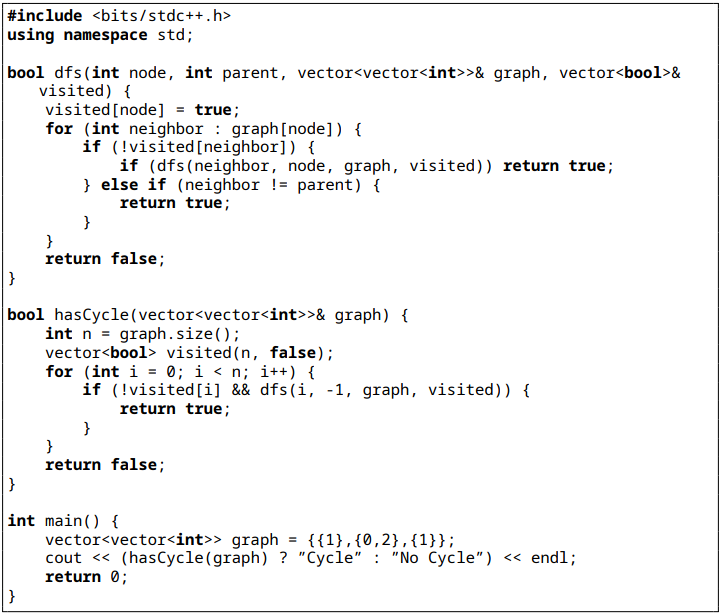
**Approach**: Use recursion (DFS) or queue (BFS).

Time: O(V+E), Space: O(V).

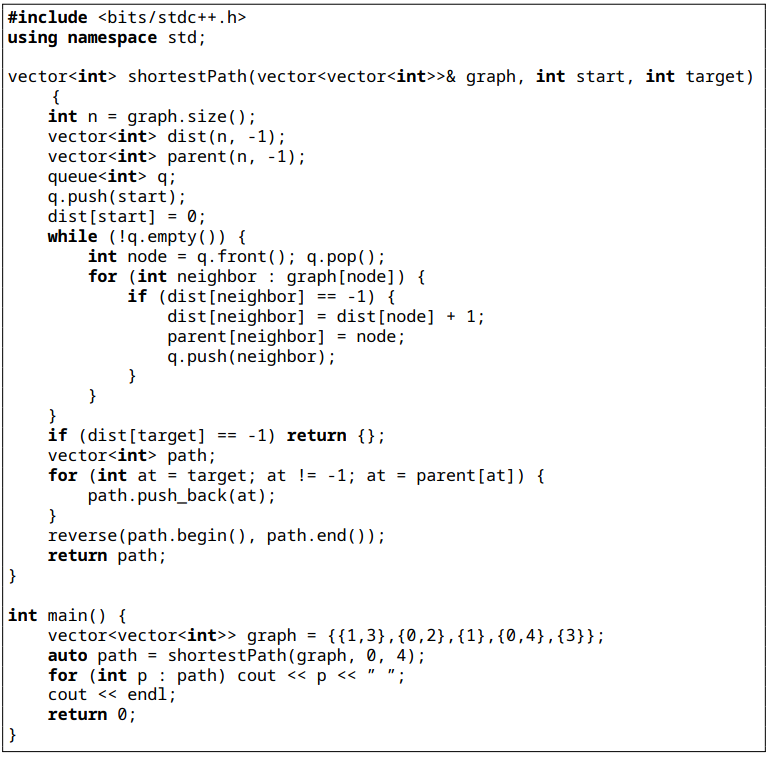
**Find connected components**

****

**Detect cycle in a graph**



**Shortest path in an unweighted graph**

****

1. **Shortest Path Algorithms**

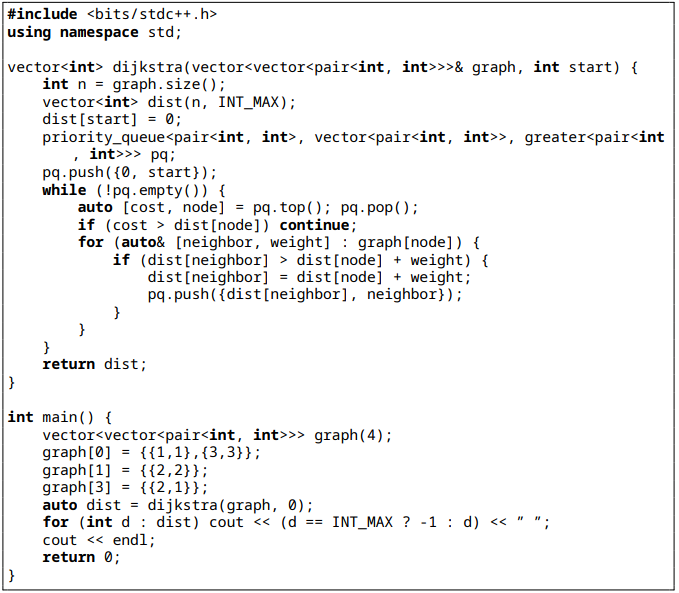
**Description**: Find shortest paths in weighted graphs.

**Examples**:

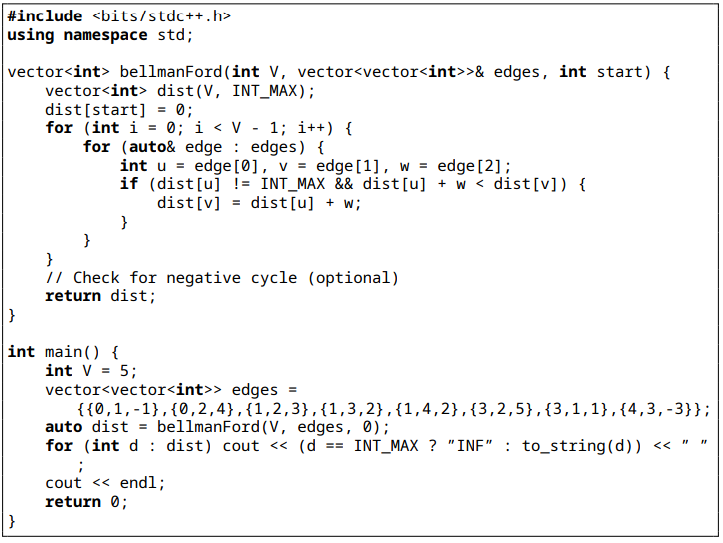
* Dijkstra’s Algorithm for single-source shortest paths.
* Bellman-Ford for negative weights.
* Floyd-Warshall for all-pairs shortest paths.

**Approach**: Use priority queue (Dijkstra) or dynamic programming (Floyd-Warshall). Time: O(V2 )orO(V ∗ E), Space : O(V ).

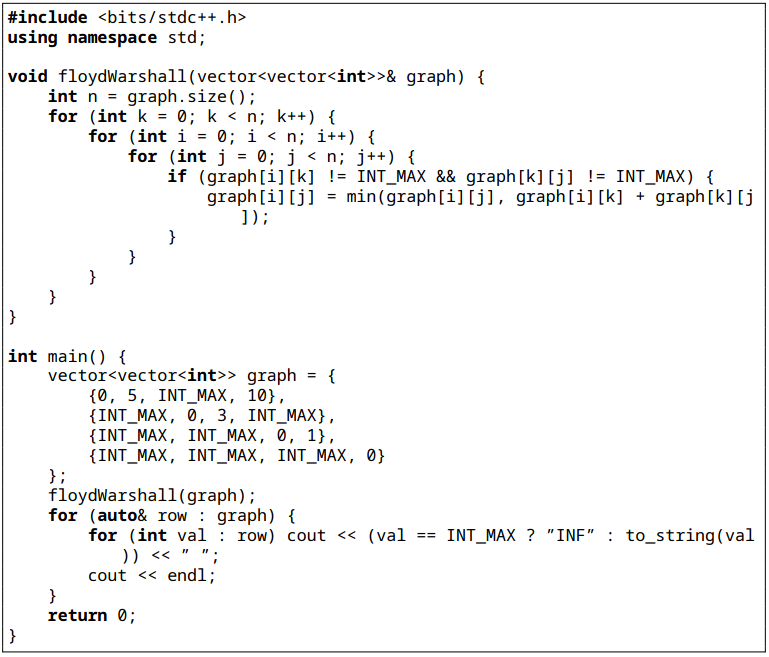
**Dijkstra’s Algorithm for single-source shortest paths**

****

**Bellman-Ford for negative weights**



**Floyd-Warshall for all-pairs shortest paths**



1. **Topological Sorting**

**Description**: Order nodes in a directed acyclic graph (DAG).

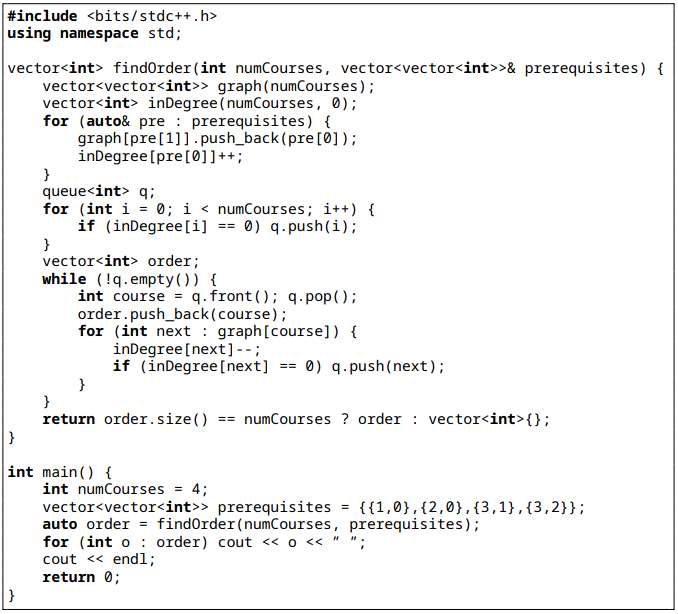
**Examples**:

* Course Schedule (find valid order of courses).
* Detect cycle in a directed graph
* Alien Dictionary.

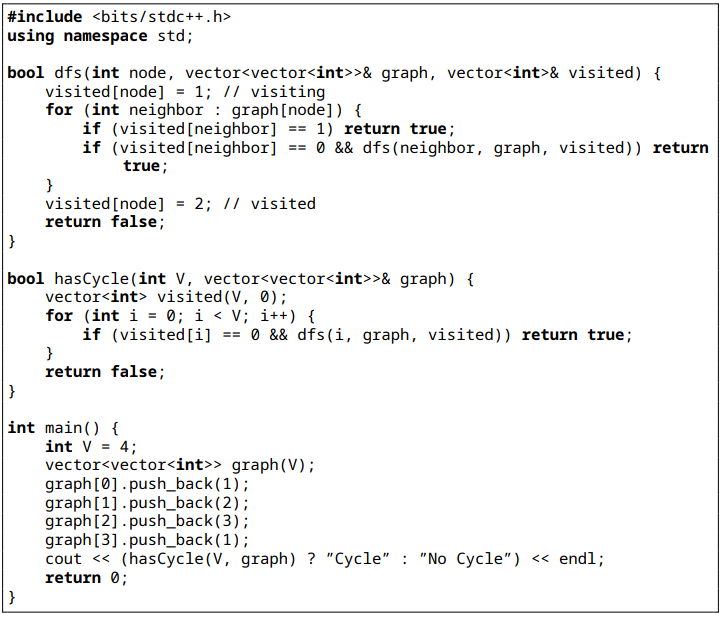
**Approach**: Use DFS or Kahn’s algorithm with in-degree tracking.

Time: O(V+E), Space: O(V).

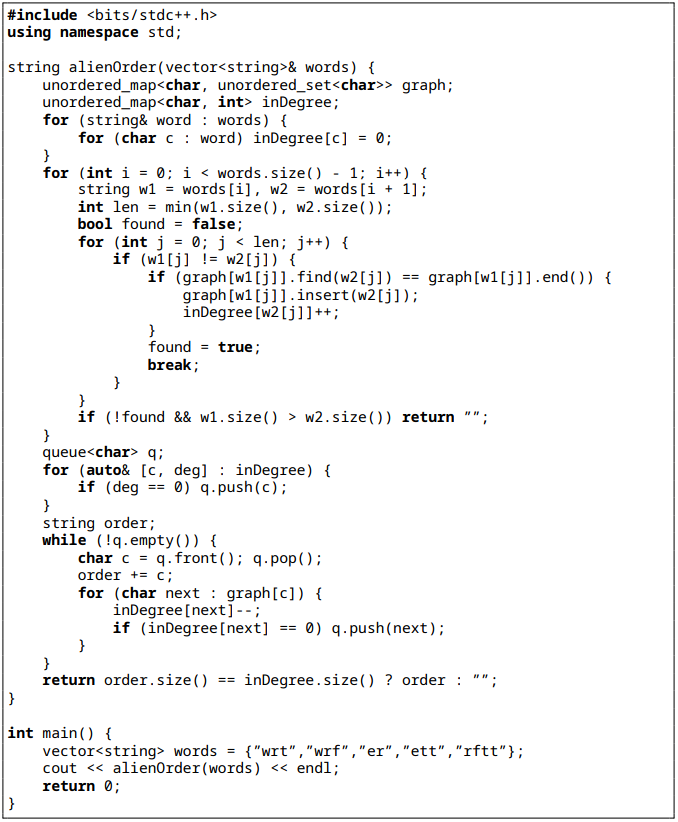
**Course Schedule (find valid order of courses)**

****

**Detect cycle in a directed graph**



**Alien Dictionary**

****

1. **Minimum Spanning Tree (MST)**

**Description**: Find a tree connecting all nodes with minimum total edge weight.

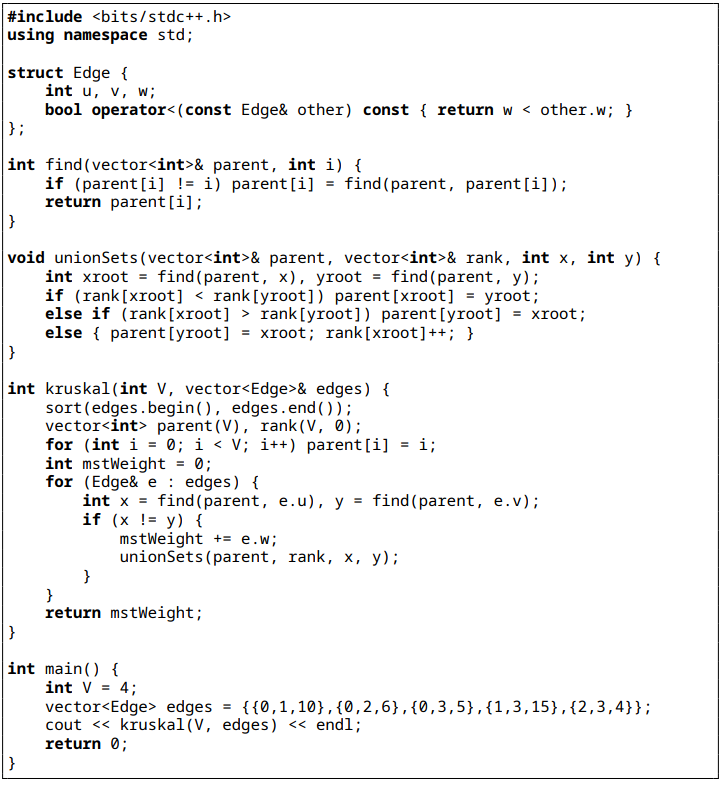
**Examples**:

* Kruskal’s Algorithm.
* Prim’s Algorithm

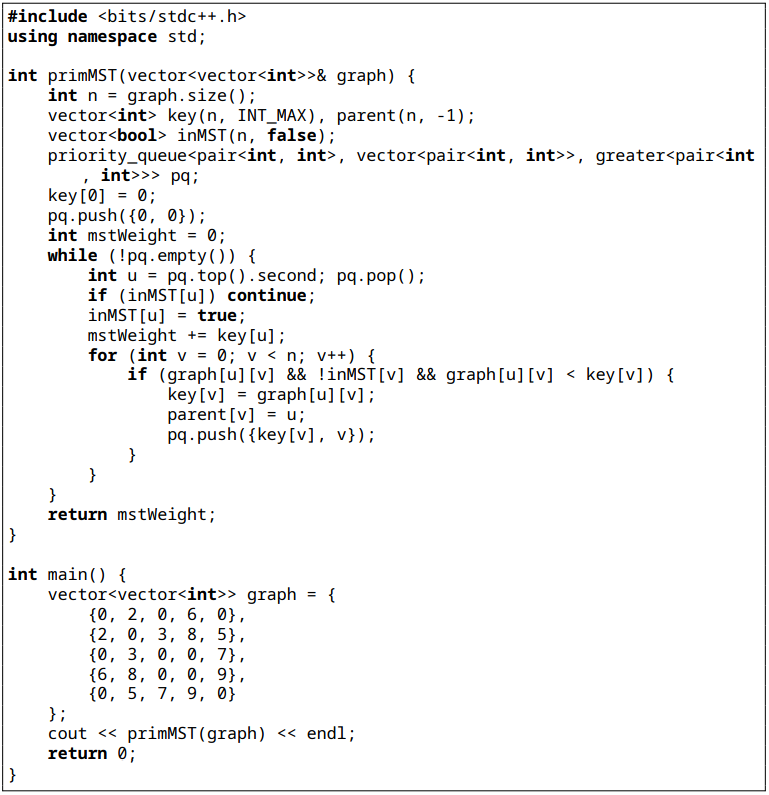
**Approach**: Use Union-Find (Kruskal) or priority queue (Prim).

Time: O(E\*log(V)), Space: O(V).

**Kruskal’s Algorithm**



**Prim’s Algorithm**

****

### **3. Trees**

Tree problems often focus on binary trees or binary search trees (BSTs).

1. **Tree Traversal**

**Description**: Visit nodes in pre-order, in-order, post-order, or level-order

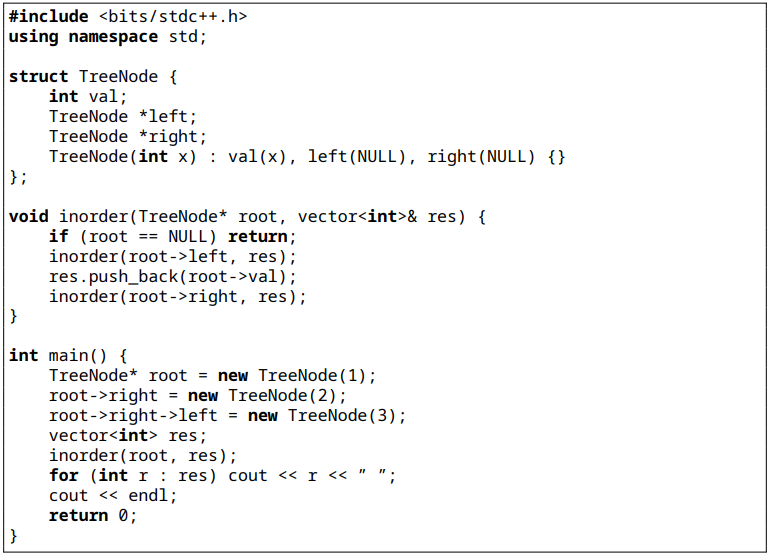
**Examples**:

* In-order traversal of a binary tree.
* Print nodes at a given level
* Level-order traversal using a queue.

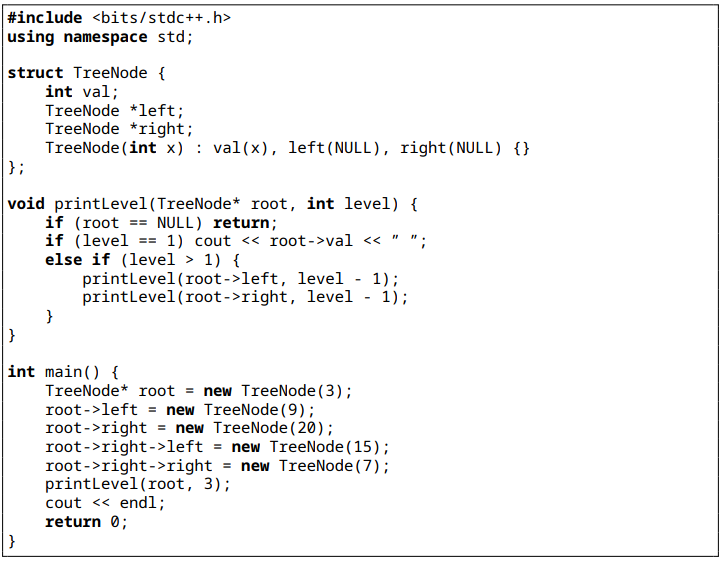
**Approach**: Use recursion or stack/queue.

Time: O(n), Space: O(h) or O(n).

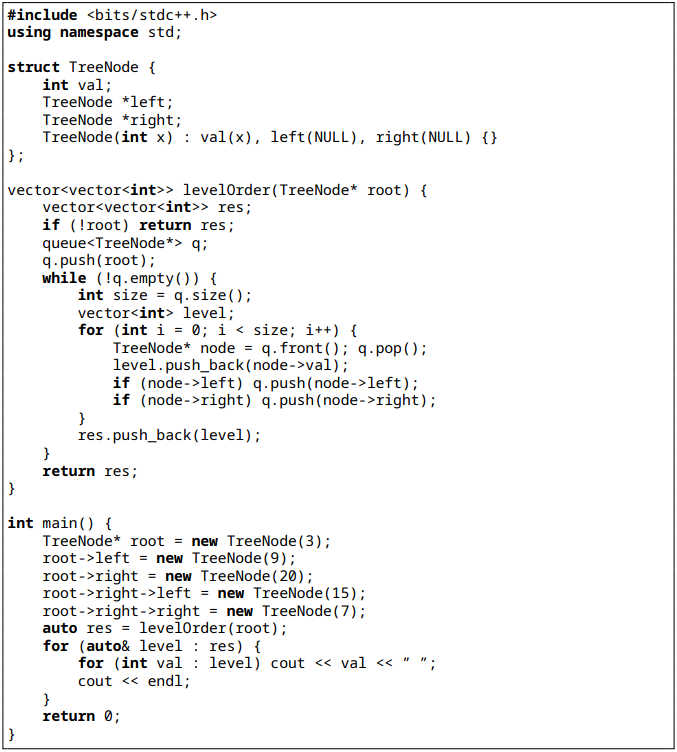
**In-order traversal of a binary tree**

****

**Print nodes at a given level.**



**Level-order traversal using a queue**

****

1. **BST Operations**

**Description**: Leverage BST properties for insertion, deletion, or validation

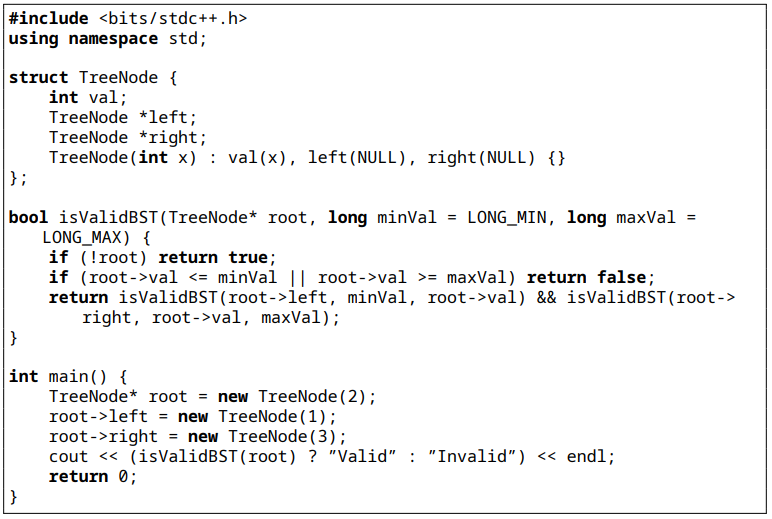
**Examples**:

* Validate a BST.
* Find kth smallest element
* Insert/delete a node in a BST.

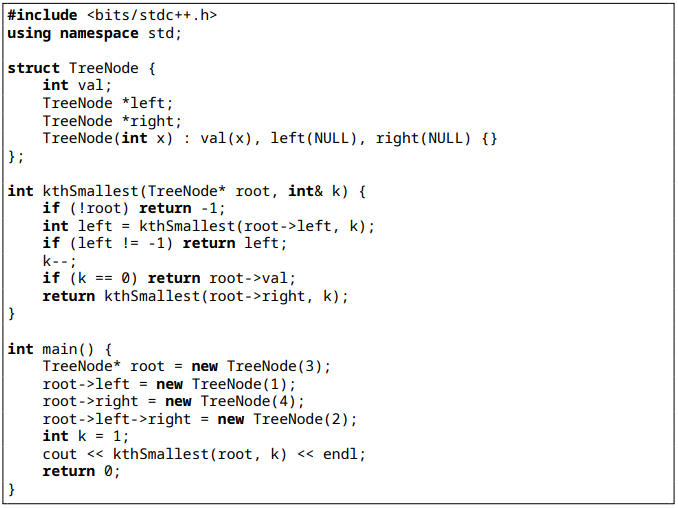
**Approach**: Use recursive range checks or in-order traversal.

Time: O(n) or O(h), Space: O(h).

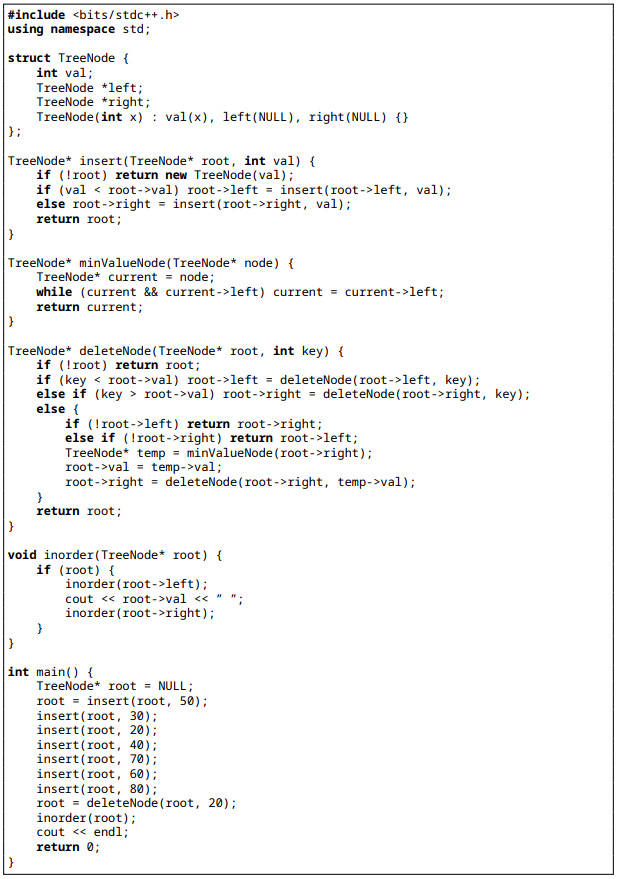
**Validate a BST**



**Find kth smallest element**



**Insert/delete a node in a BST**

****

1. **Lowest Common Ancestor (LCA)**

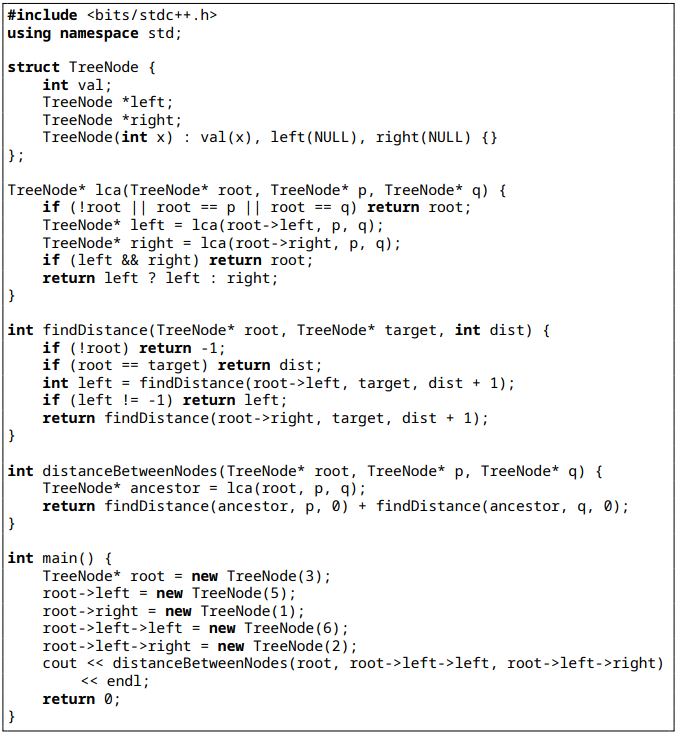
**Description**: Find the lowest node with two given nodes as descendants.

**Examples**:

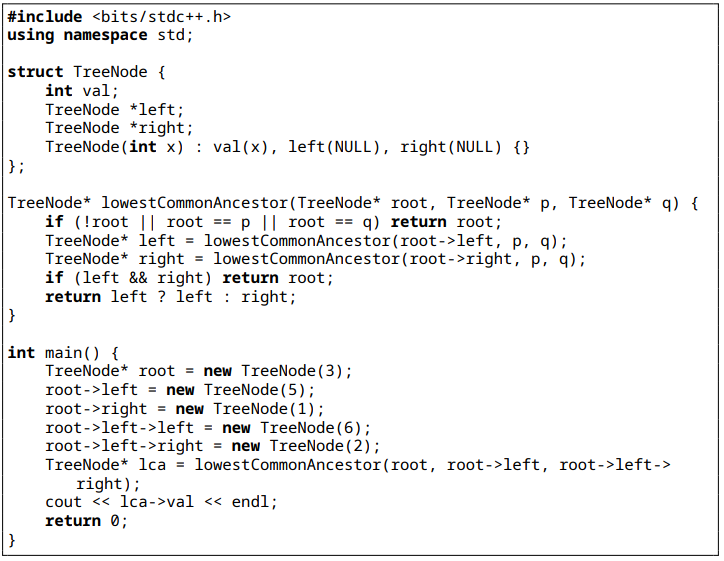
* Distance between two nodes
* LCA in a binary tree.
* LCA in a BST.

**Approach**: Recursively traverse, return LCA when nodes are in different subtrees. Time: O(n), Space: O(h).

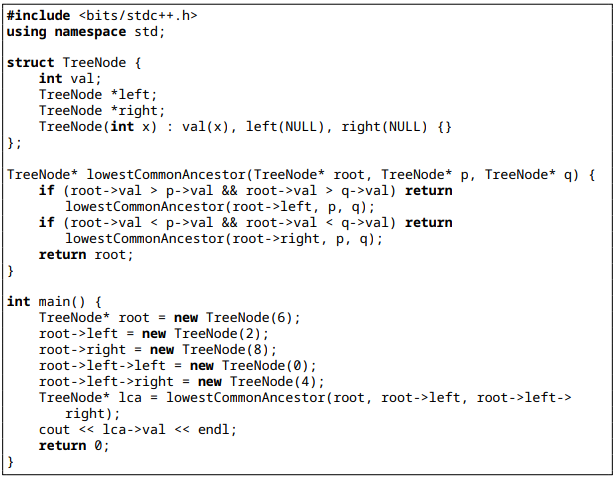
**Distance between two nodes**

****

**LCA in a binary tree**



**LCA in a BST**

****

1. **Tree Path Problems**

**Description**: Find or sum paths in a tree.

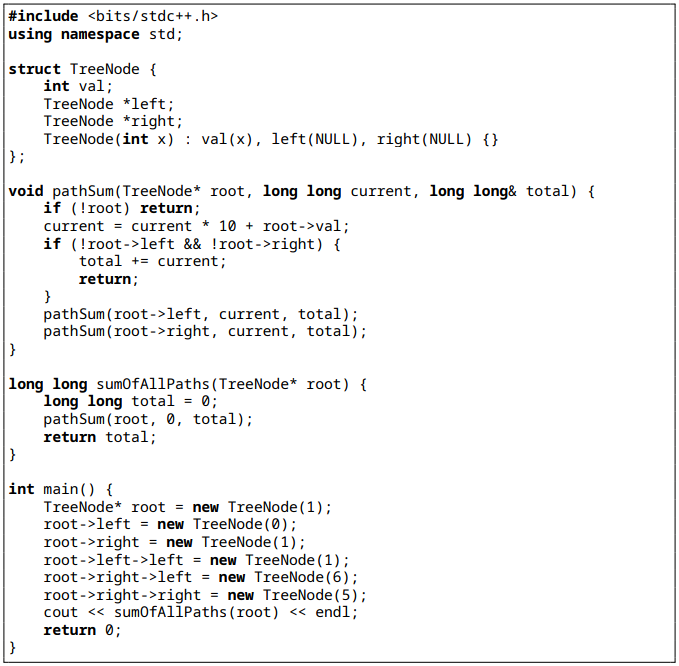
**Examples**:

* Sum of all paths
* Print all root-to-leaf paths.
* Maximum path sum.

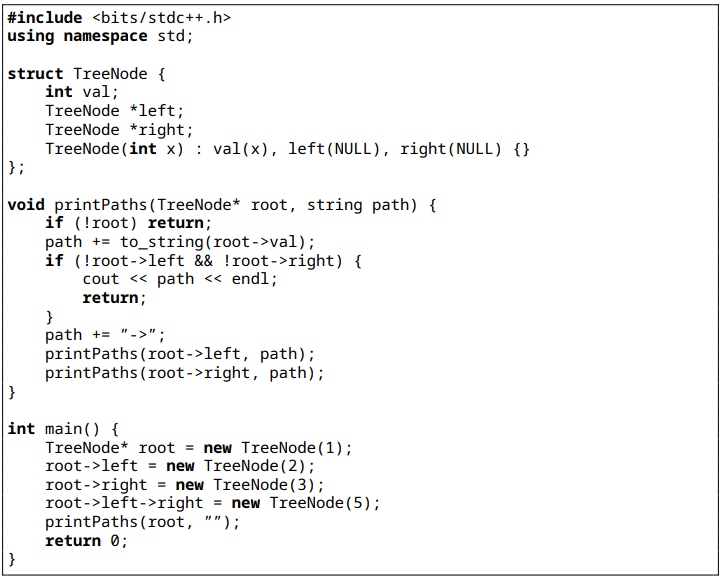
**Approach**: Use DFS with backtracking.

Time: O(n), Space: O(h).

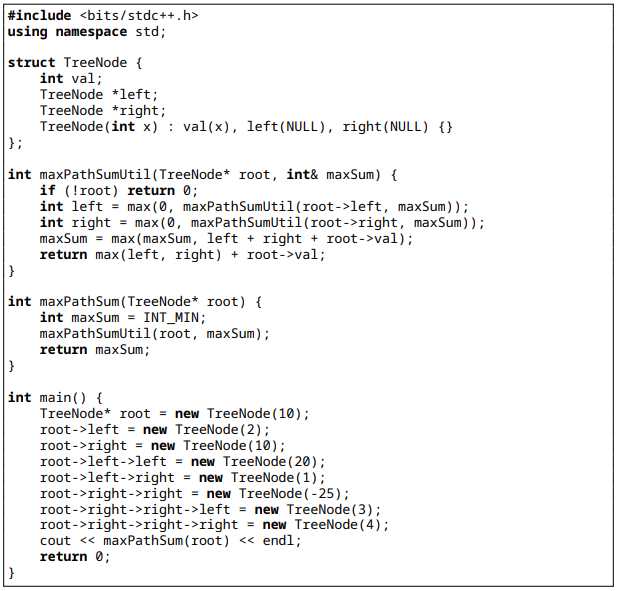
**Sum of all paths**

****

**Print all root-to-leaf paths**



**Maximum path sum**



1. **Serialize/Deserialize Binary Tree**

**Description**: Convert a binary tree to a string representation and reconstruct the tree from the string

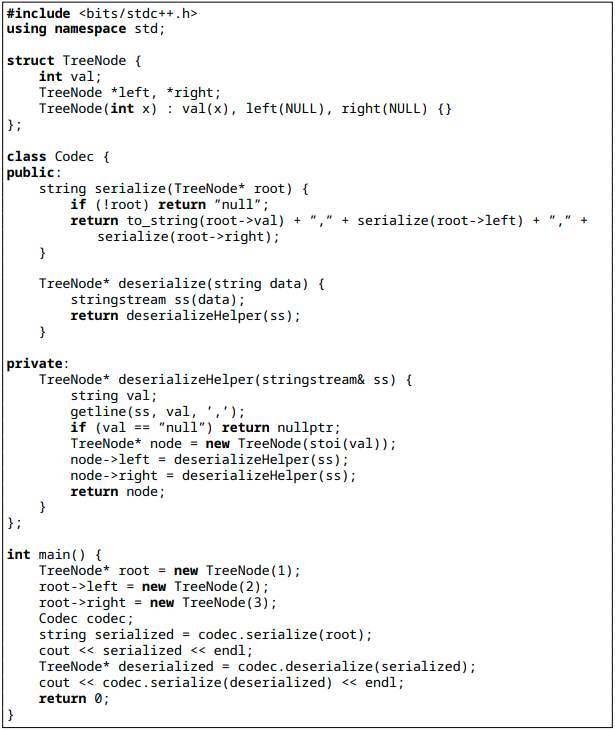
**Examples**:

* Serialize a binary tree to a string.
* Deserialize a string back to a binary tree.

**Approach**: Use preorder traversal with markers (e.g., ”null”) for serialization. Parse the string recursively for deserialization.

Time: O(n), Space: O(n).

**Serialize/Deserialize Binary Tree**

****

### **4. Arrays**

Array problems involve manipulation, searching, or optimization.

1. **Two-Pointer Technique**

**Description**: Use two pointers to solve problems like finding pairs or partitioning.

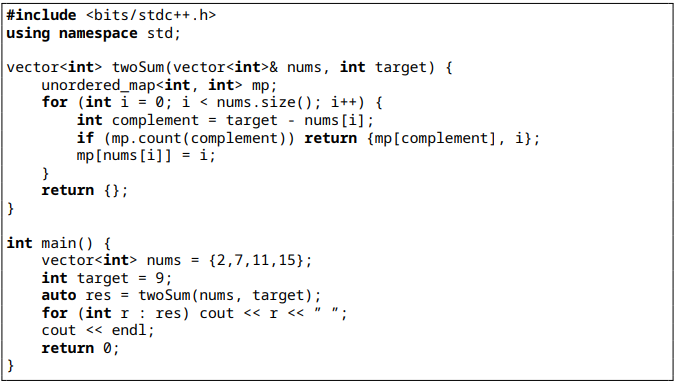
**Examples**:

* Two Sum.
* Remove duplicates from sorted array.
* Merge sorted arrays

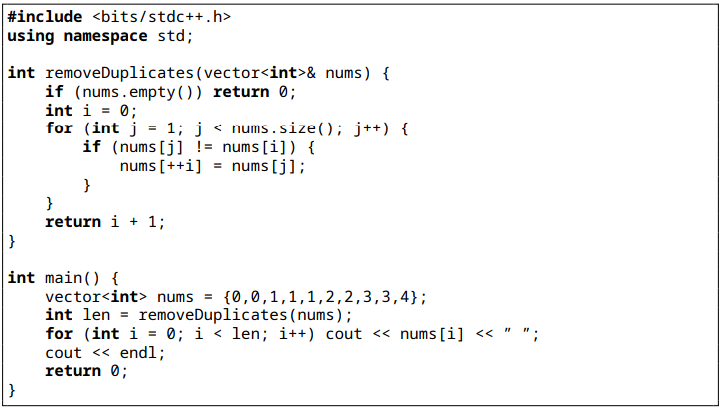
**Approach**: Move pointers based on conditions (e.g., sum comparison).

Time: O(n), Space: O(1).

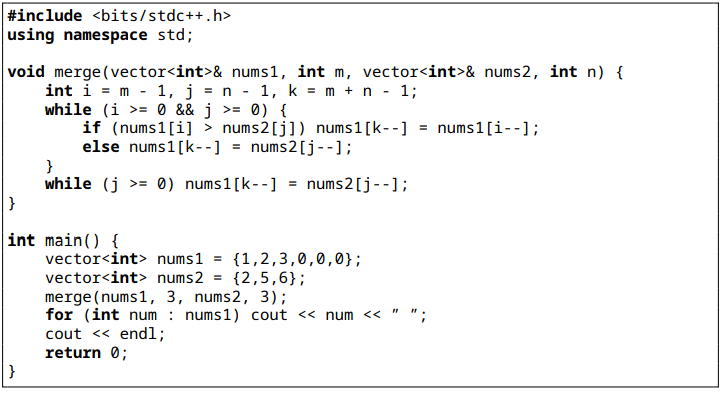
**Two Sum**

****

**Remove duplicates from sorted array**



**Merge sorted arrays**



1. **Sliding Window**

**Description**: Process a subarray of variable or fixed size.

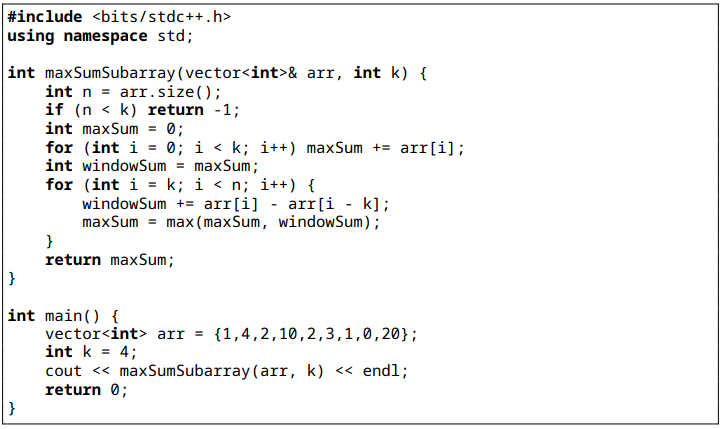
**Examples**:

* Maximum sum subarray of size k.
* Longest substring with k distinct characters.
* Minimum window substring.

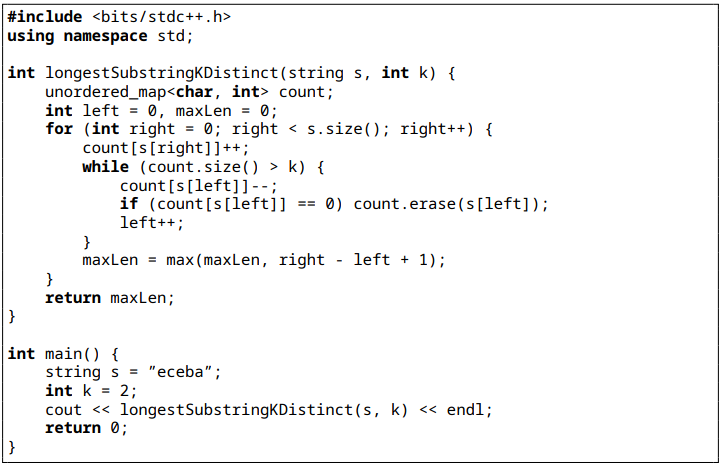
**Approach**: Adjust window boundaries while maintaining conditions.

Time: O(n), Space: O(k).

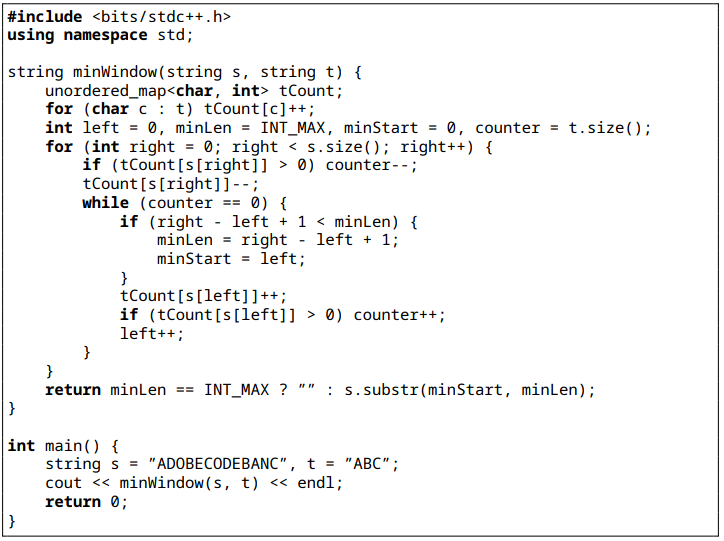
**Maximum sum subarray of size k**

****

**Longest substring with k distinct characters**

****

**Minimum window substring**

****

1. **Prefix Sum**

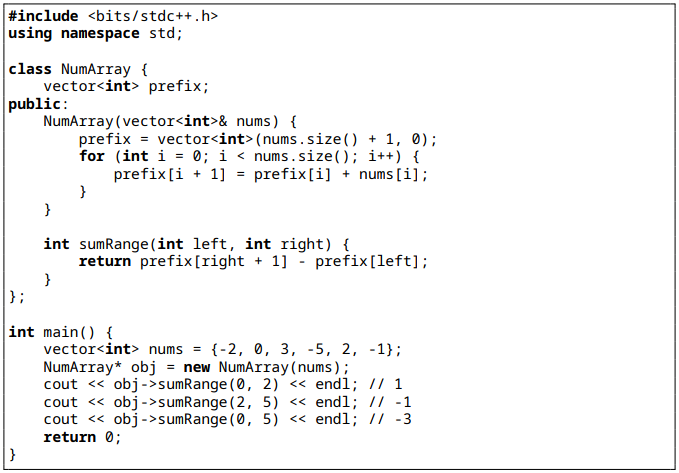
**Description**: Use cumulative sums for range queries or optimization.

**Examples**:

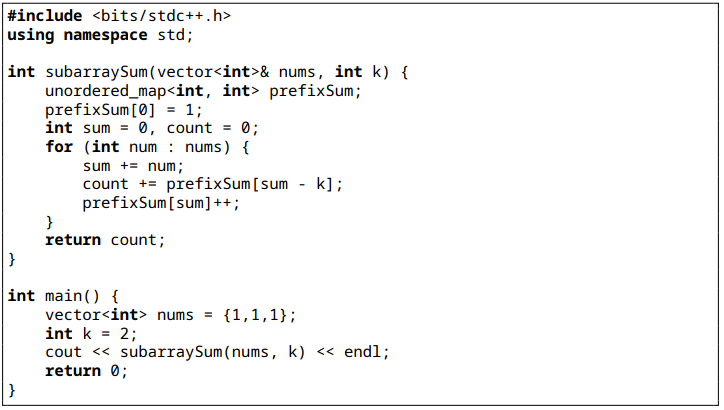
* Range sum queries.
* Subarray sum equals k.
* Maximum size subarray with given sum.

**Approach**: Build prefix sum array for O(1) range queries or use hash map for subarray sums. Time: O(n), Space: O(n).

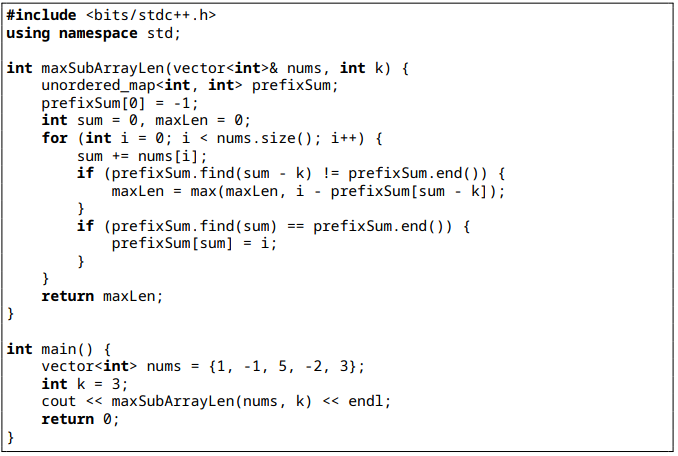
**Range sum queries**

****

**Subarray sum equals k**



**Maximum size subarray with given sum**

****

### **5. Tries**

Trie patterns focus on string processing and prefix-based queries

1. **Autocomplete System**

**Description**: Implement a system to suggest words based on a given prefix, often used in search bars.

**Examples**:

* Design an autocomplete system for a search bar.
* Find all words with a given prefix in a dictionary

**Approach**: Build a trie from the dictionary. For a prefix, traverse to the prefix node and use DFS to collect all words.

Time: O(p+m) where p is prefix length and m is total length of results, Space: O(n).

**Autocomplete System**

