OOP

* 1. Why oops? What are the advantages of Oops?
  2. What are class and object?
  3. What is the difference between struct and class?
  4. What is polymorphism in C++?
  5. Explain the constructor in C++. What are various types of constructors?
  6. What are a virtual function
  7. Compare compile time polymorphism and Runtime polymorphism
  8. What is a friend class and friend function?
  9. What are the C++ access specifiers?
  10. Define inline function
  11. What do you mean by abstraction in C++?
  12. Is destructor overloading possible? If yes then explain, and if no then why?
  13. What is an abstract class, and when do you use it?
  14. What are the static members and static member functions?
  15. What is a copy constructor?
  16. What is the difference between shallow copy and deep copy?
  17. What is the difference between virtual functions and pure virtual functions?
  18. Can static members be accessed without object? How do static member affect the size of the class?
  19. Is it always necessary to create objects from class?
  20. Diff between c struct and c++ struct
  21. What is an interface?
  22. How is an abstract class different from an interface?
  23. What is enumeration?
  24. How to virtual function work internally?
  25. What are vtable and vpointer?
  26. Diamond Problem
  27. Polymorphism and its real use cases
  28. What is enumeration
  29. Data Hiding vs Data Abstraction
  30. Encapsulation vs Abstraction

# C++

* + 1. What are void pointers?

**Void Pointer in C**

[**https://byjus.com/gate/void-pointer-in-c/#:~:text=both%20the%20same.-,The%20void%20pointer%20in%20C%20is%20a%20pointer%20that%20is,return%20void%20\*%20or%20generic%20pointers.**](https://byjus.com/gate/void-pointer-in-c/%23:~:text=both%20the%20same.-,The%20void%20pointer%20in%20C%20is%20a%20pointer%20that%20is,return%20void%20*%20or%20generic%20pointers.)

**Definition:**

A pointer in a program that isn’t associated with a data type is known as a void pointer in C. It points to some data location in the storage. It is also called the general-purpose pointer.

**Detail:**

As the name suggests, the void pointer indicates that the pointer is basically empty- and thus, it can hold any data type address in the program. For example when we are declaring the float pointer, it will only be able to point to the float type variable. We thus use a pointer to void to overcome this very problem.

The pointer to void can be used in generic functions in C because it is capable of pointing to any data type. One can assign the void pointer with any data type’s address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting. So, it reduces complications in a code.

**Syntax:** Void \*ptr

**Code:**

#include<stdlib.h>

int main() {

int v = 7; // created a int variable

float w = 7.6; // created a float variable

void \*u; // void ptr is made

u = &v; //void pointer points to location of v (int)

printf(“The Integer variable in the program is equal to = %d”, \*( (int\*) u) ); //dereference , print

u = &w; //void pointer points to location of v (int)

printf(“\nThe Float variable in the program is equal to = %f”, \*( (float\*) u) ); //dereference , print

return 0;

}

The output obtained out of the program would be:

The Integer variable in the program is equal to = 7

The Float variable in the program is equal to = 7.600000

void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer. \*( (int\*) u

Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

### **Is there any difference between the null pointer and the void pointer in C?**

The null pointer is basically used in a program to assign the value 0 to a pointer variable of any data type. The void pointer, on the other hand, has no value assigned to it and we use it to store the addresses of other variables in the program- irrespective of their data types.

### **Why do we use a void pointer in C programs?**

We use the void pointers to overcome the issue of assigning separate values to different data types in a program. The pointer to void can be used in generic functions in C because it is capable of pointing to any data type. One can assign the void pointer with any data type’s address, and then assign the void pointer to any pointer without even performing some sort of explicit typecasting. So, it reduces complications in a code.

[**Wild pointer**](https://www.geeksforgeeks.org/what-are-wild-pointers-how-can-we-avoid/)

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

**int** main()

{

**int** \*p;  /\* wild pointer \*/

**int** x = 10;

    // p is not a wild pointer now

    p = &x;

**return** 0;

}

[**NULL Pointer**](https://www.geeksforgeeks.org/few-bytes-on-null-pointer-in-c/)

NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  **int** main()  {  **int** \*ptr = NULL;   // Null Pointer      cout << "The value of " << ptr;  **return** 0;  } |

* + 1. What is size of a pointer?

The size of a pointer in C/C++ is not fixed. It depends upon different issues like Operating system, CPU architecture etc.

Usually, it depends upon the word size of underlying processor for example for a 32-bit computer the pointer size can be 4 bytes for a 64-bit computer the pointer size can be 8 bytes. So, for a specific architecture pointer size will be fixed.

It is common to all data types like int \*, float \* etc.

**The sizeof() operator can be used to find the size of data types and expressions.**

The general syntax of using it is: *sizeof(data/data type/expression)*

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **16 bit** | **32 bit** | **64 bit** |
| **char** | 1 byte | 1 byte | 1 byte |
| **short** | 1 byte | 2 bytes | 2 bytes |
| **int** | 2 bytes | 4 bytes | 4 bytes |
| **long** | 4 bytes | 4 bytes | 8 bytes |
| **float** | 4 bytes | 4 bytes | 4 bytes |
| **double** | 8 bytes | 8 bytes | 8 bytes |

### What is the difference between a pointer to an array and an array of pointers?

A pointer to an array is a variable that **stores the address of the first cell in the array**.

An array of pointers is an array whose **elements are pointer-type variables**.

### To find the number of elements in an array:

There is an alternative method to find the length of an array rather than with a counter by running a loop.

Let us finally see what that method is:

#include <stdio.h>

int main()

{

int arr[] = { 1, 2, 3, 4, 5};

printf("The number of elements is %d ",sizeof(arr)/sizeof(arr[0]));

return 0;

}

Output:

The number of elements is 5

* + 1. What is meant by exception handling

**Or**

How to identify errors/exceptions in code?

One of the advantages of C++ over C is **Exception Handling**. Exceptions are runtime anomalies that a program encounters during its execution.

There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program’s control, such as disc failure, keyboard interrupts etc.)

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

**C++ try and catch:**

Exception handling in C++ consists of three keywords: try, throw, and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The **try and catch keywords come in pairs:**

Example:

We use the try block to test some code: If the value of a variable “age” is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that’s why we are throwing an exception of type int in the try block (age), we can pass “int myNum” as the parameter to the catch statement, where the variable “myNum” is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

**int** main()

{

**int** x = -1;

   // Some code

   cout << "Before try \n";

**try**

{

      cout << "Inside try \n";

**if** (x < 0)

{

**throw** x;

         cout << "After throw (Never executed) \n";}

    }

**catch** (**int** x )

{cout << "Exception Caught \n";}

   cout << "After catch (Will be executed) \n";

**return** 0;

}

|  |
| --- |
| **Output:**  Before try  Inside try  Exception Caught  After catch (Will be executed) |

* + 1. What is an overflow error?

A [data type overflow error](http://www.computerhope.com/jargon/o/overflow.htm) is when the data type used to store data was not large enough to hold the data.

**The integer overflow** occurs when a number is greater than the maximum value the data type can hold.

**The integer underflow** occurs when a number is smaller than the data type’s minimum value.

#include <iostream>

**using** **namespace** std;

// Driver code

**int** main()

{

**int** a = 100000;

**int** b = 100000;

**int** c = a \* b;

    cout << "The product of a and b is " << c << endl;

**return** 0;

}

The product of a and b is 1410065408

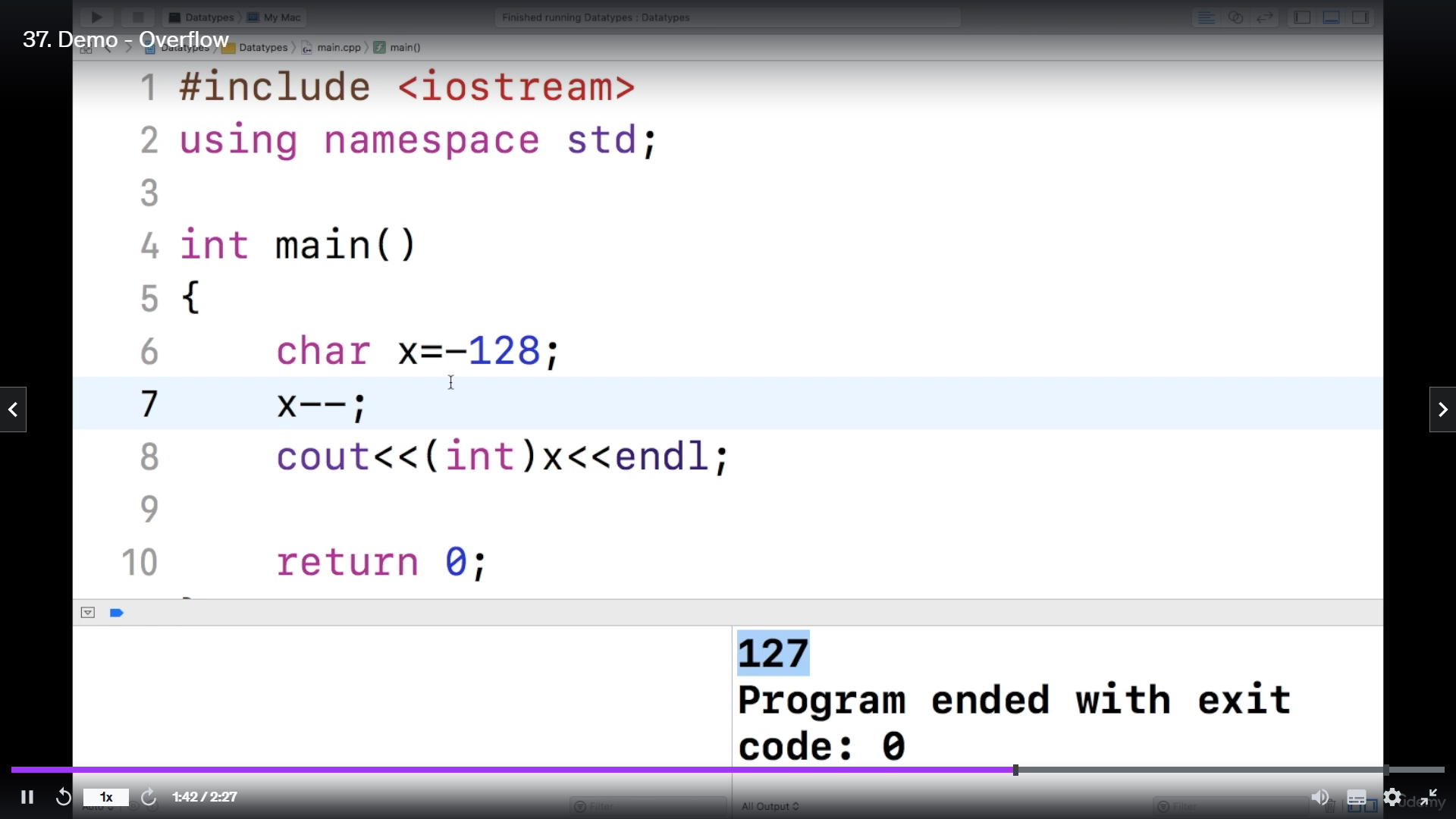
**Explanation:**The expected value of **c**is **1010** but the output is **1410065408**. This is because **int** **c** can store a maximum range of **109**.

* Initialize **a** or **b** as **long long** data types. Since multiplication of **int** and **long long** is **long long**. So, **a** and **b** will result in a **long long** range.
* Instead of changing the data types of **a** and **b**, we can multiply **a** and **b** with **1LL**while initializing the value of **c** so that multiplication of **a** and **b** with **long long 1** also results into **long long**and that value will be stored in **long long c.**

**When the value is more than the capacity it will take the values again from the beginning this concept is called overflow**

**• Suppose there is a byte and we are already having values in it , if 1 is added to this max byte value it becomes a signed bit therefor it become -128 magically this happens when we try going to the next value**

**Cyclic value**



* + 1. Implementation of Vectors

**Definition/Overview:**

Vectors are one of the containers offered to us by the STL(Standard Template Library) in C++. Vectors are containers representing arrays used to store elements of the same data type placed in contiguous memory locations, but the size of a vector grows and shrinks dynamically, that is, it can change its size during runtime. The elements of a vectors can be accessed and traversed using iterators.

Basically, this array is controlled by the vector class and as soon as it goes out of bounds, the copy constructor copies it to another array of a bigger size. This bigger array is later used as our main internal array. Since internally copy is being created which takes time so it is recommended to give size if known while initializing it.

We are provided the vector template by these header files –

#include<vector> *//header file to implement vectors*

#include<bits/stdc++.h> *//header file that provides all STL containers and functions*

Since both files provide us with the vector class for implementation on their own. Hence we need to include only one of the above header files to use vectors.

Syntax:

### Syntax:

vector<Data\_Type> vector\_name; *// initialisation*

vector<Data\_Type> vector\_name(size); *// initialisation with size*

vector<Data\_Type> vector\_name(size, value); *// initialisation with size and value*

|  |  |
| --- | --- |
| Array does not offer us any iterators so we have to depend on loops for traversal. | Vectors offer iterators which enable us to perform different types of traversals. |
| Elements are accessed using [] operator | Elements are accessed using \* operator |

#### code for traversing array

int b[5] = { 11, 22, 33, 44, 55 };

for (int i = 0; i < 5; i++)

cout << b[i] << " "; *// 11 22 33 44 55*

#### code for traversing Vector

|  |
| --- |
| //header files  #include <iostream>  #include <vector>  #include <stdio.h>  using namespace std;  int main() {    vector<int> myVector = {1, 2, 3}; // create a vector with some elements  // print the contents of the vector using an iterator  cout << "The contents of the vector are: ";  for (auto it = myVector.begin(); it != myVector.end(); ++it) {  cout << \*it << " ";  }  cout << std::endl;  // add an element to the vector using an iterator  auto it = myVector.insert(myVector.begin() + 1, 4);  // print the contents of the vector again  cout << "After inserting an element, the vector contains: ";  for (auto it = myVector.begin(); it != myVector.end(); ++it) {  cout << \*it << " ";  }  cout << endl;  //  myVector.erase(it);  // print the contents of the vector one more time  cout << "After erasing an element, the vector contains: ";  for (auto it = myVector.begin(); it != myVector.end(); ++it) {  cout << \*it << " ";  }  cout << endl;  return 0;  } |

* + 1. Difference between C and C++

| **C** | **C++** |
| --- | --- |
| C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs. | C++ was developed by Bjarne Stroustrup in 1979. |
| C supports [procedural programming](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/), C does not support oops concepts like polymorphism, encapsulation, and inheritance | C++ is known as hybrid language because C++ supports both [procedural](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) and [oops](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) concepts like [polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/), [encapsulation](https://www.geeksforgeeks.org/encapsulation-in-c/), and [inheritance](https://www.geeksforgeeks.org/inheritance-in-c/) |
| C is a function-driven language. | C++ is an object-driven language |
| Virtual and friend functions are not supported by C. | [Virtual](https://www.geeksforgeeks.org/virtual-function-cpp/) and [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are supported by C++. |
| C provides [malloc()](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/) and [calloc()](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/) functions for [dynamic memory allocation](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/), and [free()](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/) for memory de-allocation. | C++ provides [new operator](https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/) for memory allocation and [delete operator](https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/) for memory de-allocation. |
| Direct support for exception handling is not supported by C. | [Exception handling](https://www.geeksforgeeks.org/exception-handling-c/) is supported by C++. |
| C structures don’t have access modifiers. | C ++ structures have access modifiers. |
| C follows the top-down approach | C++ follows the Bottom-up approach |

* + 1. dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. In this case, the pointer is pointing to the memory, which is de-allocated. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Dangling Pointers are generated when we do not modify the value of a pointer after deallocation of a memory block or when a variable goes out of scope.

**Different ways where pointers act as Dangling Pointers in C**

There are three different ways in which a *pointer* can act as a dangling pointer in C :

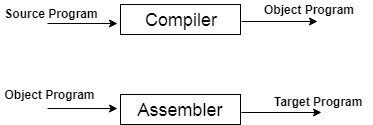
1. Deallocation of memory
2. Function Call
3. Variable goes out of scope

If we declare a variable inside a function, then that variable will be local to that function execution and cannot be accessed outside that function's scope. Now, suppose main() function's pointer stores the address of that local variable inside the function, this way we can access address of that local variable as long as the function is executing, but once the function execution gets over, all internal variables goes to garbage collection and are not in memory anymore, but main() function's pointer is still pointing to that particular address which is now not available in memory, hence creating a dangling condition and would be called as a Dangling Pointer.

1. A function() is called inside the main() function, memory is allocated by the system for the function() block.
2. A local variable temp is declared and initialized inside the function(). Let the address of temp is 2000. After returning the address of the temp variable function execution finishes and temp also gets deleted from the memory.
3. Returned address 2000 is stored in ptr pointer but as temp is not there in the memory anymore, ptr points to some garbage value and acts as a *dangling pointer*.
4. #include <stdio.h>
5. *// definition of danglingPointer() function*
6. int \*danglingPointer() {
7. *// temp variable has local scope*
8. int temp = 10;
9. *// returning address of temp variable*
10. return &temp;
11. }
12. int main() {
13. *// ptr will point to some garbage value*
14. *// as temp variable will be destroyed*
15. *// after the execution of below line*
16. int \*ptr = danglingPointer();
17. *// ptr is a Dangling Pointer now*
18. *// ptr contains some random address and*
19. *// is pointing to some garbage value*
20. printf("%d", \*ptr);
21. return 0;
22. }
    * 1. Arrays vs Vectors
23. **Resizing:** Vectors can dynamically resize themselves, while arrays have a fixed size.
24. **Memory allocation:** Vectors handle memory allocation and deallocation automatically, while arrays require manual management.
25. **Element access:** Vectors provide direct access to elements using an iterator, while arrays require using an index.
26. **Sorting:** Vectors provide built-in sorting functions, while arrays require manual sorting.
27. **Insertion and deletion:** Vectors provide efficient insertions and deletions at the end, while arrays require shifting of elements.
28. **STL compatibility:** Vectors are a part of the Standard Template Library (STL) and are compatible with other STL functions and algorithms.
29. **Exception handling:** Vectors have built-in exception handling for out-of-bounds access, while arrays do not.
30. **Memory utilization:** Vector can save memory space if the number of elements increases after the array created by using dynamic memory allocation.
31. **Iterators:** Vectors provide iterators that make it easy to traverse the elements, whereas arrays require manual pointer manipulation.
    * 1. Linear Ds, Non-Linear DS

| **S.NO** | **Linear Data Structure** | **Non-linear Data Structure** |
| --- | --- | --- |
| 1. | In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent. | In a non-linear data structure, data elements are attached in hierarchically manner. |
| 2. | In linear data structure, single level is involved. | Whereas in non-linear data structure, multiple levels are involved. |
| 3. | Its implementation is easy in comparison to non-linear data structure. | While its implementation is complex in comparison to linear data structure. |
| 4. | In linear data structure, data elements can be traversed in a single run only. | While in non-linear data structure, data elements can’t be traversed in a single run only. |
| 5. | Its examples are: array, stack, queue, linked list, etc. | While its examples are: trees and graphs. |
| 7. | Applications of linear data structures are mainly in application software development. | Applications of non-linear data structures are in Artificial Intelligence and image processing |

* + 1. What is compiler
* A compiler is a translator that converts the high-level language into the machine language.
* High-level language is written by a developer and machine language can be understood by the processor.
* Compiler is used to show errors to the programmer.
* The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
* When you execute a program which is written in HLL programming language then it executes into two parts.
* In the first part, the source program compiled and translated into the object program (low level language).
* In the second part, object program translated into the target program through the assembler.



**Fig: Execution process of source program in Compiler**

* + 1. Difference between switch and if else

Both If-else and Switch both are conditional statements in programming

| **Parameter** | **If-else** | **Switch** |
| --- | --- | --- |
| Definition | The if and else blocks are executed depending on the condition in the if statement | The switch statement has multiple cases, and the code block corresponding to that case is executed |
| Evaluation | Used for integer, character, pointer, floating-point type, or Boolean type. | Used for character expressions and integers. |
| Expression | Multiple statements for multiple decisions | Single statements for multiple decisions |
| Default Execution | If the condition inside the if-statement is false, then the code block under the else condition is executed | If the condition inside switch statements does not match any of the cases, the default statement is executed. |
| Sequence of Execution | Either the code block in the if statement is executed or the code block in the else statement. | The switch case statement performs each case until a break statement is encountered or the end of the switch statement is reached. |
| Speed | If you use 'if-else' to implement several options, the speed will be slow. | The switch statement is the best solution if we have numerous options because it executes considerably faster than the 'if-else' statement. |

#include<iostream>

using namespace std;

int main()

{

int x = 20;

switch (x)

{

case 10:

cout<<"X is 10"; break;

case 20:

cout << "X is 20"; break;

case 30:

cout << "X is 30"; break;

default:

cout<<"X is not 10, 20 or 30"; break;

}

return 0;

}

**Output:** C++ Switch Case Statement

* + 1. explain a real life example in which a non-linear data structure is used

Linear

Arrays :-

1. [2D arrays](https://www.geeksforgeeks.org/multidimensional-arrays-in-java/), commonly known as, matrices, are used in image processing.
2. Online ticket booking.
3. Contacts on a cell phone.
4. For CPU scheduling in computer.

Strings : -

1. Spam email detection.
2. Plagiarism detection.
3. Search engine.
4. Digital forensic and information retrieval system
5. Spell checkers.
6. In the database to check valid information of the user

Linked List : -

1. Web pages can be accessed using the previous and the next URL links which are linked using a linked list.
2. The music players also use the same technique to switch between music.
3. Social media content “feeds”.
4. Used in switching between applications and programs (Alt + Tab) in the Operating system (implemented using Circular Linked List)
5. It can be used to implement Stacks, Queues, Graphs, and Trees.

Stacks :-

Some Applications of a stack are:

1. Converting infix to postfix expressions.
2. Undo/Redo button/operation in word processors.
3. Forward-backward surfing in the browser.
4. Recursion.
5. Used in IDEs to check for proper parentheses matching
6. Media playlist. T o play previous and next song

Queue:-

1. [Operating System](https://www.geeksforgeeks.org/types-of-operating-systems/) uses queues for job scheduling.
2. To handle congestion in the networking queue can be used.
3. Data packets in communication are arranged in queue format.
4. Sending an e-mail, it will be queued.
5. Server while responding to request

Non Linear :-

Graphs :-

1. The GPS navigation system also uses shortest path APIs.- Dijkstra’s algorithm 🡪 shortest path
2. Google map to find nearest location.
3. Facebook, Instagram, and all social media networking sites every user is Node
4. Data organization

Trees:- hierarchical structures having a single root node.

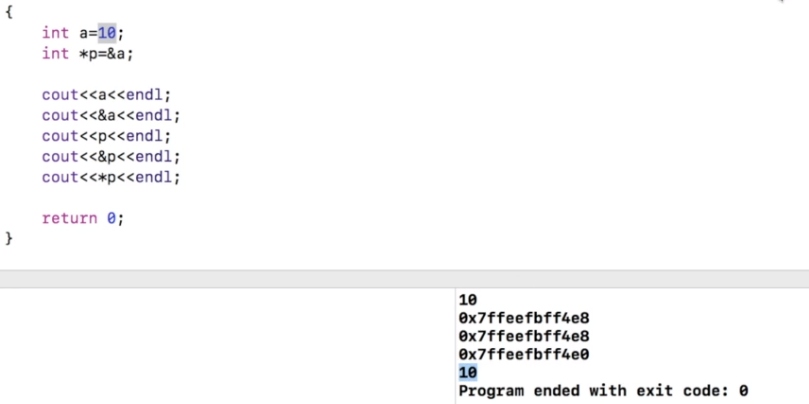
Some applications of the trees are: a

1. The decision-based algorithm is used in machine learning which works upon the algorithm of the tree.
2. File explorer/my computer of mobile/any computer
3. Posting questions on websites like Quora, the comments are a child of questions.
4. To store the possible moves in a chess game.

Pointers: -

3 major steps-

* + 1. Declaration - int \*p;
    2. Initialization - p=&x
    3. Dereferencing - cout<<\*p



* + 1. what is memory leak

Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leakage occurs in C++ when programmers allocates memory by using [new keyword](https://www.geeksforgeeks.org/new-vs-operator-new-in-cpp/) and forgets to deallocate the memory by using delete() function

Allocate memory by new keyword and deallocate memory by delete keyword and write all code between them.

**Example to handle memory leaks**

|  |
| --- |
| #include <bits/stdc++.h>  **using** **namespace** std;  // function to see memory handling  **void** func\_to\_handle\_mem\_leak()  {  **int**\* ptr = **new** **int**(5);      // body      // Now delete pointer ptr using delete  **delete** (ptr);  }  // Driver code  **int** main()  {        // Call function to handle      // the memory leak      func\_to\_handle\_mem\_leak()  **return** 0;  } |

Therefore, **Always write delete pointer for matching of new pointer in C++**and always write code between these new and delete as explained in above example. In above example, no memory is wasted because when we are coming out from the function we are deallocating the memory by using delete function.

The consequences of memory leak is that it reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.

* + 1. What is Hashing? How to resolve collision in hashing?

*Hashing is storing and Retrieving data in O(1) time.*

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word “**efficiency**“. Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

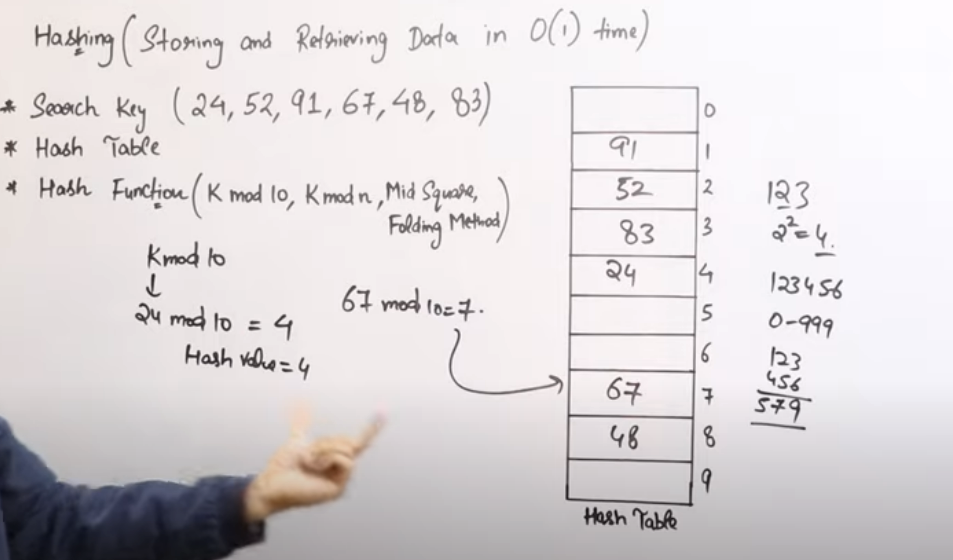
## Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:**The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the**hash index**.

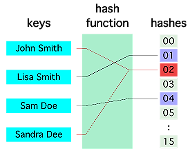
**Hash function** 🡪 kmod10 , kmodn, folding method

1. **Hash Table:**Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data associatively in an array where each data value has its unique index.



**Collision:**

Situation of collision occurs when more than one keys (hash functions) map to the same location of hashes. In this situation, two or more data elements qualify to be mapped to the same location in hash table.



Collision resolution can be done using two techniques:

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,

Elements = 12, 15, 22, 25 and 37.

Let’s see step by step approach to how to solve the above problem:

|  |  |
| --- | --- |
| **Step 1:**First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided. | Hash table |
| **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.  *Insert 12 into hash table* | Insert 12 into hash table |
| **Step 3:** Now the next key is 22.  It will map to bucket number 2 because 22%5=2.  But bucket 2 is already occupied by key 12.  *Insert 22 into hash table* | Insert 15 into hash table |
| **Step 4:** The next key is 15. It will map to slot number 0 because 15%5=0.  **Step 5:** Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.  *Insert 25 into hash table* | *Insert 15 into hash table*  Insert 25 into hash table |

Hence In this way, the separate chaining method is used as the collision resolution technique.

### 2) [**Open Addressing**](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/?ref=lbp)

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

### 2.a) Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

**Algorithm:**

1. *Calculate the hash key. i.e.****key = data % size***
2. *Check, if****hashTable[key]****is empty*
   * *store the value directly by****hashTable[key] = data***
3. *If the hash index already has some value then*
   * *check for next index using****key = (key+1) % size***
4. *Check, if the next index is available hashTable[key] then store the value. Otherwise try for next index.*
5. *Do the above process till we find the space.*

**Example:** Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

|  |  |
| --- | --- |
| **Step1:**First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided | Hash table |
| **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.  *Insert 50 into hash table* | Insert 50 into hash table |
| **Step 3:**The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.  *Insert 70 into hash table* | Insert 70 into hash table |
| **Step 4:**The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.  *Insert 76 into hash table* | Insert 76 into hash table |
| **step 5:**The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.  *Insert 93 into hash table* | Insert 93 into hash table |

* + 1. what is Asymptotic analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

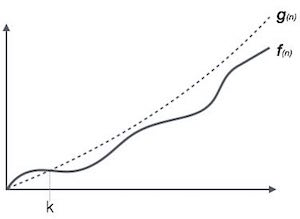
## **Asymptotic Notations**

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

### **Big Oh Notation, Ο**

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.



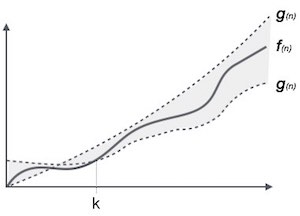
### **Omega Notation, Ω**

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



### **Theta Notation, θ**

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



* + 1. DS used for STL implementation
    2. Where does static and global variable reside

## **Global variables** are those which are not defined inside any function and have a global scope whereas **local variables** are those which are defined inside a function and its scope is limited to that function only. In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function. **Local variables**are those which are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function.

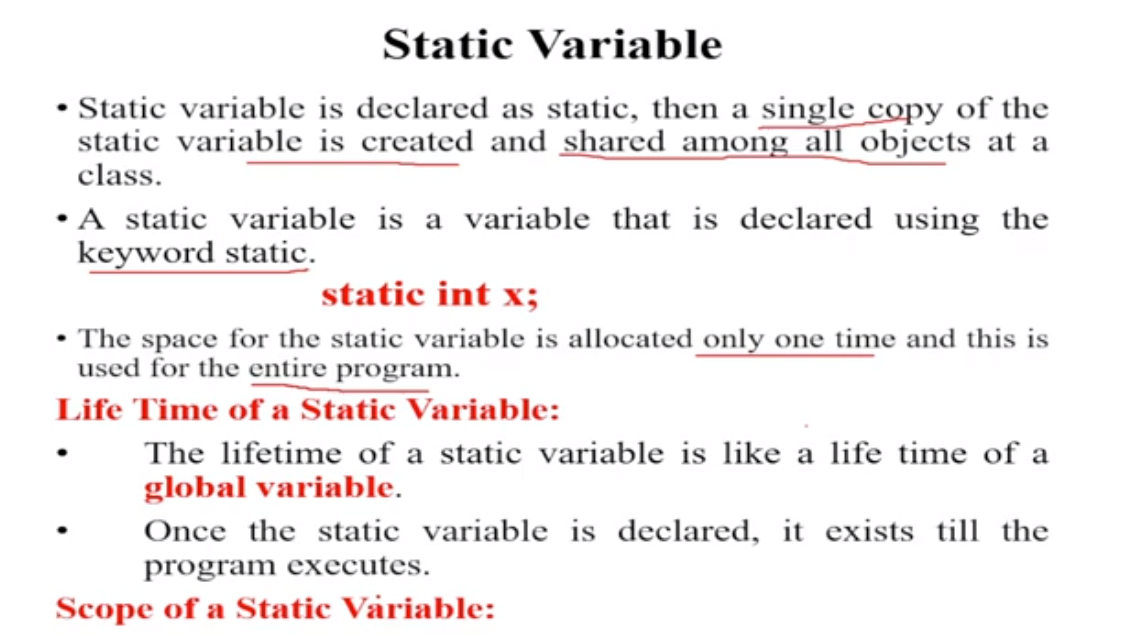
**Static variables** have a property of preserving their value even after they are out of their scope! Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope. 

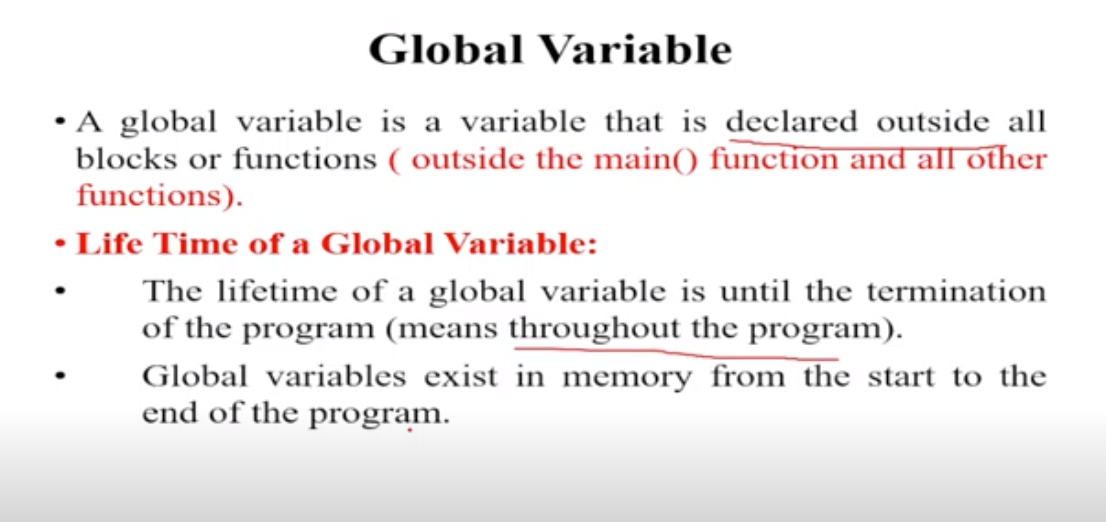
Syntax: **static data\_type var\_name = var\_value;**

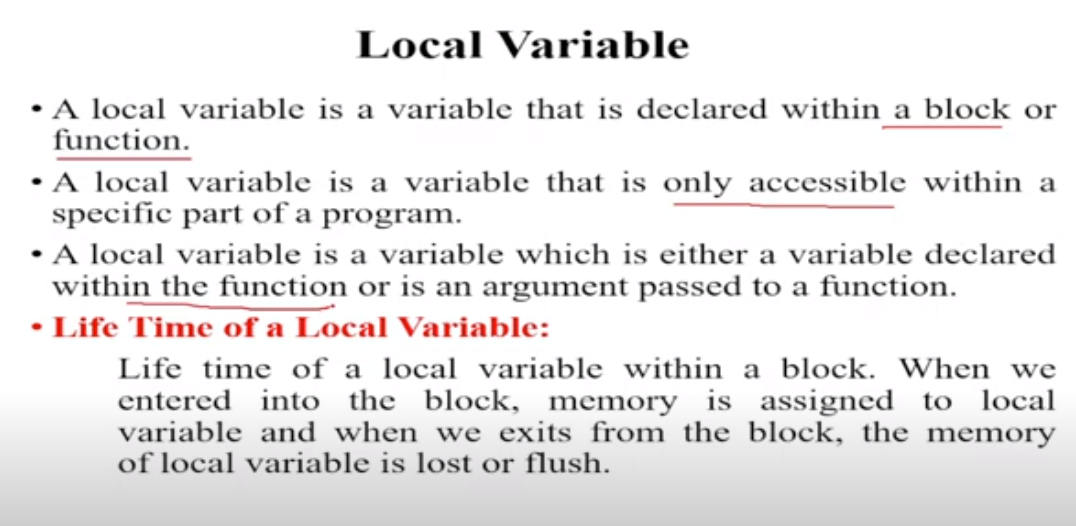
* Following are some interesting facts about static variables in C.  
  A static int variable remains in memory while the program is running. A normal variable is destroyed when a function call where the variable was declared is over.   
  For example, we can use static int to count a number of times a function is called, but an auto variable can’t be used for this purpose.
* **Static variables are allocated memory in data segment,** not stack segment.   
  Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, the value of x is printed as 0, while the value of y is garbage.

|  |
| --- |
| #include <stdio.h>  **int** main()  {  **static** **int** x;  **int** y;  **printf**("%d \n %d", x, y);  } |

Output: 0 , [some\_garbage\_value]







* + 1. Memory layout of c

Text segment

Initialized data segment

Uninitialized data segment

Stack

Heap

**Let's understand each section one by one.**

### **1. Text segment**

The text segment is also known as the code segment. When we compile any program, it creates an executable file like a.out, .exe, etc., that gets stored in the text or code section of the RAM memory.

### **2. Data section**

The data which we use in our program will be stored in the data section. Since the variables declared inside the main() function are stored in the stack, but the variables declared outside the main() method will be stored in the data section. The variables declared in the data section could be stored in the form of initialized, uninitialized, and it could be local or global. Therefore, the data section is divided into four categories, i.e., initialized, uninitialized, global, or local.

**Let's understand this scenario through an example.**

1. #include<stdio.h>
2. **int** var1;
3. **int** var2 = 10;
4. **void** function1()
5. {
6. printf("I am function1");
7. }
8. **int** main()
9. {
10. function1();
11. **return** 0;
12. }

In the above code, var1 and var2 variables are declared outside the main() function where var1 is the uninitialized variable, whereas the var2 is an initialized variable. These variables can be accessed anywhere in the program because these variables are not a part of the main() in the stack.

**The data section consists of two segments:**

* Uninitialized data segment
* Initialized data segment

**Uninitialized data segment**

The uninitialized data segment is also known as a **.bss** segment that stores all the uninitialized global, local and external variables. If the global, static and external variables are not initialized, they are assigned with zero value by default.

The .bss segment stands for **Block Started by symbol**. The bss segment contains the object file where all the statically allocated variables are stored. Here, statically allocated objects are those objects without explicit initialization are initialized with zero value. In the above code, var1 is an uninitialized variable so it is stored in the uninitialized data section.

### **3. Stack**

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. The function arguments are also stored in the function as the arguments are also a part of the function. Such a type of memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time. The stack section plays a very important role in the memory because whenever the function is called, a new stack frame is created.

Stack is also used for recursive functions. When the function is called itself again and again inside the same function which causes the stack overflow condition and it leads to the segmentation fault error in the program.

### **4. Heap**

Heap memory is used for the dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap.

* + 1. Steps involved in execution of a C program
    2. Storage classes in C
    3. difference between macro and inline function
    4. distinction between structure and union
    5. How to detect if stack overflow has occurred while running a code
    6. What are exceptions, and who is responsible for throwing exceptions
    7. Struct vs. union
    8. structural padding in structure (how does it work? Why is it there? Can we tell compiler to not to do padding?
    9. what is the base of log(n) in bst. told 2 then asked if I have 3-ary, 4-ary tree then base of log is 3 and 4
    10. what happens when we push back elements in vector (here he explained the role of RAII)
    11. debugging tools
    12. What is recursion, and what are the important points we need to take care of while writing a recursive program?
    13. Virtual Destructor
    14. Internal working of virtual function
    15. delete vs free
    16. malloc vs calloc , what is the return type of malloc(), the difference between malloc() and calloc(), can we dereference a void pointer
    17. SDCL
    18. What is the difference between reference and pointer?
    19. How delete [] is different from delete?
    20. What is the difference between new() and malloc()? , new vs malloc
    21. How do you allocate and deallocate memory in C++

38

OS

* + 1. What do you mean by an operating system? What are its basic functions?
    2. What are the benefits of a multiprocessor system?
    3. How does a computer start?
    4. What is a BIOS?
    5. What is bootloader?
    6. What is kernel?
    7. What is Real-Time Operating Systems? Which scheduling algorithm is used in RTOS?
    8. Explain demand paging?
    9. What do you mean by process synchronization?
    10. What is an interrupt?
    11. What is different between main memory and secondary memory?
    12. What is virtual memory?
    13. What can be the size of virtual memory?
    14. What is a thread?
    15. What is Multithreading?
    16. What are advantages od threads? Why not use multiple process instead of thread?
    17. What is PCB?
    18. What is context switching?
    19. Why is context switching isfast in threads?
    20. What is Thrashing?
    21. What is difference between Process and Thread.
    22. Whats difference between malloc and new?
    23. If a variable reference created using malloc, can child thread access it? Do thread need a process to run?
    24. Schedulers, types of schedulers, role of every scheduler
    25. How are processes scheduled
    26. Types of queues in the system
    27. Interprocess Communication Types
    28. kernel mode, user mode
    29. difference between Mutex and semaphore
    30. Round robin algorithm
    31. why threads are lightweight
    32. what is segmentation fault
    33. multithreading and its in-depth working
    34. system calls
    35. OS Memory Hierarchy is Most Important
    36. Questions on Cache, TLB where it placed in the hierarchy of Memory and their importance
    37. Multiprogramming vs Multitasking
    38. Multitasking vs MultiThreading
    39. What is Dispatcher
    40. Dispatcher vs Scheduler
    41. What is accounting info in PCB
    42. What are Interrupts
    43. CPU Bound and I/O Bound
    44. Context Pointer
    45. Preamtive vs Non Preamtive
    46. Type of scheduling method with their disavantages
    47. What are problems in What is Convoy Effect
    48. Starvation and solution for starvation
    49. Types of Queue Scheduling
    50. Synchronization
    51. Producer Consumer Problem
    52. Dinners Philosopy Problem
    53. Race Condition
    54. Printer Spooler Problem
    55. Critical Section
    56. Condition to acheive sync
    57. Fork system call
    58. Spin Lock
    59. Process vs Thread
    60. User level thread vs Kernel Level Thread
    61. Binary Semaphore and Counting Semaphore
    62. Peterson's Solution
    63. What is deadlock? How is deadlock Prevented
    64. What is Busy Wait? What are the solutions for busy wait
    65. Bankers Algo
    66. Memory Management Fixed Paertioning vs Variable Partioning
    67. Types of memory allocation technique and their advantages and disadvantages
    68. Page Table Entry
    69. Paging
    70. How to convert logical add to physical add
    71. How does os know when to load page from virtual mem to main mem
    72. Inverted Paging
    73. Segmentation
    74. Paging vs Segmentation
    75. Virtual Mem
    76. TLB
    77. Page Replacement Algo

DBMS & SQL

1. ACID
2. Indexing
3. DML
4. DDL
5. Joins
6. Subquery
7. Normalization
8. Denormalization
9. what is Subquery and the types of Subquery?
10. https://stackoverflow.com/questions/38851360/sql-query-to-create-a-report-of-matches-played
11. given employee table , with id, name , salary, mgr\_id, find the number of employees whose salary is higher than their managers
12. real-world databases
13. Aggregate functions in DBMS in SQL
14. design the database schema for the library management system. then asked to normalize it
15. Delete vs Drop vs Truncate

Coding

* + 1. Given a linkedlist, traverse m nodes and delete n nodes in it
    2. given an array of words, print all the pairs of palindrome formed after concatenation of 2 words
    3. anagram problem
    4. Code to find if a number is divisible by 32 without using arithmetic operators
    5. worst time and average time complexity of Binary Search tree
    6. i). set a bit at position pos ii). clear a bit at position pos iii) toggle a bit at position pos
    7. Write a C program to reverse a string (iterative and recursive version)
    8. write the code to insert an element in BST
    9. code for binary tree creation
    10. Given two numbers x and y (x <= y), find out the total number of natural numbers, say i, for which i! is divisible by x but not y
    11. max diameter of binary tree
    12. Then he asked about type casting in C
    13. Gobal vs Static Variable
    14. code min stack implementation, which I did on notepad with two approaches(1st by using only one stack and 2nd by using two stacks)
    15. Finbonacci series
    16. Find all permuatations strings
    17. second max in an array with any extra DS in one pass O(N)
    18. occurrence of String B in String A
    19. Reverse only words of that sentence Eg. "India is my country" O/p-"Country my is India"
    20. Given a string and a character, remove occurrences of that character from that string and then print it
    21. Write an algorithm to insert a node in a doubly linked list

HR

1. what will u do if ur team member is in disagreement with u
2. overcoming difficult situation
3. How will you convince your manager to implement your idea
4. what was the situation when u felt that u have no way to go in life, how did u deal with it?
5. What is the meaning of success according to you.
6. If your colleague is lazy what would you do?
7. my weakness, my strength, what I have done to overcome my weakness
8. how I resolve conflict as a team player, how I work as a team
9. Why should I hire you
10. What will you do If you come across a grumpy boss.
11. Tell me some moments where you have shown creativity In your work.
12. Tell me a situation where you have taken ownership of a task.
13. Tell me if you were given a complex problem, what would be your approach to the problem.
14. Are you interested in sports, who is your favourite sportsperson and why?