OOPS && C++

1. Why c++ is faster than python

1) c++ is static typed compiled language. c++ code is directly compiled in machine code. The compiled binary is typically optimized for speed and directly run on hardware
Python is interpreted language. Python code is executed line by line it is more overhead

2) c++ is static typed. types are checked at compile time
Python is dynamic typed. types are checked at runtime
This flexiblity comes the cost of performance, as the interpreter needs to determine the type of variable and object dynamically during execution.

3) c++ memory allocation and deallocation can be explicitly controlled by programer (new, delete)
In Python automatic memory managment via garbadge Collection. This introduce overhed due to memory that grabage collector take

4) In c++ compilers are highly optimized and perform extensive optimizations during compilation. This include loop unrolling, inlining functions
In Python improving with each release through bytecode optimizations

5) c++ require more lines of code and explicit memory managment which can slow down development compared to Python
Python simplicity and readability often result in faster development times, making it ideal for

prototyping, scripting and scenarios where rapid development and ease of maintenance are priorities

6) C++ widely used in performance-critical applications such as games engines, system software, embedded systems, and high-frequency taking platforms, where speed and control over hardware are crucial.
Python excels in areas such as web development, data analysis, scientific computing and machine learning

2. Diff b/w c++ and java

1) c++ is multi-paradigm language, object oriented, and generic programming paradigms. It allow low-level memory manipulation and direct hardware access making it suitable for system programming.
Java is primarily an object-oriented language with support of concurrent programming (via threads).
write once run anywhere

2) c++ memory allocation and deallocation can be explicitly controlled by programer (new, delete) which leads to efficient memory usage but also increase the risk of memory leaks and segmentation faults
In Java automatic memory managment via garbadge Collection. This introduce overhed due to memory that grabage collector take

3) c++ uses try, catch, throw keywords for exception handling. It supports both checked exceptions (which must be caught or declared) and unchecked exceptions (which do not need to be explicitly caught)
Java uses try, catch, throw keyword. It primarily supports uncheck exception (RuntimeExceptions) and checked exception (IOExceptions, SQLExceptions) which must be caught or declared in the method signature.

4) c++ provide Standard Templete Library (STL)
Java provide a rich set of libraries in its Java Standard Library (Java API) covering networking, I/O, utilities, GUI

5) c++ Supports pointers, which are variables that store memory address Pointers allow direct manipulation of memory and can lead to more efficient code

but also increase the risk of bugs like null pointer dereferencing and memory leaks.
Java does not support pointers. Insted all objects are accessed through reference which are automatically managed by the JVM. Java reference are safer than C++ pointers but do not provide low-level memory control.

6) C++ provide operator overloading
Java does not provide operator overloading

7) C++ provide multiple inheritence of class
Java does not provide multiple inheritence of class

3. Difference b/w C and C++

1) C is procedural programming language. It provide structure approach to programming
C++ is multi-paradigm language. It supports procedural object-oriented and generic programming paradigms. In addition to function it introduce classes and objects for data abstraction and encapsulation

2) C does not support classes and objects
C++ introduces classes which are user-defined types that bind data and methods together. objects are instance of classes, allowing for data abstraction and encapsulation.

3) C does not support object oriented programming concepts like inheritance, polymorphism and encapsulation
C++ provide OOPS concepts such as inheritance (both single and multiple), polymorphism (via operator overloading, virtual function, function overloading), encapsulation (dat hiding within (lasses), and abstraction (seperation of interface and implementation)

4) C uses malloc, calloc, free for dynamic memory allocation and deallocation
C++ support new delete and automatic memory managment (via destructor and smart pointers)

5) C does not support function overloading
C++ support function overloading

6) C provide stdio.h, stdlib.h for input/output, memory allocation etc.
C++ support STL that have additional Components

like vector, map, sorting, searching, iterators
it uses iostream

7) C++ is not directly compatible with C libraries unless they are specifically written to be compatible or use technique like 'extern "C"'.

8) C uses traditional error handling technique like return codes (eg: return -1, errno)
C++ have try, catch, throw keywords for error handling

9) C does not provide namespaces - All global identifiers share the same global namespace.
C++ introduces namespaces, which allow grouping identifiers (variables, functions, classes) into named scopes to avoid naming conflicts

10) C does not support operater overloading
C++ support operator overloading

4. What is polymorphism and how it is achived in c++

It refers to the ability of diff classes to be treated as instance of the same class through a commen instance. It allow objects of different types to be handled using a uniform interface, providing flexiblity and extensibility in code design.

Types of polymorphism

1) Compile time :- operator overloading
function overloading
Templates

2) Run time :- function Overriding
virtual function

1) operator Overloading

c++ allow operator overloading meaning you can redefine how operator behave for user-defined type (classes and structure)

2) Function overloading

You can define multiple function with same name but diff parameter list. Compiler determines which fun to call based on no and types of arguments passed at compile time.

3) Templates

Templates allow generic programming in c++ enabling you to write fun and classes that can handle diff data Types without having to rewrite the code for

each special types.

1) Function overriding

Inheritance in c++ allows derived classes to provide specific implementations of methods that are already defined in their base classes this is achived using virtual functions
The correct fun to call is determined at runtime based on the type of object being referred to (dynamic dispatch)

```cpp
#include<iostream>
#include<string>
using namespace std;
class Animal
{
    public:
    int height;
    int size;
    int weight;
    string color;
    //constructor cant be declare as virtual
    //constructor can be private
    Animal()
    {
        cout<<"I Am In The Animal Class"<<endl;
    }
};

class cat:public Animal
{
    public:
    int height;
    string color;
    cat()
    {
        cout<<"I Am In The Cat Class"<<endl;
    }
};

int main()
{
    // //upstreaming
    // //firstly we use cat class and cat obj
    // //when we make obj of cat than firstly animal constructor than cat constructor is called
    // dynamically allocate memory for an object of type cat and initialize a pointer to this object.
    // cat * c=new cat;


    // //upstreaming
```

```
37
38
39          // //upstreaming
40          // //here we use Animal class and Animal obj
41          // //when we make animal obj than it call only animal constructor
42          // dynamically allocate memory for an object of type Animal and initialize a pointer to this object.
43          // Animal * c=new Animal;
44
45
46          // //upstreaming
47          // //here we use Animal class and cat obj
48          // //when we make the Animal class obj than it call firstly Animal constructor than call cat constructor
49          // Animal * c=new cat;
50
51
52          // //downstreaming
53          // //here we use cat class and Animal class as object
54          // //it give error when we use previous method for downstreaming
55          // //cat * c=new Animal;
56          // //for downstreaming we use this
57          // //when we make the cat class object than it call only Animal constructor
58          // cat* c=(cat*) new Animal;
59          // Animal* a = new cat(); // Correct upcasting, cat is a type of Animal it calls animal constructor
60          // cat* c = dynamic_cast<cat*>(a); // Safe downcasting it calls cat constructor
61
62
63
64      }
```

2) virtual function

Virtual functions are functions in base classes that overridden in derived classes. They enable dynamic dispatch and allow the correct fun to be called for objects of derived classes when accessed through pointers or references of base class type

```
class Shape {
public:
    virtual void draw() {
        cout << "Drawing shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing circle" << endl;
    }
};

int main() {
    Shape* shape = new Circle();
    shape->draw();  // Calls Circle's draw() due to dynamic dispatch
    delete shape;
    return 0;
}
```

5. What are Abstract classes and interfaces and what is the diff b/w the two

Abstract classes

An abstract class in C++ is a class that can't be instantiated on its own. It is ment to serve as a base class (or superclass) for other classes. Abstract classes often contain one or more pure virtual function. These pure virtual functions act as placeholders for functions that must be implemented by any concrete subclass.

Interfaces

An interface in C++ is a class that only contains pure virtual fu^n. It defines a contract that concrete classes can choose to implement. Interfaces do not have member variables or concrete fu^n implementations

Diff b/w Abstract Class and Interfaces

1. Purpose
Abstract class provides a partial implementation and can have data members and implemented fu^n along with pure virtual fu^n
Interfaces does not contain any implementation, only pure virtual fu^n

2. Member variables
Abstract class can have member variables
Interface can't have member variables

3. Virtual fu^n
Abstract class have both pure virtual fu^n and implemented virtual fu^n

Interface contains only pure virtual fu"

4. Inheritance
Abstract class inherited to provide a common interface along with some default behaviour.
Interface inherited but to enforce the implementation of certain behaviour without any default behaviour or data.

6. if you can define method in interfaces also, then what's the need of abstract classes?

Abstract class can provide concrete (non-pure) methods along with pure virtual methods. This allow them to define default behaviour can be inherited by subclasses. Subclasses have the option to override these methods if needed but are not required to do so.
Interfaces (typically represented by classes with only pure virtual functions in C++) can't contain any default implementations. They strictly define a contract that derived classes must implement.

7. What is abstraction in c++?

Data abstraction is one of more essential & imp feature. it means display only essential info and hiding the details Data abstraction refers to providing only essential info about the data to the outside world, hiding the background details or implementation

eg:- Car :- all its implementation is hidden from the driver

Types of abstraction

1. Data Abstraction :- This type only shows the required info about the data and hides the unnessary data.
2. Control abstraction :- This type only shows the required info about the implementation and hides unnecessary info.

Abstraction in Header files

we import header files but we don't know what data on it and what implementation are their

Abstraction using Access specifiers

These are the main piller of implementing abstraction in c++ we can use access specifiers to restrictions on class members

Public :- access anywher
Private :- access only within the class
Protected :- access only within the class with privately

Advantages of Data abstraction

→ Helps the user to avoid writing the low-level code
→ Avoid code duplication and increases reusability
→ Can change the internal implementation of class independently without affecting the user
→ Helps to increase the security of an application or program as only imp details are provided to the user
→ It reduces the complexity as well as the redudancy of code, therefore increase the readability.

Abstraction through interface

Interfaces in C++ are often defined using abstract classes (classes contain pure virtual fun^n) or pure virtual fun^n within regular classes. They define a contract that derived classes must adhere to by implementing the abstract method.

8. what is method hiding in C++

when derived class defines a method with the same name as a method in its base class the derived class method hides or "shadows" the base class method This means when you call the method on an object of the derived class the derived class's method is invoked. The base class's method remains available but is not directly accessible through the derived class unless explicitly qualified
To access the hidden base class method you must qualify the call with the base class name

```
BaseClassName :: function();
```

9. what is regular languages

Regular language are a class of languages that can be described by regular expression and recognized by finite automata.

1. Finite Automata :- Regular language can be recognized by finite automata, which are abstract machines with a finite number of states.

There are 2 types of finite Automata

Deterministic Finite Automata (DFA) :- Each state has exactly one transition for each symbol in the alphabate.

Non-deterministic Finite Automata (NFA) :- States may have zero, one, or multiple transitions for each symbol and the machine can be in multiple states at once.

2. Regular Expressions :- like union ('|'), concatenation, ('*')

3. Closure Expressions :- Union, intersection, complement, Concatenation, Kleene star

4. Regular Grammars.

5. Closure Under homomorphisms.

6. Decidability :- To take decision

Practicle Applications :- Text processing, Lexical Analysis, Network Protocal

Lexical Analysis :- This is first phase of a Compiler or interpreter where the input source code is processed to produce a sequence of tokens

Tokens :- Keywords, identifiers, operators, literals, punctuation (; , :)

10. what is virtual classes

Virtual inheritance ensure that only one instance of a base class is inherited by a derived class in a multiple inheritance hierarchy. This is crucial when you have a diamond-shaped inheritance pattern where base class is inherited through multiple paths.

Diamond problem

The diamond problem occurs when a class inherits from two classes that both derive from a common base class. This can lead to ambiguity and duplicate of base class's members.

```cpp
#include <iostream>

class Base {
public:
    void show() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived1 : public Base {
};

class Derived2 : public Base {
};

class Final : public Derived1, public Derived2 {
};

int main() {
    Final obj;
    obj.show(); // Error: Ambiguous call to Base::show
    return 0;
}
```

To solve this problem we use virtual inheritance By declaring the base class as a virtual base class C++ ensures that there is only one instance of the base class, no matter how many paths lead to it

```cpp
#include <iostream>

class Base {
public:
    Base() {
        std::cout << "Base constructor" << std::endl;
    }

    void show() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived1 : virtual public Base {
public:
    Derived1() {
        std::cout << "Derived1 constructor" << std::endl;
    }
};

class Derived2 : virtual public Base {
public:
    Derived2() {
        std::cout << "Derived2 constructor" << std::endl;
    }
};

class Final : public Derived1, public Derived2 {
public:
    Final() {
        std::cout << "Final constructor" << std::endl;
    }
};
```

Remember one thing you must initialize the virtual base class explicity in the most derived class's

cpp Copy code

```cpp
class Final : public Derived1, public Derived2 {
public:
    Final() : Base() {  // Explicitly initialize Base
        std::cout << "Final constructor" << std::endl;
    }
};
```

11. What is diff b/w the static, auto, extern, register.

Static :- static variable inside the function maintain its value b/w fun call. it is accessible only within the function

Static variable outside the function we can use this static variable during whole file we can't use it in another file that we (link internally)

extern :- extern is used to declare a variable or fun that is defined in another file. This allow you to share global variables and functions accross multiple files.

It provide external linkage, meaning the variable or function can be accessed from other files.

extern is often used in header files to declare variables or functions that are defined in corresponding .cpp or .c file

```cpp
// In file1.cpp
int globalVar = 10;  // Definition of the global variable

// In file2.cpp
extern int globalVar;  // Declaration of the global variable
```

auto :- By default local variables in C++ are auto (automatic) variables, meaning they are automatically created and destroyed when they come in and out of scope.

The storage duration of auto variables is limited to the scope they are defined in (usually a fun^n or a block). They are destroyed once the scope ends.

Register :- The scope of a register variable is local to the block in which it is defined.
its scope is in within the block

The register keyword is a request to the compiler to store the variable in a CPU register instead of RAM for faster access. However the compiler may ignore this request based on availability and optimization strategies

You can't take the address of a register variable (eg:- '&variable' is not allowed)

Typically, the compiler has final say on whether a variable is placed in a register, regardless of the 'register' keyword

It is used for performance-critical variables that are frequently accessed such as loop counters

```
void fun()
{
  register int i;
  for (i=0; i<1000; i++)
  {

    //Fast access to i
  }
```

we can't declare register variable as globally

```
register int globalVar; // Invalid
```

12. Compilation stages of C program?

1. Preprocessing :- input - Source file (.c)
output - Preprocessed source code (expanded source file) (.i)

It handel header files #include, #define, #if, macro substitution

2. Compilation :- input - Preprocessed source code (.i file)
output - Assembly code (.s file)

Compiler perform syntax checking, semantic analysis, and optimization during this phase and convert code to Assembly code

3. Assembly :- input - Assembly code (.s file)
output - Object code (.o or .obj file) (binary format)

It convert it in machine code but is not yet a complete executable. The obj file includes machine instructions, as well as info about external references (eg: fun or variables defined in other files)

Linking :- input - object code (.o file) and libraries
output - exe file (a.out, a.exe etc.)

here all necessary libraries (standard C library, math library) are combine to object file and

make executable file

There are 2 types of linking

Static linking :- All the required libraries are included in the final executable file.

Dynamic linking :- Only references to shared libraries are included and actual linking happen at runtime

Loading (Runtime) :- input - Executable file
output - Running program in memory

once the executable is created the operating system loader loads the program into memory and start execution. If dynamic linking is used, the necessary shared libraries are loaded at this time as well.

13. What is Conditional Compilation

Condition compilation allow sections of code to be compiled or ignored depending on certain conditions, which is useful for platform-specific code, debugging, and configuration options

This is controlled by preprocessor directives like #if, #ifdef, #ifndef, #else, #elif and #endif

14. Hive we have static fun in a class then we make 10 obj of that class. how many static fun copies will be made in memory.

Static member fu" do not operate on a specific instance of the class, therefore they do not have access to non-static member variables or non-static member fu" . They can only access static member variables and other static member fu"
They are shared along all instances of the class

Even if you have 10 objects of a class only one copy of each static fu" will be made in memory.

15. what is diff b/w inline and macro

1) inline :- Compiler replace fu" call with the function code
macro :- Preprocessor handled these . This is textually replaced wherever the macro is used

2) inline :- Compiler check when he replace it so it is efficent or not if it is not efficient then the fu" still has all features of a normal fu" like type checking, scope etc .
macro :- The preprocessor replaces the macro with the defined code before the compilation phase . This is purely a text substitution , with no type checking or scope rules applied.

3) Inline :- They are safe because they are type-checked by the compiler , scope rules etc .
Macro :- They are less safe they not type-checked , scope rule, espacially if you do not carefully parenthesize the macro parameter.

4) Inline :- Easy debug
Macro :- difficult to debug because they do not generate seprate code they expanded inline as text . Debuggers typically do not show macro expansions, making it harder to

trace issues related to macros.

5) Inline :- Inline fun^n respect scope. They are defined within the scope of a class or namespace and behave accordingly.

Macro :- Macro do not respect scope. Once defined, a macro is available globally in the code that follows its definition, which can lead to naming conflicts or accidental redefinitions.

16. Dynamic memory allocation in C

1. Malloc()

Purpose :- Allocates a block of memory of the specified size (in bytes). The contents of the allocated memory are unitialized, meaning they will contain garbage values.

Return :- A pointer to the allocated memory
NULL if allocation fails.

```
int * ptr = (int*) malloc (sizeof(int)*5);
if (ptr == NULL)
{
    // Handle exception failures
}
```

2. Calloc()

Purpose :- Allocates memory for an array of elements and initializes all bytes to zero. This can be useful if you need zero-initialized memory.

Return:- A pointer to the allocated memory
NULL if the allocation fails

```
int *ptr = (int*) calloc (5, sizeof(int));
if (ptr == NULL)
{
    // Handle allocation failure
}
```

2. realloc()

Purpose:- Resize a previously allocated memory block allowing it to grow or shink. The content of the block are preserved up to the minimum of the old and new sizes.

Return:- A pointer to the reallocated memory
NULL if the reallocation fails. The original block remains intact if reallocation fails.

Note:- if ptr is NULL, realloc() behaves like malloc();

if size is 0 and ptr is not NULL the memory is freed.

if the new size is larger the additional memory is uninitialized

```
ptr = (int*) realloc (ptr, sizeof(int) * 10);
if(ptr == NULL)
{
  //Handle reallocation failure
}
```

if one memory form 1000 to 1012 and other memory from 1013 to 1025 when we resize first memory so it can't increase because after 1012 other memory is allocated.
So it allocate new resized memory in other place copy prev content to new add an free old address

4. free()

Purpose :- Deallocates the memory that was previously allocated by 'malloc()', 'calloc()', realloc().

```
free(ptr)
ptr = NULL  // Good practice to avoid dangling pointer
```

Note :- Always free dynamically allocated memory when it is no longer needed to avoid memory leaks

Heap Vs Stack

Heap Memory :- Dynamic memory is allocated on the Heap, which is a region of memory that provides storage for variables whose size and lifetime are not known at compile-time.

Stack Memory :- local variables with functions are typically allocated on the stack - Stack memory is automatically managed and released when the fun exits.

Issues :-

Memory Leaks :- when dynamically allocated memory is not freed causing a program to consume more and more over time

Dangling Pointers :- Occur when a pointer is used after the memory it points to has been freed.

Double free :- Occur when a program attempts to free the same memory block more than Once leading to undefined behaviour.

Buffer Overflow :- Occur when a program writes more data to a memory block than it was allocated for, potentially corrupting adjacent memory.

12. When we declare a pointer to a class that has some integer and an object of another class as a member then how the memory is allocated to the variables?