C++ TECHNICAL INTERVIEW QUESTIONS

Q1. What is C++?
Q2. Differentiate between C and C++?
Q3. Explain the basic structure of a C++ program.
Q4. What are the key features of C++?
Q5. Define object-oriented programming (OOP) and its principles
Q6. What are the advantages of using C++ over C?
Q7. Explain the difference between classes and objects.
Q8. What is a constructor? Explain its types.
Q9. What is a destructor? Why is it used?
Q10. Differentiate between pass-by-value and pass-by-reference with examples.
Q11. Describe the concept of function overloading.
Q12. Explain the difference between public, private, and protected access specifiers.
Q13. What is inheritance in C++? Explain its types.
Q14. Describe the concept of polymorphism.
Q15. What is the virtual function in C++? Why is it used?
Q16. Explain the purpose of the 'new' and 'delete' operators.
Q17. Define templates in C++. Explain their types.
Q18. What is the STL (Standard Template Library)? Mention its components.
Q19. Describe the differences between arrays and vectors in C++
Q20. Explain the difference between overloading and overriding.
Q21. What is the 'this' pointer in C++?
Q22. Describe the concept of operator overloading.
Q23. Explain the use of the 'const' keyword in C++.
Q24. What is a namespace? Why is it used in C++?
Q25. Describe the importance of the 'iostream' library in C++.
Q26. How does exception handling work in C++?
Q27. Explain the concept of dynamic memory allocation in C++.
Q28. What is a friend function in C++?
Q29. Describe the role of the 'static' keyword in C++.
Q30. How does C++ support multiple inheritances?
Q31. Explain the concept of smart pointers in C++.
Q32. Describe the use of function pointers in C++.
Q33. What is the purpose of the 'volatile' keyword in C++?
Q34. Explain the concept of 'RAII' (Resource Acquisition Is Initialization).
Q35. Describe the difference between shallow copy and deep copy.
Q36. Explain the concept of threading in C++.
Q37. What are lambda expressions in C++? How are they used?
Q38. Explain the differences between stack and heap memory in C++.

Q39. Describe the role of the 'mutable' keyword in C++.
Q40. What is a virtual destructor? Why is it needed?
Q41. Explain the concept of move semantics in C++.
Q42. Describe the use of 'const' member functions in C++.
Q43. What are C++ references? How are they different from pointers?
Q44. Explain the concept of abstract classes in C++.
Q45. Describe the differences between function overloading and function overriding.
Q46. What are 'constexpr' variables in C++? How are they used?
Q47. Explain the role of the 'volatile' keyword in multi-threading.
Q48. What is the purpose of the 'typeid' operator in C++?
Q49. Describe the concept of 'decltype' in C++.
Q50. Explain the use of the 'std::move' function in C++.
Q51. What are the advantages of using the 'auto' keyword in C++?
Q52. Describe the differences between static and dynamic polymorphism.
Q53. Explain the concept of 'try', 'throw', and 'catch' in exception handling.
Q54. What is the role of 'std::bind' in C++?
Q55. Describe the concept of 'CRTP' (Curiously Recurring Template Pattern).
Q56. Explain the concept of 'std::forward' in C++.
Q57. Describe the purpose of the 'std::unique_ptr' and 'std::shared_ptr' in C++.
Q58. What is the 'std::function' in C++?
Q59. Explain the concept of 'constexpr' functions in C++.
Q60. Describe the use of 'std::thread' in C++ multithreading.
Q61.What are the different Data Types present in C++?
Q62. What are References in C++?
Q63.Define token in C++?
Q64.Is Deconstructor Overloading Possible? If Yes then Explain and if no then why?
Q65.What are the Static Members and Static Member Functions in C++?
Q66.Difference between equal to (==) and assignment operator(=)?
Q67.What is Loops in C++? Explain different types of loops in C++
Q68.What is the difference between virtual functions and pure virtual functions in C++?
Q69.When should we use multiple inheritance in C++?
Q70.Which operations are permitted on pointers?
Q71.How delete [] is different from delete in C++?
Q72.Can you compile a program without the main function?
Q73.Define inline function. Can we have a recursive inline function in C++?
Q74.What is the difference between C and C++? The main difference between C and C++ are provided in the table below:
Q75.What is the difference between struct and class?
Q76.How do you allocate and deallocate memory in C++?
Q77.why C++ called as an object oriented

Q78.What are void pointers?
Q79.Compare compile time polymorphism and Runtime polymorphism
Q80.What is the difference between shallow copy and deep copy?
Q81.Differences between References and Pointers:
Q82. Difference b/w scanf & printf comparing with cat & cin ?

C++ Interview Questions and Answers

Q1. What is C++?
Ans: C++ is a powerful, high-level programming language developed by Bjarne Stroustrup. It is an extension of the C programming language and supports procedural, object-oriented, and generic programming paradigms. C++ incorporates both low-level features for system-level programming and high-level features for application development. It provides a rich set of libraries and features that allow developers to write efficient, portable, and versatile code.

Q2. Differentiate between C and C++?
Ans:

- **Paradigm:** C is a procedural programming language while C++ supports procedural as well as object-oriented programming paradigms.
- **Abstraction:** C is less abstract and provides fewer high-level constructs, whereas C++ offers more abstraction with classes, objects, and templates.
- **Code Style:** C uses functions and structures, while C++ emphasizes the use of classes and objects.
- **Memory Handling:** C++ supports features like constructors, destructors, and RAII (Resource Acquisition Is Initialization), which allow better memory management compared to C.
- **Standard Libraries:** C++ provides richer standard libraries and additional functionalities, such as the Standard Template Library (STL), which are not present in C.

Q3. Explain the basic structure of a C++ program.
Ans: A basic structure of a C++ program consists of :

- **Preprocessor Directives:** It includes libraries and header files using #include.
- **Namespace Declaration:** Using using namespace std; for easier access to standard library functions.
- **Main Function:** The entry point of execution, defined as int main().
- **Statements and Expressions:** The actual code logic written within the main function.
- **Return Statement:** Exiting the main function using return 0; or a value denoting the program's completion status.

Example:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!";
    return 0;
}
```

Q4. What are the key features of C++?
Ans:

- **Object-Oriented:** Support for classes, objects, inheritance, polymorphism, encapsulation.
- **Standard Template Library (STL):** Collection of classes and functions for data structures, algorithms, and iterators.
- **Strongly Typed:** Type-safe language with static and dynamic type checking.
- **Memory Management:** Allows dynamic memory allocation and deallocation using new and delete.
- **Operator Overloading:** The ability to redefine operators for user-defined types.
- **Function Overloading:** Multiple functions with the same name but different parameters.
- **Exception Handling:** Enables handling and propagation of runtime errors gracefully.

Q5. Define object-oriented programming (OOP) and its principles.
Ans: Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which encapsulate data and behavior. Its key principles include:

- **Encapsulation:** Bundling of data (attributes) and methods (functions) that operate on the data within a single unit (object).
- **Inheritance:** Capability of a class to inherit properties and behavior from another class, promoting code reuse and hierarchy.
- **Polymorphism:** Ability to present one interface for multiple data types, allowing objects of different classes to be treated as objects of a common superclass.
- **Abstraction:** Conceptualization of essential features without including the background details.
- **Modularity:** Organizing code into independent units (classes), enabling better maintainability and reusability.

Q6. What are the advantages of using C++ over C?
Ans:

- **Object-Oriented Paradigm:** C++ supports OOP, offering better code organization and reusability through classes and objects.
- **Stronger Type Checking:** C++ has stronger type checking mechanisms, reducing errors during compile-time.
- **Richer Standard Library:** C++ provides a more extensive standard library (STL) for containers, algorithms, and utilities.
- **Support for Exception Handling:** C++ supports exceptions for handling runtime errors in a structured manner.
- **Function Overloading and Operator Overloading:** C++ allows functions and operators to be overloaded, enhancing code readability and versatility.
- **Improved Memory Management:** C++ features RAII, constructors, and destructors for better memory management compared to C.
- **Compatibility with C:** C++ is largely compatible with C, allowing C code to be easily integrated into C++ programs.

Q7. Explain the difference between classes and objects.
Ans:

- **Class:** A class in C++ is a blueprint or a template that defines the properties (data members) and behaviors (member functions) common to all objects of that class. It serves as a user-defined data type.
- **Object:** An object is an instance of a class. It is a real-time entity that represents a specific instance of the class, possessing its own set of data members and functions.

Example:

```
class Car {     // Class definition
    public:
        string brand;
        int year;
};
int main() {
    Car carObj;     // Object of class Car
    carObj.brand = "Toyota";
    carObj.year = 2022;
    return 0;
}
```

Q8. What is a constructor? Explain its types.
Ans: A constructor is a special member function in a class that gets automatically called when an object of that class is created. It is used to initialize the object's data members.

Types of constructors in C++:

- **Default Constructor:** Constructor with no parameters. It is automatically invoked when an object is created without any arguments.
- **Parameterized Constructor:** Constructor with parameters. It allows initialization of data members during object creation.
- **Copy Constructor:** Constructor that initializes an object using another object of the same class.

Example:

```
class Rectangle {
    public:
        int length, breadth;
      // Parameterized Constructor
        Rectangle(int l, int b) {
            length = l;
            breadth = b;
        }
};
```

```
int main() {
    Rectangle rect1(5, 10);  // Object creation and initialization using a parameterized constructor
    return 0;
}
```

Q9. What is a destructor? Why is it used?
Ans: A destructor is a special member function in a class that is automatically called when an object goes out of scope or is explicitly deleted. Its purpose is to release resources or perform cleanup activities before the object is destroyed.

Usage and Importance:

- **Resource Deallocation:** Destructors are used to release resources acquired by the object during its lifetime, such as memory, files, or database connections.
- **Memory Deallocation:** They play a crucial role in freeing dynamically allocated memory managed by the object.
- **Cleanup Activities:** Destructors can be utilized to perform clean-up operations like closing files, releasing locks, or freeing resources used by the object.

Example:

```
class MyClass {
public:
    // Constructor
    MyClass() {
        // Constructor code
    }
    // Destructor
    ~MyClass() {
        // Cleanup code
    }
};
int main() {
    MyClass obj; // Object creation

    // obj goes out of scope here, and its destructor is called automatically
    return 0;
}
```


Q10. Differentiate between pass-by-value and pass-by-reference with examples.
Ans:

- **Pass-by-Value:** In pass-by-value, a copy of the actual parameter is passed to the function. Any modifications made to the parameter inside the function do not affect the original value.
- **Pass-by-Reference:** In pass-by-reference, the memory address (reference) of the actual parameter is passed to the function. Modifications made to the parameter inside the function directly affect the original value.

Example:

```
// Pass-by-Value
void incrementValue(int num) {
    num++; // Changes are local to this function
}

int main() {
    int value = 5;
    incrementValue(value);
    cout << value; // Output: 5 (original value remains unchanged)
    return 0;
}

// Pass-by-Reference
void incrementReference(int &num) {
    num++; // Changes affect the original value
}

int main() {
    int value = 5;
    incrementReference(value);
    cout << value; // Output: 6 (original value is modified)
    return 0;
}
```


Q11. Describe the concept of function overloading.
Ans: Function overloading is a feature in C++ that allows multiple functions within the same scope to have the same name but different parameters or argument lists. The compiler differentiates these functions based on the number, type, or order of parameters.

Example:

```
// Function Overloading
int add(int a, int b) {
    return a + b;
}
float add(float a, float b) {
    return a + b;
}
int main() {
```

```
    cout << add(5, 10) << endl;    // Calls the int version of add
    cout << add(2.5f, 3.7f) << endl;  // Calls the float version of add
    return 0;
}
```

Q12. Explain the difference between public, private, and protected access specifiers.
Ans:

- **Public:** Members declared as public within a class are accessible from outside the class as well as from derived classes. They have no access restrictions.
- **Private:** Members declared as private are only accessible within the same class. They are not accessible from outside the class or from derived classes.
- **Protected:** Similar to private members, but they are accessible within the class and by derived classes.

Example:

```
class Example {
public:
    int publicVar;     // Public member
private:
    int privateVar;    // Private member
protected:
    int protectedVar;  // Protected member
};

int main() {
    Example obj;
    obj.publicVar = 10;    // Accessible
    // obj.privateVar = 20; // Not accessible (compile error)
    // obj.protectedVar = 30; // Not accessible (compile error)
    return 0;
}
```


Q13. What is inheritance in C++? Explain its types.
Ans: Inheritance is a fundamental feature in object-oriented programming that enables a new class (derived class) to inherit properties and behaviors from an existing class (base or parent class). This allows code reuse and promotes the concept of hierarchy.

Types of inheritance in C++:

- **Single Inheritance:** A derived class inherits from only one base class.
- **Multiple Inheritance:** A derived class inherits from multiple base classes.
- **Multilevel Inheritance:** A class is derived from another derived class, forming a hierarchy of classes.
- **Hierarchical Inheritance:** Multiple classes are derived from a single base class.

Example:

```
class Animal {
public:
    void eat() {
        cout << "Animal eats" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
int main() {
    Dog myDog;
    myDog.eat();    // Accessing the inherited function
    myDog.bark();
    return 0;
}
```

Q14. Describe the concept of polymorphism.
Ans: Polymorphism, derived from Greek meaning "many forms," refers to the ability of different objects to be treated as objects of a common type. In C++, polymorphism can be achieved through two mechanisms: compile-time (early binding) and runtime (late binding) polymorphism.

- **Compile-time Polymorphism (Static Binding):** This is achieved through function overloading and operator overloading. The decision of which function to call is made at compile time based on the parameters or operators used.
- **Runtime Polymorphism (Dynamic Binding):** This is achieved through inheritance and virtual functions. It allows a function to be defined in multiple forms and the decision on which function to call is made at runtime.

Example:

```
// Compile-time Polymorphism (Function Overloading)
int add(int a, int b) {
    return a + b;
}
float add(float a, float b) {
    return a + b;
}
// Runtime Polymorphism (Virtual Functions)
class Animal {
public:
    virtual void makeSound() {
```

```
        cout << "Animal makes a sound" << endl;
    }
};
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};
int main() {
    // Compile-time Polymorphism
    cout << add(5, 10) << endl;        // int version of add
    cout << add(2.5f, 3.7f) << endl;  // float version of add

    // Runtime Polymorphism
    Animal *animalPtr;
    Dog dog;
    animalPtr = &dog;  // Pointing to a Dog object through base class pointer
    animalPtr->makeSound();  // Calls the overridden function in Dog class
    return 0;
}
```

Q15. What is the virtual function in C++? Why is it used?
Ans: A virtual function is a member function in a base class that is declared using the virtual keyword and can be overridden in derived classes. When a function is marked as virtual, it enables dynamic polymorphism, allowing the correct function to be called at runtime based on the object's actual type rather than the pointer or reference type.

Purpose:

- Facilitates runtime polymorphism by allowing the function to be overridden in derived classes.
- Enables the base class pointer to invoke the overridden function in derived classes.

Example:

```
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 *shapePtr;
    Circle circleObj;
    shapePtr = &circleObj; // Pointing to a Circle object through base class pointer
    shapePtr->draw();      // Calls the overridden function in Circle class
    return 0;
}
```

Q16. Explain the purpose of the 'new' and 'delete' operators.
Ans:

- **new Operator:** It is used for dynamic memory allocation in C++. It allocates memory for a single object or an array of objects on the heap and returns a pointer to the allocated memory.
- **delete Operator:** It is used to deallocate memory that was allocated using the new operator. It releases the memory occupied by a single object or an array of objects from the heap.

Example:

```
int *numPtr = new int; // Allocating memory for a single integer
*numPtr = 10;
delete numPtr; // Deallocating memory

int *arrPtr = new int[5]; // Allocating memory for an array of integers
delete[] arrPtr; // Deallocating memory for the entire array
```

Q17. Define templates in C++. Explain their types.
Ans: Templates in C++ allow for generic programming by enabling the creation of functions and classes that work with any data type. They help in writing reusable code by defining algorithms without specifying the actual data types.

Types of templates:

- **Function Templates:** They allow defining generic functions that can work with multiple data types.
- **Class Templates:** They enable the creation of generic classes that can work with different data types.

Example:

```
// Function Template
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

// Class Template
template <class T>
```

```
class Pair {
private:
    T first, second;

public:
    Pair(T a, T b) : first(a), second(b) {}
    T getMax() {
        return (first > second) ? first : second;
    }
};

int main() {
    cout << maximum(5, 10) << endl; // Calls the maximum function template with int arguments
    Pair<int> pairObj(3, 7); // Creates an object of Pair class with int type
    cout << pairObj.getMax() << endl; // Calls the getMax function in Pair class
    return 0;
}
```

Q18. What is the STL (Standard Template Library)? Mention its components.
Ans: The Standard Template Library (STL) is a powerful set of C++ template classes and functions provided by the C++ Standard Library. It comprises various components that offer generic algorithms, containers, and iterators, promoting code reuse and efficiency.

Components of STL:

- **Containers:** Like vectors, lists, stacks, queues, maps, sets, etc., for data storage and manipulation.
- **Algorithms:** Pre-defined functions (e.g., sort, find) for performing operations on containers.
- **Iterators:** Objects used to traverse through elements of a container.

Example:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    // Vector container
    vector<int> nums = {5, 2, 9, 1, 7};

    // Sorting the vector using STL sort algorithm
    sort(nums.begin(), nums.end());

    // Displaying sorted elements using iterators
    for (auto it = nums.begin(); it != nums.end(); ++it) {
        cout << *it << " ";
    }
```

```
    return 0;
}
```

Q19. Describe the differences between arrays and vectors in C++
Ans:

- **Size:** Arrays have a fixed size determined at compile-time, while vectors can dynamically resize themselves and grow or shrink as needed during runtime.
- **Memory Management:** Arrays have a fixed memory allocation that needs to be defined beforehand, while vectors manage memory dynamically and can expand or contract as elements are added or removed.
- **Functions and Methods:** Arrays do not have built-in methods for resizing or manipulating elements, whereas vectors provide various methods like push_back(), pop_back(), and resize() for easy element handling.
- **Access:** Arrays are simpler in syntax and use square brackets ([]) for element access, whereas vectors use similar syntax but also offer additional functions like at() that perform bounds checking.

Example:

```
// Array
int arr[5]; // Fixed-size array
arr[0] = 10;

// Vector
#include <vector>
vector<int> vec; // Dynamic vector
vec.push_back(10); // Adding elements dynamically
```

Q20. Explain the difference between overloading and overriding.
Ans:

- **Overloading:** Refers to defining multiple functions in the same scope with the same name but different parameters. Overloading can occur within the same class or across derived classes, and it's resolved at compile-time based on the function signature.
- **Overriding:** Occurs in inheritance when a derived class provides a specific implementation for a method that is already defined in its base class. The function in the derived class must have the same signature (name, parameters, return type) as the base class function. Overriding is resolved at runtime through virtual functions.

Example:

```
// Overloading
void print(int num) { cout << "Integer: " << num << endl; }
void print(float num) { cout << "Float: " << num << endl; }

// Overriding
```

```
class Base {
public:
    virtual void display() {
        cout << "Base class display()" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived class display()" << endl;
    }
};
```

Q21. What is the 'this' pointer in C++?
Ans: this is a keyword in C++ that is a pointer available in non-static member functions of a class. It points to the object for which the member function is called. It is used to access the object's own members within the member functions and disambiguate between parameter names and data members.

Example:

```
class MyClass {
private:
    int num;

public:
    void setNum(int num) {
        this->num = num; // 'this' pointer used to distinguish class member 'num' from function parameter 'num'
    }
};
```

Q22. Describe the concept of operator overloading.
Ans: Operator overloading in C++ allows defining custom implementations for operators to work with user-defined types. It enables extending the behavior of built-in operators to work with user-defined objects by defining appropriate member or friend functions for those operators.

Example:

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator+(const Complex &c) {
```

```
        Complex temp(0, 0);
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }
};

int main() {
    Complex num1(3, 4), num2(2, 5);
    Complex sum = num1 + num2; // Operator '+' overloaded for Complex objects
    return 0;
}
```

Q23. Explain the use of the 'const' keyword in C++.
Ans: The const keyword in C++ is used to specify that a variable, function parameter, or member function does not modify the object's state. It provides additional constraints on the variable or function, making them read-only or ensuring that they cannot modify certain data.

Example:

```
const int MAX_VALUE = 100; // Declaring a constant variable

void display(const int num) { // Function parameter as constant
    cout << num << endl;
}
class MyClass {
private:
    int value;

public:
    void getValue() const { // Const member function that doesn't modify the object
        cout << value << endl;
    }
};
```

Q24. What is a namespace? Why is it used in C++?
Ans: A namespace is a feature in C++ that allows the grouping of logically related identifiers (variables, functions, classes) under a unique name. It helps in preventing naming collisions between different parts of a program by organizing the code into different scopes.

Purpose of namespaces:

- **Avoiding Name Clashes:** Namespace helps in avoiding conflicts between identifiers with the same name declared in different parts of the codebase.
- **Maintaining Code Modularity:** It helps in better organizing and structuring the code into logical units or libraries.
- **Encapsulation and Better Code Readability:** Namespaces make the code more readable and understandable by providing a clear context for the identifiers.

Example:

```
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    int result = Math::add(3, 5); // Using the add function within the Math namespace
    return 0;
}
```

Q25. Describe the importance of the 'iostream' library in C++.
Ans: The iostream library in C++ is essential for handling input and output operations. It provides functionalities to interact with the console (standard input/output), enabling reading input from the user and displaying output to the screen.

Importance of iostream:

- **Input/Output Operations:** It allows reading input (using cin) and displaying output (using cout) to the console.
- **Standard Streams:** Provides standard streams (cin, cout, cerr, clog) for input and output handling.
- **Data Formatting:** Supports formatting output with manipulators like setw, setprecision, etc.
- **File Handling:** It provides functionality to perform file-based input and output operations using fstream.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "The entered number is: " << num << endl;
    return 0;
}
```

Q26. How does exception handling work in C++?
Ans: Exception handling in C++ allows dealing with unexpected errors or exceptional conditions in a program. It involves the use of try, catch, and throw keywords.

Working of exception handling:

- try: The code that might throw an exception is enclosed in a try block.
- throw: When an exceptional condition occurs, an exception is thrown using the throw keyword.
- catch: catch blocks are used to handle exceptions. When an exception is thrown, the control jumps to the appropriate catch block that matches the thrown exception type.

Example:

```
#include <iostream>
using namespace std;

int main() {
    try {
        int age;
        cout << "Enter your age: ";
        cin >> age;
        if (age < 0) {
            throw "Invalid age"; // Throwing an exception
        }
        cout << "Your age is: " << age << endl;
    } catch (const char* msg) { // Catching the thrown exception
        cout << "Exception: " << msg << endl;
    }
    return 0;
}
```


Q27. Explain the concept of dynamic memory allocation in C++.
Ans: Dynamic memory allocation in C++ refers to the allocation and deallocation of memory during runtime. It allows programs to request memory dynamically from the heap at runtime, unlike static memory allocation, which occurs at compile-time.

Operators used for dynamic memory allocation:

- **new:** It is used to allocate memory for a single object or an array of objects on the heap.
- **delete:** It is used to deallocate memory that was allocated using new.

Example:

```
int* numPtr = new int; // Allocating memory for a single integer
*numPtr = 10;
delete numPtr; // Deallocating memory

int* arrPtr = new int[5]; // Allocating memory for an array of integers
delete[] arrPtr; // Deallocating memory for the entire array
```

Q28. What is a friend function in C++?
Ans: In C++, a friend function is a function that is not a member of a class but has access to the private and protected members of that class. It can access these members as if it were a part of the class, without being a member of the class itself.

Key points about friend functions:

- Declared inside a class with the friend keyword.
- Not a member of the class but has access to its private and protected members.
- Friendship is not mutual, meaning if function A is a friend of class B, it does not imply that class B is a friend of function A.

Example:

```
class MyClass {
private:
    int privateVar;

public:
    MyClass() : privateVar(0) {}

    friend void friendFunc(MyClass obj); // Declaration of friend function
};

// Definition of friend function accessing private member of MyClass
void friendFunc(MyClass obj) {
    cout << "Private variable of MyClass accessed through friend function: " << obj.privateVar
<< endl;
}

int main() {
    MyClass obj;
    friendFunc(obj); // Calling the friend function
    return 0;
}
```

Q29. Describe the role of the 'static' keyword in C++.
Ans: In C++, the static keyword has various uses depending on its context:

- **Static Variables:** When used with variables inside a function, it retains its value between function calls.
- **Static Member Variables:** When used inside a class, it declares a static member variable that is shared among all instances of the class.
- **Static Member Functions:** Declares a function that belongs to the class rather than to any particular object instance. It can be called using the class name without creating an object.

Example:

```
class MyClass {
public:
    static int count; // Static member variable
    int normalVar;

    static void incrementCount() { // Static member function
        count++;
    }
};
int MyClass::count = 0; // Definition of static member variable

int main() {
    MyClass obj1, obj2;
    obj1.normalVar = 10;

    MyClass::incrementCount(); // Calling static member function using class name
    cout << "Count: " << MyClass::count << endl; // Accessing static member variable using class
name
    return 0;
}
```

Q30. How does C++ support multiple inheritances?
Ans: C++ supports multiple inheritances, allowing a class to inherit properties and behaviors from more than one base class. This means a derived class can inherit members from multiple parent classes.

Example:

```
class Base1 {
public:
    void display1() {
        cout << "Base1 display" << endl;
    }
};

class Base2 {
public:
    void display2() {
```

```
        cout << "Base2 display" << endl;
    }
};

class Derived : public Base1, public Base2 { // Multiple inheritance
public:
    void displayDerived() {
        cout << "Derived class display" << endl;
    }
};

int main() {
    Derived derivedObj;
    derivedObj.display1(); // Accessing function from Base1
    derivedObj.display2(); // Accessing function from Base2
    derivedObj.displayDerived();
    return 0;
}
```

In this example, the Derived class inherits from both Base1 and Base2, allowing it to access members from both parent classes.

These concepts and examples demonstrate various aspects of C++ programming, showcasing its features and functionalities.

Q31. Explain the concept of smart pointers in C++.
Ans: Smart pointers in C++ are classes that provide automatic memory management for dynamically allocated objects. They act as wrappers around raw pointers and ensure proper memory deallocation to prevent memory leaks.

Types of smart pointers:

- **std::unique_ptr:** It ensures exclusive ownership of an object and automatically deallocates the memory when the unique pointer goes out of scope.
- **std::shared_ptr:** It allows multiple pointers to share ownership of the same object. The object is only deallocated when all shared pointers are out of scope.
- **std::weak_ptr:** It is used in conjunction with std::shared_ptr to prevent cyclic dependencies and break strong reference cycles.

Example:

```
#include <memory>
#include <iostream>
int main() {
    // Using unique_ptr
    std::unique_ptr<int> uniquePtr(new int(5));
    std::cout << *uniquePtr << std::endl;
    // Using shared_ptr
```

```
    std::shared_ptr<int> sharedPtr1 = std::make_shared<int>(10);
    std::shared_ptr<int> sharedPtr2 = sharedPtr1; // Sharing ownership
    std::cout << *sharedPtr1 << " " << *sharedPtr2 << std::endl;
    return 0; // Smart pointers automatically deallocate memory when out of scope
}
```

Q32. Describe the use of function pointers in C++.
Ans: Function pointers in C++ store the address of functions, allowing dynamic selection and invocation of functions at runtime. They provide flexibility by enabling the use of different functions based on conditions or user choices.

Example:

```
#include <iostream>

void add(int a, int b) {
    std::cout << "Sum: " << a + b << std::endl;
}
void subtract(int a, int b) {
    std::cout << "Difference: " << a - b << std::endl;
}
int main() {
    void (*funcPtr)(int, int); // Declaration of function pointer
    int choice;
    std::cout << "Enter 1 for addition, 2 for subtraction: ";
    std::cin >> choice;

    if (choice == 1) {
        funcPtr = &add; // Assigning address of 'add' function
    } else if (choice == 2) {
        funcPtr = &subtract; // Assigning address of 'subtract' function
    }
    // Calling function through function pointer
    (*funcPtr)(10, 5); // Passing arguments to the selected function
    return 0;
}
```


Q33. What is the purpose of the ‘volatile’ keyword in C++?
Ans: The volatile keyword in C++ is used to indicate that a variable may be changed unexpectedly by external factors outside the control of the program. It tells the compiler not to optimize the code related to that variable, as its value might change unpredictably.

Purpose of volatile:

- **Preventing Compiler Optimization:** It ensures that the compiler does not optimize or make assumptions about the variable's value.
- **Handling Memory-Mapped Hardware:** Useful in embedded systems and device drivers where variables represent hardware registers whose values can change due to external factors.
- **Multithreading and Signal Handling:** In scenarios where variables can be modified by signals or multiple threads concurrently.

Example:

```
#include <iostream>
int main() {
    volatile int sensorValue; // Declaring volatile variable
    // Assume sensorValue is being updated by external hardware
    while (true) {
        // Using volatile variable in a loop
        if (sensorValue > 100) {
            std::cout << "Alert: Sensor value exceeded threshold" << std::endl;
        }
        // Without 'volatile', compiler may optimize the loop by caching the value
        // 'volatile' prevents such optimizations
    }
    return 0;
}
```

Q34. Explain the concept of 'RAII' (Resource Acquisition Is Initialization).
Ans: RAII is a programming idiom in C++ that associates the acquisition and release of resources with the lifecycle of an object. It ensures that resources (like memory, file handles, locks) are acquired when an object is created (initialized) and automatically released when the object goes out of scope (destructed), irrespective of exceptions or early returns.

Principles of RAII:

- Resource acquisition should occur during object creation (in the constructor).
- Resource release should happen during object destruction (in the destructor).
- Objects are responsible for managing resources they acquire.

Example:

```
#include <iostream>
#include <fstream>

class FileHandler {
private:
    std::ofstream file;
public:
    FileHandler(const std::string& filename) : file(filename) {
        if (!file.is_open()) {
```

```
            throw std::runtime_error("Unable to open file");
        }
    }
    ~FileHandler() {
        if (file.is_open()) {
            file.close(); // Release resource in destructor
        }
    }
    void writeToFile(const std::string& data) {
        file << data << std::endl;
    }
};
int main() {
    try {
        FileHandler handler("sample.txt"); // Resource acquired when object is created
        handler.writeToFile("RAII example"); // Using the file resource
    } catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
    // File resource automatically released when handler object goes out of scope
    return 0;
}
```

Q35. Describe the difference between shallow copy and deep copy.
Ans:

- **Shallow Copy:** Shallow copy involves copying the memory addresses of the data rather than duplicating the entire data. It creates a new object but copies the references of the dynamically allocated memory to the original object's memory. Both the original and copied objects point to the same memory locations, which can lead to issues if one object modifies the shared data.

Example of shallow copy:

```
#include <iostream>

class ShallowCopy {
private:
    int* data;

public:
    ShallowCopy(int val) {
        data = new int;
        *data = val;
    }

    // Shallow copy constructor
    ShallowCopy(const ShallowCopy& source) : data(source.data) {}
```

```
    int getData() const {
        return *data;
    }
    void setData(int val) {
        *data = val;
    }
    ~ShallowCopy() {
        delete data;
    }
};
int main() {
    ShallowCopy original(5);
    ShallowCopy shallowCopy(original); // Shallow copy

    std::cout << "Original data: " << original.getData() << std::endl;
    std::cout << "Shallow copy data: " << shallowCopy.getData() << std::endl;
    shallowCopy.setData(10);
    std::cout << "Original data after shallow copy modification: " << original.getData() <<
std::endl;
    std::cout << "Shallow copy data after modification: " << shallowCopy.getData() << std::endl;
    return 0;
}
```

- **Deep Copy:** Deep copy creates a complete copy of the data, including dynamically allocated memory. It allocates new memory locations and copies the values from the original object to the new object. This way, changes in one object do not affect the other, ensuring data independence.

Example of deep copy:

```
#include <iostream>

class DeepCopy {
private:
    int* data;
public:
    DeepCopy(int val) {
        data = new int;
        *data = val;
    }
    // Deep copy constructor
    DeepCopy(const DeepCopy& source) : data(new int(*source.data)) {}

    int getData() const {
        return *data;
    }
```

```
    void setData(int val) {
        *data = val;
    }
    ~DeepCopy() {
        delete data;
    }
};
int main() {
    DeepCopy original(5);
    DeepCopy deepCopy(original); // Deep copy

    std::cout << "Original data: " << original.getData() << std::endl;
    std::cout << "Deep copy data: " << deepCopy.getData() << std::endl;
    deepCopy.setData(10);

    std::cout << "Original data after deep copy modification: " << original.getData() << std::endl;
    std::cout << "Deep copy data after modification: " << deepCopy.getData() << std::endl;

    return 0;
}
```

Q36. Explain the concept of threading in C++.
Ans: Threading in C++ refers to the execution of multiple threads concurrently, allowing programs to perform multiple tasks simultaneously. A thread is a sequence of instructions that can execute independently within a process. Multithreading is utilized for tasks such as parallel processing, asynchronous operations, and improving program responsiveness.

Key concepts:

- **Thread:** A unit of execution within a process. Multiple threads can run simultaneously within the same program.
- **Concurrency:** The execution of multiple threads that appear to execute simultaneously.
- **Synchronization:** Mechanisms used to coordinate and control access to shared resources among threads.
- **Thread Safety:** Ensuring that shared data and resources are accessed in a way that avoids conflicts and race conditions.

Example:

```
#include <iostream>
#include <thread>
// Function to execute in a separate thread
void threadFunction(int id) {
    std::cout << "Thread ID: " << id << " running" << std::endl;
}
int main() {
    std::thread thread1(threadFunction, 1); // Creating a thread
```

```
    std::thread thread2(threadFunction, 2);
    // Joining threads to the main thread
    thread1.join();
    thread2.join();
    std::cout << "Main thread exiting" << std::endl;
    return 0;
}
```

In this example, two threads (thread1 and thread2) are created to execute the threadFunction concurrently with the main thread.

Q37. What are lambda expressions in C++? How are they used?

Ans: Lambda expressions in C++ are anonymous functions that can capture variables from their surrounding scope and be passed around as function objects. They provide a concise way to define small, inline functions, especially when using algorithms or working with function pointers.

Features of lambda expressions:

- Concise syntax for defining inline functions.
- Can capture variables from the surrounding scope by value or reference.
- Allow the creation of function objects (closures) that can be stored or passed as arguments to other functions.

Example:

```
#include <iostream>
int main() {
    // Lambda expression to calculate square of a number
    auto square = [](int x) { return x * x; };

    int num = 5;
    std::cout << "Square of " << num << " is: " << square(num) << std::endl;
    return 0;
}
```

Lambda expressions can also capture variables from the enclosing scope:

```
#include <iostream>
int main() {
    int factor = 10;
    // Lambda expression capturing 'factor' by value
    auto multiply = [factor](int x) { return x * factor; };
    int num = 5;
    std::cout << "Product of " << num << " and " << factor << " is: " << multiply(num) <<
std::endl;
    return 0;
}
```

Lambda expressions provide a convenient way to create small, disposable functions without explicitly defining a separate function. They are commonly used with algorithms, event handling, and other situations requiring a function object.

Q38. Explain the differences between stack and heap memory in C++.
Ans:

- **Stack Memory:**
  - **Characteristics:** Stack memory is a region of memory managed by the CPU. It's typically of fixed size and operates in a Last-In-First-Out (LIFO) manner.
  - **Usage:** Used for storing local variables, function call information (such as return addresses and parameters), and maintaining the call stack.
  - **Allocation/Deallocation:** Memory allocation and deallocation are automatic and handled by the compiler.
  - **Speed:** Access to stack memory is faster compared to heap memory.
  - **Lifetime:** Variables allocated on the stack have a limited lifetime within their respective scope.
- **Heap Memory:**
  - **Characteristics:** Heap memory is a larger pool of memory used for dynamic memory allocation. It is managed manually or automatically by the programmer.
  - **Usage:** Used for dynamic memory allocation, allowing objects to be allocated and deallocated at runtime.
  - **Allocation/Deallocation:** Memory allocation and deallocation are explicit (done using new, delete, malloc, free, etc.) and require manual management.
  - **Speed:** Access to heap memory is slower compared to stack memory due to dynamic allocation.
  - **Lifetime:** Objects allocated on the heap can exist beyond the scope where they were created.

```
int main() {
    // Stack allocation
    int stackVar = 10;

    // Heap allocation
    int* heapVar = new int(20);
    // Deleting heap memory
    delete heapVar;

    return 0;
}
```

In the above example, stackVar is allocated on the stack and heapVar is allocated on the heap using new. The memory for heapVar needs to be explicitly deallocated using delete to prevent memory leaks.

Q39. Describe the role of the 'mutable' keyword in C++.
Ans: In C++, the mutable keyword is used in a class to specify that a particular data member (usually declared as const) can be modified even within a const member function of that class. It allows changing the value of a mutable member even when the object itself is considered const.

Example:

```
class Example {
private:
    mutable int mutableVar;
public:
    Example(int value) : mutableVar(value) {}

    void modifyMutableVar() const {
        mutableVar = 20; // Allowed despite being in a const member function
    }

    int getMutableVar() const {
        return mutableVar;
    }
};
```

In this example, mutableVar is declared as mutable, allowing its modification within the const member function modifyMutableVar(). However, modifying other const data members within a const member function is not allowed.

Q40. What is a virtual destructor? Why is it needed?
Ans: A virtual destructor in C++ is a destructor declared with the virtual keyword in a base class. When a class is designed to be inherited and objects of derived classes are created, using a virtual destructor in the base class ensures proper destruction of derived class objects when they are deleted through a base class pointer.

Need for virtual destructors:

- When a derived class object is deleted through a base class pointer, the compiler invokes the destructor corresponding to the base class pointer's type.
- Without a virtual destructor in the base class, only the base class destructor is called. This results in incomplete cleanup of resources in the derived class, leading to memory leaks or undefined behavior.
- By making the base class destructor virtual, the destructor of the derived class is also called, ensuring proper cleanup of resources in both base and derived classes.

Example:

```
class Base {
public:
    virtual ~Base() {
        std::cout << "Base destructor" << std::endl;
```

```
    }
};
class Derived : public Base {
public:
    ~Derived() override {
        std::cout << "Derived destructor" << std::endl;
    }
};
int main() {
    Base* basePtr = new Derived();
    delete basePtr; // Deleting through a base class pointer

    return 0;
}
```

In this example, the ~Base() destructor is declared as virtual in the Base class. When delete basePtr is called, it correctly calls the ~Derived() destructor along with the ~Base() destructor, ensuring proper cleanup of resources in both classes.

Q41. Explain the concept of move semantics in C++.

Ans: Move semantics in C++ refers to the efficient transfer of resources (such as memory or ownership) from one object to another without unnecessary copying. It is implemented using move constructors and move assignment operators, introduced to optimize the performance of resource-intensive operations like dynamic memory allocation.

Key components of move semantics:

- **Move Constructor (&&)**: A constructor that allows the transfer of resources from an existing object to a new object using an rvalue reference (&&).
- **Move Assignment Operator (&&)**: An assignment operator that enables the transfer of resources from one object to another using an rvalue reference (&&).

Benefits of move semantics:

- Reduces unnecessary copying of resources, improving performance.
- Allows the efficient transfer of ownership or resources from temporary objects.

Example:

```
class MyData {
private:
    int* data;
public:
    // Move constructor
    MyData(MyData&& other) noexcept : data(other.data) {
        other.data = nullptr; // Resetting the source object's pointer
    }
    // Move assignment operator
```

```
    MyData& operator=(MyData&& other) noexcept {
        if (this != &other) {
            delete data; // Deleting the existing data
            data = other.data;
            other.data = nullptr; // Resetting the source object's pointer
        }
        return *this;
    }
    // Other constructors, destructor, and functionalities...
};
```

In this example, the move constructor and move assignment operator transfer the ownership of data from the source object to the target object, allowing efficient resource management during object transfers.

Q42. Describe the use of 'const' member functions in C++.

Ans: In C++, const member functions are member functions of a class that promise not to modify the state of the object on which they are called. Declaring member functions as const is a part of const correctness, ensuring that these functions do not change the object's data members.

Characteristics of const member functions:

- They are declared using the const keyword at the end of the function declaration.
- They can be called on both const and non-const objects of the class.
- They can't modify non-mutable data members or call non-const member functions within the class.

Example:

```
class MyClass {
private:
    int data;
public:
    // Constant member function that doesn't modify data
    int getData() const {
        // data = 10; // Error: Cannot modify data in a const member function
        return data;
    }

    // Non-const member function that can modify data
    void setData(int value) {
        data = value;
    }
};

int main() {
    const MyClass obj1; // Object of MyClass declared as const
    MyClass obj2;
```

```
    int value1 = obj1.getData(); // Can call const member function on const object
    int value2 = obj2.getData(); // Can call const member function on non-const object
    std::cout << "Value 1: " << value1 << ", Value 2: " << value2 << std::endl;

    // obj1.setData(5); // Error: Cannot call non-const member function on const object
    obj2.setData(5); // Can call non-const member function on non-const object

    return 0;
}
```

In the example, getData() is a const member function of the MyClass that can be called on both const and non-const objects. It promises not to modify the state of the object on which it is called.

Q43. What are C++ references? How are they different from pointers?
Ans:

- **C++ References:**
  - **Characteristics:** A reference in C++ is an alias or an alternative name for an existing variable. Once initialized, a reference cannot be re-bound to refer to another variable.
  - **Declaration:** Declared using & symbol after the data type.
  - **Null Reference:** References cannot be null and must be initialized when declared.
  - **Pointer vs. Reference:** References provide a simpler syntax and are safer than pointers in some cases but lack the flexibility and capabilities of pointers.
- **Pointers:**
  - **Characteristics:** Pointers are variables that store memory addresses. They can be re-assigned to point to different objects or null.
  - **Declaration:** Declared using * symbol.
  - **Null Pointer:** Pointers can be null or uninitialized, causing issues if not handled properly.
  - **Flexibility:** Pointers offer more flexibility in dynamic memory allocation and memory manipulation.

Q44. Explain the concept of abstract classes in C++.
Ans:

- **Abstract classes** in C++ are classes that cannot be instantiated directly. They serve as blueprints for other classes to inherit from and contain one or more pure virtual functions.
- **Pure virtual function**: A pure virtual function is a function declared within an abstract class but has no implementation.
- Abstract classes may contain both normal member functions and pure virtual functions.
- Classes that inherit from abstract classes must provide implementations for all pure virtual functions; otherwise, they will also become abstract.

Key points about abstract classes:

- Abstract classes cannot be instantiated; they exist to be inherited from.
- They often define an interface that derived classes must implement.

- Abstract classes can have data members, constructors, and concrete member functions alongside pure virtual functions.

Example:

```
class Shape {
public:
    virtual void draw() const = 0; // Pure virtual function
    virtual double area() const = 0; // Pure virtual function
    virtual ~Shape() {} // Virtual destructor
    void displayInfo() {
        std::cout << "This is a shape." << std::endl;
    }
};
class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing Circle." << std::endl;
    }
    double area() const override {
        // Implementation of area calculation for a circle
        return 3.14 * radius * radius;
    }
private:
    double radius;
};
```

In this example, Shape is an abstract class with two pure virtual functions draw() and area(). The Circle class inherits from Shape and provides concrete implementations for the pure virtual functions, making it a concrete class.

Q45. Describe the differences between function overloading and function overriding.
Ans:

- **Function Overloading:**
  - **Definition:** Function overloading involves defining multiple functions within the same scope but with different parameters or argument types.
  - **Signature:** Overloaded functions must have different signatures (different number or types of parameters).
  - **Resolution:** The appropriate function to be called is determined by the number and types of arguments during compilation (static polymorphism).
  - **Example:** void sum(int a, int b), void sum(double a, double b).
- **Function Overriding:**
  - **Definition:** Function overriding occurs in inheritance when a derived class provides a specific implementation for a function that is already defined in its base class.
  - **Inheritance:** Occurs in the context of inheritance with a base class and its derived classes.
  - **Signature:** The overriding function in the derived class must have the same signature (name and parameters) as the base class function.

- **Resolution:** The appropriate function to be executed is determined during runtime based on the object's actual type (dynamic polymorphism).

Example:

```
class Base {
public:
    virtual void display() const {
        std::cout << "Base display" << std::endl;
    }
};
class Derived : public Base {
public:
    void display() const override {
        std::cout << "Derived display" << std::endl;
    }
};
```

In function overloading, multiple functions with the same name exist in the same scope but with different parameters. In function overriding, a function in the derived class replaces a function in the base class to provide a specific implementation.

Q46. What are 'constexpr' variables in C++? How are they used?
Ans:

- constexpr in C++ is a keyword used to declare variables as constants that can be evaluated at compile time.
- constexpr variables must be initialized with a value that can be determined at compile time.
- They are suitable for defining constants or expressions whose values are known at compile time.
- constexpr can be used for variables, functions, and constructors.

Example:

```
constexpr int square(int x) {
    return x * x;
}
int main() {
    constexpr int side = 5; // Declaration of a constexpr variable
    constexpr int area = square(side); // Evaluation at compile time

    int nonConstVar = 10;
    constexpr int constFromNonConst = square(nonConstVar); // Error: non-constexpr variable

    return 0;
}
```

In this example, side and area are declared as constexpr variables and are evaluated at compile time. The function square() is marked as constexpr and computes the square of a number at compile time when the argument is a constexpr.

Q47. Explain the role of the 'volatile' keyword in multi-threading.
Ans: In C++, the volatile keyword indicates to the compiler that a variable's value might change unexpectedly, and the compiler should not optimize or cache the variable's value. In a multi-threading context, volatile helps in scenarios where variables might be modified by multiple threads or asynchronously by signals.

Role of volatile in multi-threading:

- **Preventing Compiler Optimizations:** volatile ensures that the compiler does not optimize away or cache the variable's value in registers or memory.
- **Memory Visibility:** In multi-threading, it helps in scenarios where variables are accessed and modified by multiple threads, ensuring that changes made by one thread are visible to other threads.
- **Access to Hardware Registers:** Useful in embedded systems where variables represent hardware registers that can change asynchronously.

Example:

```
#include <iostream>
#include <thread>
volatile bool flag = false;
void doWork() {
    while (!flag) {
        // Do some work until the flag is set
    }
    std::cout << "Flag is set, exiting thread." << std::endl;
}
int main() {
    std::thread worker(doWork);
    // Setting the flag after some time
    std::this_thread::sleep_for(std::chrono::seconds(2));
    flag = true;
    worker.join();
    return 0;
}
```

In this example, the volatile keyword ensures that changes made to the flag variable by one thread are immediately visible to other threads. It prevents the compiler from optimizing or caching the value of flag in a register or memory, ensuring correct behavior in a multi-threaded environment.

Q48. What is the purpose of the 'typeid' operator in C++?
Ans: The typeid operator in C++ is used to determine the data type of an object or an expression at runtime. It returns a reference to a std::type_info object that represents the type information of the specified object.

Purpose of typeid:

- **Run-time Type Identification:** Allows obtaining the type information of an object dynamically at runtime.
- **Useful for Polymorphism:** Helpful in scenarios where different derived classes are used through base class pointers or references.

Example:

```
#include <iostream>
#include <typeinfo>
class Base {
public:
    virtual ~Base() {}
};
class Derived : public Base {};
int main() {
    Base* basePtr = new Derived();

    // typeid operator to get type information
    const std::type_info& typeInfo = typeid(*basePtr);
    std::cout << "Object type: " << typeInfo.name() << std::endl;

    delete basePtr;
    return 0;
}
```

In this example, typeid(*basePtr) returns the type information of the object pointed to by basePtr. It's useful for determining the actual derived type when dealing with polymorphic objects through base class pointers.

Q49. Describe the concept of 'decltype' in C++.
Ans: The decltype specifier in C++ is used to deduce the type of an expression or variable at compile time without evaluating the expression itself. It helps in defining a variable or specifying a return type based on the type of an existing expression or variable.

Key points about decltype:

- **Type Deduction:** decltype deduces the type of an expression or variable.
- **Expression Evaluation:** It does not evaluate the expression but deduces its type.
- **Use Cases:** Useful in template metaprogramming, creating aliases, and specifying return types in functions.

- **Handling Reference and const/volatile Qualifiers:** decltype preserves reference types and const/volatile qualifiers.

Example:

```
#include <iostream>
int main() {
    int x = 5;
    const double& y = 10.5;
    decltype(x) newX = x; // Deduces the type of 'x' as 'int'
    decltype(y) newY = y; // Deduces the type of 'y' as 'const double&'
    std::cout << "Type of newX: " << typeid(newX).name() << std::endl;
    std::cout << "Type of newY: " << typeid(newY).name() << std::endl;
    return 0;
}
```

In this example, decltype(x) deduces the type of x as int, and decltype(y) deduces the type of y as const double&. It allows the programmer to create variables whose types mirror existing variables or expressions.

Q50. Explain the use of the 'std::move' function in C++.

Ans: The std::move function in C++ is used to explicitly convert a value to an rvalue reference, enabling the efficient transfer of resources (such as ownership) from one object to another. It is often used with move semantics to facilitate moving objects instead of making costly deep copies.

Purpose of std::move:

- **Enabling Move Semantics:** Allows objects to be moved (transfer ownership or resources) efficiently rather than copied.
- **Converting to Rvalue Reference:** Converts an object into an rvalue reference, making it eligible for move operations.
- **Optimizing Resource Management:** Useful for optimizing memory-intensive operations like moving large objects or containers.

Example:

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> source = {1, 2, 3, 4, 5};
    // Using std::move to transfer ownership
    std::vector<int> destination = std::move(source);
    std::cout << "Source size after move: " << source.size() << std::endl; // Output: 0
    std::cout << "Destination size after move: " << destination.size() << std::endl; // Output: 5

    return 0;
}
```

In this example, std::move is used to transfer the ownership of the source vector to the destination vector efficiently. After the move, source becomes empty, and destination contains the elements previously held by source.

Q51. What are the advantages of using the 'auto' keyword in C++?
Ans: The auto keyword in C++ allows the compiler to automatically deduce the data type of a variable based on its initialization expression. It offers several advantages:

- **Simplified Syntax:** Reduces verbosity and simplifies code by allowing the compiler to infer the type.
- **Ease of Maintenance:** Eases maintenance by making the code more readable and less error-prone during refactorings.
- **Compatibility with Templates:** Facilitates working with complex types, especially in template-based programming.
- **Enhanced Readability:** Improves code readability, especially with lengthy and complex types.

Example:

```
#include <iostream>
#include <vector>
int main() {
    // Using auto for deducing type
    auto x = 5; // int
    auto y = 3.14; // double
    auto z = "Hello, Auto!"; // const char*
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    for (auto num : numbers) {
        std::cout << num << " ";
    }

    return 0;
}
```

In this example, auto deduces the appropriate data types for variables x, y, and z based on their initialization expressions. Additionally, auto is used in a range-based for loop to simplify iterating through a std::vector<int>.

Q52. Describe the differences between static and dynamic polymorphism.
Ans:

- **Static Polymorphism:**
  - **Definition:** Achieved through function overloading, templates, and inheritance with function overriding (early binding).
  - **Resolution:** Determined at compile time, allowing the compiler to select the appropriate function based on the static type of the object.
  - **Examples:** Function overloading, templates, virtual functions with overridden implementations.

- **Dynamic Polymorphism:**
  - **Definition:** Achieved through inheritance and virtual functions (late binding).
  - **Resolution:** Determined at runtime based on the actual type of the object, allowing for different behaviors through base class pointers or references.
  - **Examples:** Virtual functions, inheritance with function overriding.

Key Differences:

- **Binding Time:** Static polymorphism involves early binding (compile-time), while dynamic polymorphism involves late binding (runtime).
- **Flexibility:** Dynamic polymorphism offers more flexibility, allowing different behaviors based on the runtime type of the object.
- **Performance:** Static polymorphism is generally more efficient as the function resolution is determined at compile time, whereas dynamic polymorphism incurs some runtime overhead due to virtual function calls.

Q53. Explain the concept of 'try', 'throw', and 'catch' in exception handling.
Ans:

- **try:** The try block is used to enclose the code that might throw exceptions. If an exception occurs within the try block, the control is transferred to the appropriate catch block.
- **throw:** The throw statement is used to raise an exception explicitly. It allows a program to signal that an exceptional condition has occurred.
- **catch:** The catch block is used to handle exceptions thrown by the try block. It catches and handles specific types of exceptions based on their type.

Example:

```
#include <iostream>
int main() {
    try {
        int numerator = 10;
        int denominator = 0;
        if (denominator == 0) {
            throw std::runtime_error("Division by zero error.");
        }

        int result = numerator / denominator;
        std::cout << "Result: " << result << std::endl;
    }
    catch (std::runtime_error& e) {
        std::cout << "Exception caught: " << e.what() << std::endl;
    }

    return 0;
}
```

In this example, the try block contains code that might throw an exception (division by zero). If an exception occurs, the control goes to the catch block, which handles the std::runtime_error exception and displays the error message.

Q54. What is the role of 'std::bind' in C++?
Ans: std::bind in C++ is used to create function objects by binding arguments to function parameters. It allows the creation of callable objects with a subset of parameters predefined, enabling delayed invocation or customization of function calls.

Key points about std::bind:

- **Function Adaptation:** Adapts functions or function objects with different signatures.
- **Partial Function Application:** Binds arguments to the parameters of a callable object, creating a new function object.
- **Delay Invocation:** Creates function objects that can be called later, with some or all parameters already set.
- **Customization:** Enables customization of function parameters at the time of creation.

Example:

```
#include <iostream>
#include <functional>
void display(int a, int b, int c) {
    std::cout << "a: " << a << ", b: " << b << ", c: " << c << std::endl;
}
int main() {
    auto bindFunc = std::bind(display, std::placeholders::_1, 20, std::placeholders::_2);
    bindFunc(10, 30); // Calls display(10, 20, 30)

    return 0;
}
```

In this example, std::bind is used to create a function object bindFunc from the display function, binding arguments to parameters. It creates a callable object with 20 as the second argument fixed and the first and third arguments open to be passed later.

Q55. Describe the concept of 'CRTP' (Curiously Recurring Template Pattern).
Ans: The Curiously Recurring Template Pattern (CRTP) is an idiom in C++ where a class Base template is parameterized by a derived class Derived. The derived class inherits from the base class by providing itself as a template argument to the base class.

Key features of CRTP:

- **Static Polymorphism:** Provides a form of compile-time polymorphism.
- **Facilitates Code Reuse:** Allows implementing common functionality in the base class, reused by multiple derived classes.
- **Enables Method Injection:** Allows derived classes to inject their behavior into the base class by defining the base class's methods.

Example:

```
template <typename Derived>
class Base {
public:
    void commonFunction() {
        static_cast<Derived*>(this)->specificFunction();
    }
    // More functionalities...
};
class DerivedClass : public Base<DerivedClass> {
public:
    void specificFunction() {
        // Implementation specific to DerivedClass
    }
    // More functionalities...
};
int main() {
    DerivedClass derivedObj;
    derivedObj.commonFunction(); // Calls specificFunction() through CRTP

    return 0;
}
```

In this example, Base is a template class using CRTP where DerivedClass is derived from Base by providing itself (DerivedClass) as a template argument. Base contains a method commonFunction() that delegates its call to specificFunction() implemented in the derived class.

Q56. Explain the concept of 'std::forward' in C++.

Ans: std::forward in C++ is used in the context of perfect forwarding within templates. It enables the forwarding of arguments while preserving their value category (lvalue or rvalue) as received by the forwarding function.

Purpose of std::forward:

- **Preserving Value Category:** Allows forwarding of arguments received by a function as either lvalues or rvalues to another function.
- **Used in Templates:** Primarily used in forwarding arguments in template functions, especially in forwarding references (T&&) to maintain their original value category.
- **Supports Perfect Forwarding:** Facilitates forwarding arguments exactly as received without unnecessary conversions or losing their original type.

Example:

```
#include <iostream>
#include <utility>
void process(int& x) {
```

```
    std::cout << "Processing lvalue: " << x << std::endl;
}
void process(int&& x) {
    std::cout << "Processing rvalue: " << x << std::endl;
}
template<typename T>
void wrapper(T&& arg) {
    process(std::forward<T>(arg));
}
int main() {
    int value = 10;
    wrapper(value); // Calls process(int&)
    wrapper(20);    // Calls process(int&&)
    return 0;
}
```

In this example, std::forward preserves the value category (lvalue or rvalue) of the argument passed to wrapper() when forwarding it to the process() function. This maintains the original nature of the argument within the forwarding process.

Q57. Describe the purpose of the 'std::unique_ptr' and 'std::shared_ptr' in C++.
Ans:

- **std::unique_ptr:**
  - **Ownership:** std::unique_ptr is a smart pointer that manages the ownership of dynamically allocated objects with unique ownership.
  - **Single Ownership:** Ensures that only one std::unique_ptr can own the resource at a time.
  - **Automatic Cleanup:** Automatically deallocates the memory (calls delete) when the std::unique_ptr goes out of scope.
  - **Move-Only:** Supports move semantics, enabling the transfer of ownership between std::unique_ptr instances.
- **std::shared_ptr:**
  - **Ownership:** std::shared_ptr is a smart pointer that allows multiple pointers to share ownership of the same dynamically allocated object.
  - **Reference Counting:** Keeps track of the number of std::shared_ptr instances pointing to the same resource using a reference count.
  - **Automatic Cleanup:** Deallocates the memory (calls delete) when the last std::shared_ptr pointing to the resource goes out of scope.
  - **Thread Safety:** Provides thread-safe reference counting for shared ownership across threads.

Example:

```
#include <iostream>
#include <memory>
int main() {
```

```
    // std::unique_ptr example
    std::unique_ptr<int> uniquePtr(new int(42));
    std::cout << "Unique pointer value: " << *uniquePtr << std::endl;
    // std::shared_ptr example
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(100);
    std::cout << "Shared pointer value: " << *sharedPtr << std::endl;
    return 0;
}
```

In this example, std::unique_ptr and std::shared_ptr are used to manage dynamically allocated memory. std::unique_ptr is used for exclusive ownership, while std::shared_ptr allows shared ownership among multiple pointers.

Q58. What is the 'std::function' in C++?

Ans: std::function in C++ is a general-purpose polymorphic function wrapper that can store, copy, and invoke callable entities such as functions, function pointers, lambdas, or other function-like objects.

Features of std::function:

- **Polymorphic Function Wrapper:** Stores and manages callable objects with different signatures.
- **Type Erasure:** Hides the underlying function signature by providing a common interface to call any callable entity.
- **Flexibility:** Can hold any callable entity that matches its specified signature.
- **Usage:** Useful for passing functions as arguments, callbacks, and managing callbacks in event-driven programming.

Example:

```
#include <iostream>
#include <functional>
void printMessage(const std::string& message) {
    std::cout << "Message: " << message << std::endl;
}
int main() {
    std::function<void(const std::string&)> functionPointer = printMessage;

    functionPointer("Hello, std::function!");
    return 0;
}
```

In this example, std::function is used to store a function pointer to printMessage. It demonstrates the flexibility of std::function to store and invoke callable objects with different signatures.

Q59. Explain the concept of 'constexpr' functions in C++.

Ans: constexpr functions in C++ are functions that can be evaluated at compile time. They allow computations to be performed during compilation rather than runtime when called with constant expressions as arguments.

Key points about constexpr functions:

- **Compile-Time Evaluation:** Enable evaluation of functions at compile time if provided with constant expressions.
- **Constants and Expressions:** Suitable for computing values of constants or expressions that are known at compile time.
- **Restrictions:** Must adhere to certain restrictions such as having a body consisting of a single return statement and only performing operations that are allowed in constant expressions.
- **Optimization:** Can significantly improve performance by allowing computations to occur at compile time rather than runtime.

Example:

```
#include <iostream>
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
int main() {
    constexpr int result = factorial(5); // Evaluated at compile time
    std::cout << "Factorial of 5: " << result << std::endl;

    return 0;
}
```

In this example, factorial() is a constexpr function that calculates the factorial of a number. When called with a constant expression (factorial(5)), the result is computed at compile time.

Q60. Describe the use of 'std::thread' in C++ multithreading.
Ans: std::thread in C++ is a class provided by the standard library to create and manage concurrent threads of execution. It allows developers to create and manage multiple threads within a C++ program to perform tasks concurrently.

Key features and usage of std::thread:

- **Thread Creation:** std::thread creates a new thread of execution and associates it with a callable object (function, functor, lambda).
- **Concurrency:** Enables concurrent execution of multiple threads, performing tasks in parallel to utilize multi-core CPUs effectively.
- **Synchronization:** Facilitates synchronization between threads using synchronization primitives like mutexes, condition variables, etc.
- **Joining and Detaching:** std::thread instances can be joined (wait for the thread to finish) or detached (allow thread to execute independently).
- **Thread Management:** Provides functionality to query and modify thread attributes such as ID, hardware concurrency, etc.

```
#include <iostream>
#include <thread>
```

```
void printNumbers(int start, int end) {
    for (int i = start; i <= end; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
int main() {
    std::thread t1(printNumbers, 1, 5); // Creating a thread t1

    // Other operations in the main thread
    std::cout << "Main thread doing something else..." << std::endl;

    t1.join(); // Wait for t1 to finish

    return 0;
}
```

In this example, std::thread is used to create a thread t1 that executes the printNumbers() function to print numbers from 1 to 5. The main thread continues its execution, demonstrating concurrent execution.

Q61.What are the different Data Types present in C++?

Ans: **The 4 data types in C++, are**

1. **Primitive Datatype(basic datatype). Example- char, short, int, float, long, double, bool, etc.**
2. **Derived datatype. Example- array, pointer, etc.**
3. **Enumeration. Example- enum**
4. **User-defined data types. Example- structure, class, etc.**

Q62. What are References in C++?

Ans:A variable becomes an alias of an existing variable when it is described as a reference. To put it simply, a referred variable is a named variable that is another variable that already exists, with the understanding that any changes made to the reference variable would also affect the previously existing variable. A reference variable has a '&' before it.

Syntax:

```
 int DNT = 10;

// reference variable
int& ref = DNT;
```

Q63. Define token in C++?

In C++, a token is a fundamental unit of source code, representing individual elements recognized by the compiler during lexical analysis:

- Identifiers: Names for variables, functions, classes, etc.
- Keywords: Reserved words with specific meanings, e.g., if, else, int.
- Literals: Constants like numbers (42, 3.14) or strings ("hello").
- Operators: Symbols for mathematical (+, -) or logical (&&, ||) operations.
- Comments: Single-line (//) or multi-line (/* */) comments ignored by the compiler.
- Whitespace: Spaces, tabs, and newlines separating tokens but generally not stored for further processing.

Q64.Is Deconstructor Overloading Possible? If Yes then Explain and if no then why?

Overloading a destructor is not feasible. There is just one method to delete an object since destroyers don't accept parameters. Destructor overloading is therefore not feasible.

Q65.What are the Static Members and Static Member Functions in C++?

When a class variable is marked static, memory is set aside for it for the duration of the program. There is only one copy of the static member, regardless of how many objects of that class have been produced. This means that all of the objects in that class can access the same static member. Even in the absence of any class objects, a static member function can still be invoked since it can be reached with just the class name and the scope resolution operator::

Q66.Difference between equal to (==) and assignment operator(=)?

- The equal to operator == determines if two values are equal. It returns false otherwise; if they are equal, then it is true.
- The assignment operator = gives the left operand the value of the right-side expression.

Q67.What is Loops in C++? Explain different types of loops in C++?

In C++, loops are control structures that allow the execution of a block of code repeatedly as long as a specified condition is true. There are three types of loops in C++:

- For Loop: Executes a block of code a specified number of times, with an initialization, condition check, and increment/decrement statement.
- While Loop: Executes a block of code as long as a specified condition is true.
- Do-While Loop: Similar to the while loop, but ensures the code block is executed at least once before checking the condition.

Q68.What is the difference between virtual functions and pure virtual functions in C++?

Virtual functions and pure virtual functions are concepts in C++ related to polymorphism and inheritance:

Virtual Functions:

- Declared in the base class with the virtual keyword.
- Can be overridden in derived classes.
- Provides a default implementation in the base class.
- Objects of the derived class can be used through pointers or references of the base class type.

Pure Virtual Functions:

- Declared in the base class with the virtual keyword and set to 0.
- Has no implementation in the base class.
- Forces derived classes to provide an implementation.
- Classes containing pure virtual functions are abstract and cannot be instantiated.

Q69.When should we use multiple inheritance in C++?

Multiple inheritance in C++ should be used cautiously and in specific scenarios where it simplifies the design without introducing complexity. It is appropriate when a class needs to inherit properties and behaviors from more than one unrelated class. For example, in GUI frameworks a class may inherit from both a graphical component class and an event handling class. However, it requires careful planning to avoid ambiguity issues, and alternative design patterns like composition or interfaces should be considered to enhance code readability and maintainability.

Q70.Which operations are permitted on pointers?

The variables used to hold the address position of another variable are called pointers. The following operations on a pointer are allowed:

- Increment/Decrement of a Pointer
- Addition and Subtraction of integer to a pointer
- Comparison of pointers of the same type

Q71.How delete [] is different from delete in C++?

In C++, delete and delete[] are used to deallocate memory previously allocated with new and new[] operators, respectively. The key differences are:

Memory Allocation Type:

- delete is used for single objects allocated with new.
- delete[] is used for arrays of objects allocated with new[].

Deallocation Process:

- delete calls the destructor of the single object before deallocating memory.
- delete[] calls the destructors of all objects in the array before deallocating the memory block.

Q72.Can you compile a program without the main function?

Indeed, a program may be compiled without a main() call. For instance, Use Macros that define the main

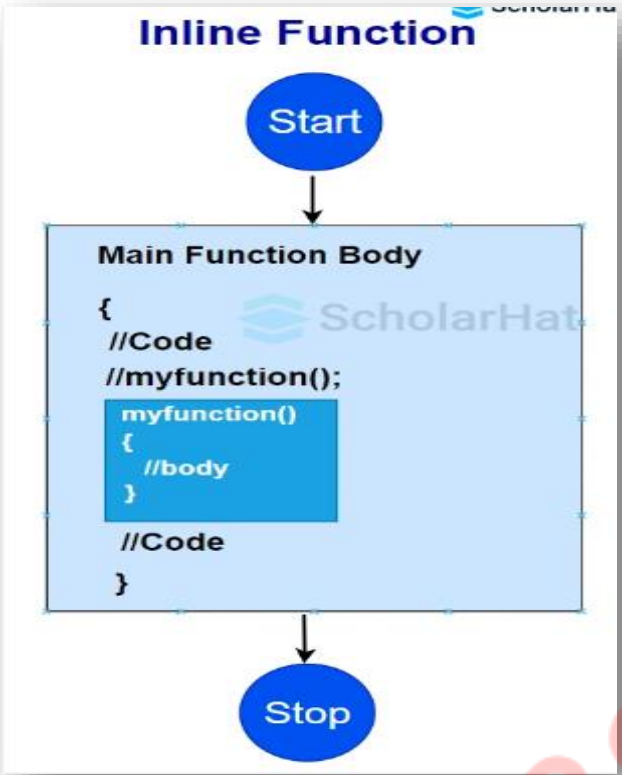
Example

```
// C++ program to demonstrate the
// a program without main()
#include

#define fun main
int fun(void)
{
printf("ScholarHat");
return 0;
}
```


Q73.Define inline function. Can we have a recursive inline function in C++?

In C++, an inline function is a function that is expanded in place where it is called, instead of being executed through a regular function call mechanism. It is declared with the inline keyword and helps reduce the function call overhead by inserting the function's code directly at the call site



Syntax

```
inline data_type function_name()
{
Body;
}
```

The answer is No; It cannot be recursive.
An inline function cannot be recursive because it simply loads the code into the location from where it is called, without keeping track of any information on the stack that would be required for recursion.
Additionally, the compiler will automatically disregard an inline keyword placed in front of a recursive function as it only interprets inlines as suggestions.

Q74.What is the difference between C and C++? The main difference between C and C++ are provided in the table below:

C	C++
C is a procedure-oriented programming language.	C++ is an object-oriented programming language.
C does not support data hiding.	Data is hidden by encapsulation to ensure that data structures and operators are used as intended.
C is a subset of C++	C++ is a superset of C.
Function and operator overloading are not supported in C	Function and operator overloading is supported in C++
Namespace features are not present in C	Namespace is used by C++, which avoids name collisions.
Functions can not be defined inside structures.	Functions can be defined inside structures.
calloc() and malloc() functions are used for memory allocation and free() function is used for memory deallocation.	new operator is used for memory allocation and deletes operator is used for memory deallocation.

Q75.What is the difference between struct and class?

In C++ a structure is the same as a class except for a few differences like security. The difference between struct and class are given below:

Structure	Class
Members of the structure are public by default.	Members of the class are private by default.
When deriving a struct from a class/struct, default access specifiers for base class/struct are public.	When deriving a class, default access specifiers are private.

Q76.How do you allocate and deallocate memory in C++?

The new operator is used for memory allocation and deletes operator is used for memory deallocation in C++.

For example-

```
int value=new int;          //allocates memory for storing 1 integer
delete value;               // deallocates memory taken by value

int *arr=new int[10];       //allocates memory for storing 10 int
delete []arr;               // deallocates memory occupied by arr
```

Q77.why C++ called as an object oriented

1. Because C++ language views the problem in the term of object oriented involved rather then the procedure for doing it.

2. Object oriented programming (OOP) is a programming paradigm based on the concept of “objects” which contain data in the form of field, often known as attributes. and codes in the form of procedure, often known as method.

Q78.What are void pointers?

A void pointer is a pointer which is having no datatype associated with it. It can hold addresses of any type.

For example:

```
void *ptr;

char *str;

 p=str; // no error

str=p; // error because of type mismatch
```

We can assign a pointer of any type to a void pointer but the reverse is not true unless you typecast it as

```
 str=(char*) ptr
```

Q79.Compare compile time polymorphism and Runtime polymorphism

The main difference between compile-time and runtime polymorphism is provided below:

Compile-time polymorphism	Run time polymorphism
In this method, we would come to know at compile time which method will be called. And the call is resolved by the compiler.	In this method, we come to know at run time which method will be called. The call is not resolved by the compiler.
It provides fast execution because it is known at the compile time.	It provides slow execution compared to compile-time polymorphism because it is known at the run time.
It is achieved by function overloading and operator overloading.	It can be achieved by virtual functions and pointers.
Example - `int add(int a, int b){ return a+b; } int add(int a, int b, int c){ return a+b+c; } int main(){ cout<<add(2,3)<<endl; cout<<add(2,3,4)<<endl; return 0; }`	Example - `class A{ public: virtual void fun(){ cout<<"base "; } }; class B: public A{ public: void fun(){ cout<<"derived "; } }; int main(){ A *a=new B; a->fun(); return 0; }`

Q80.What is the difference between shallow copy and deep copy?

The difference between shallow copy and a deep copy is given below:

Shallow Copy	Deep Copy
Shallow copy stores the references of objects to the original memory address.	Deep copy makes a new and separate copy of an entire object with its unique memory address.
Shallow copy is faster.	Deep copy is comparatively slower.
Shallow copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object

Q81.Differences between References and Pointers:

- References cannot be null, while pointers can be null or uninitialized.
- Once a reference is initialized, it cannot refer to another object. Pointers can be reassigned to point to different objects.
- References must be initialized when declared. Pointers can be declared without initialization.
- References use simpler syntax, often leading to more readable code, especially in function calls and pass-by-reference scenarios.

Example:

```
#include <iostream>
int main() {
    int value = 10;
    int& ref = value; // Reference initialization
    int* ptr = &value; // Pointer initialization
    ref = 20; // Modifying value through reference
    *ptr = 30; // Modifying value through pointer
    std::cout << "Value: " << value << std::endl; // Output: Value: 30
    return 0;
}
```

In this example, both ref and ptr are used to modify the value variable. ref is a reference, while ptr is a pointer, demonstrating their usage and differences.

Q82. Difference b/w scanf & printf comparing with cat & cin ?

1. Printf & scanf both are library functions

1. Cout and cin both are objects of classes

2. Printf and scanf needs format specifiers

2. Cout and cin doesn't require specifiers

3. Printf and scanf returns value.

3. Cout and cin doesn't return any value.

4.printf and scanf doesn't support user defined variable directly to scan & printf.

4.cout & cin does support user defined variable directly to scanf & printf with help of operator overloading.