1->answer:  
Object-oriented programming (OOP) was developed as a response to some of the drawbacks and limitations of structured programming. Here are some of the drawbacks of structured programming that OOP addresses:

1. Data encapsulation: Structured programming relies on global data which can lead to issues with data integrity and security. OOP can limit data access by using the concept of data encapsulation and abstraction which enhance data security and reduces the likelihood of unintended data modification.
2. Code reusability: structured programming achieves code reuse through function and procedure, where as OOP introduces the concept of classes and objects, which enables the creation of reusable code modules that can be easily incorporated in different parts of an application. This promotes a higher level of code reusability.
3. Manageability: structured programming can lead to convoluted code with interrelated functions. OOP divides a large program into several manageable objects, which naturally represent real-world entities. This makes it easier to manage and understand complex system by breaking them into smaller more manageable components.
4. Flexibility: in structured programming adding new features or modifying existing ones can be challenging. OOP promotes the use of inheritance, polymorphism, and encapsulation, which make it easier to extend and modify the code without affecting the existing ones.
5. Code reusability and understanding (real world object representation): It is difficult to relate structured programming code to real world objects but In OOP a car is an object. the car has the attributes such as weight and color, and methods, such as drive and brake. This is especially beneficial for developers who need to work with or modify existing code.

2->ans:

Encapsulation: the wrapping up of data and functions into a single unit (called class) is known as encapsulation. It is the most striking feature of a class. It restricts direct access to the data to the outside world, and only those function s which are wrapped in the class can access it. This insulation of data is called data hiding or information hiding.

3-> ans:

In C++, an access specifier is a keyword used to specify the visibility and accessibility of class members (variables and functions) from outside the class. Access specifiers determine how class members can be accessed and modified by other parts of the program.

C++ provides three main types of access specifiers:

1. Public: Members declared as public are accessible from anywhere in the program, including outside the class. Public members are typically used to define the interface of a class, making them accessible for external code to use.
2. Private: Members declared as private are only accessible within the class itself and are not accessible from outside the class. Private members are used to encapsulate the internal implementation details of the class and are not visible to external code.
3. Protected: Members declared as protected have a level of visibility between public and private. Protected members are accessible within the class and its derived classes (in the context of inheritance), but not from outside the class or unrelated classes.

4-> ans:

A pointer is a special type of a variable that contains a memory address instead of values. A pointer is declared with specific datatype so it can only point to variable with same datatype. A pointer name is always preceded by’\*’ operator. We use pointer for effective memory management, to manipulate array, to apply the call by reference function, to increase program execution speed, for dma and to decrease the length of the program.

Here's a breakdown of pointers in C++:

Declaration:

To declare a pointer, you use the \* symbol followed by the data type of the variable it will point to. For example:

int\* intPtr; // Declaring an integer pointer

double\* doublePtr; // Declaring a double pointer

Initialization:

Pointers must be initialized with the address of another variable before they can be used. You can initialize a pointer during declaration or later in your code.

int x = 10;

int\* ptr = &x; // Initializing ptr with the address of x

Dereferencing:

To access the value pointed to by a pointer, you use the dereference operator (\*). This operator retrieves the value stored at the memory location pointed to by the pointer.

int value = \*ptr; // Accessing the value pointed to by ptr (value will be 10)

Pointer Arithmetic:

Pointers can be incremented or decremented to move to the next or previous memory location. The actual increment depends on the data type the pointer points to.

int\* ptr = /\* some address \*/;

ptr++; // Moves ptr to the next int-sized memory location

Null Pointers:

Pointers can be set to a special value called a "null pointer" to indicate that they do not currently point to a valid memory location. This is often used for error handling.

int\* ptr = nullptr; // Initialize ptr as a null pointer

Dynamic Memory Allocation:

Pointers are commonly used with the new and delete operators to allocate and deallocate memory on the heap. This allows you to create objects with a lifetime that extends beyond the scope of the current function.

int\* dynamicPtr = new int; // Allocate an integer on the heap

\*dynamicPtr = 42; // Assign a value to the dynamically allocated integer

delete dynamicPtr; // Deallocate the memory when done

Arrays and Pointers:

Arrays in C++ are closely related to pointers. An array variable can be thought of as a pointer to the first element of the array.

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

int\* arrPtr = arr; // arrPtr points to the first element of arr

Pointer to Functions:

C++ allows you to declare pointers to functions, which can be used to dynamically call different functions at runtime.

int add(int a, int b) {

return a + b;

}

int (\*funcPtr)(int, int) = add; // Pointer to a function

int result = funcPtr(3, 4); // Calls the 'add' function through funcPtr

5-> Static variables and static member functions in C++ have unique characteristics and use cases that differentiate them from regular (non-static) variables and member functions. Here's an explanation of the use of static variables and static member functions:

Static Variables (Static Data Members):

Lifetime and Scope:

Static variables are associated with the class rather than with individual objects (instances) of the class.

They have a lifetime that extends throughout the program's execution, from the start to the end of the program.

Static variables are stored in a global storage area, separate from the individual object's memory.

Shared Data:

Static variables are shared among all instances (objects) of the class. They maintain a single copy of data that can be accessed and modified by any object of the class.

They are useful for storing data that should be common to all instances, such as class-wide counters, constants, or configuration settings.

Initialization:

Static variables are typically initialized at the time of declaration or can be initialized outside the class definition.

They are initialized only once, regardless of how many instances of the class are created.

Accessing Static Variables:

Static variables are accessed using the class name followed by the scope resolution operator (::) or through an object of the class.

They can also be accessed without creating an instance of the class.

class MyClass {

public:

static int staticVar; // Declaration of a static variable

};

int MyClass::staticVar = 0; // Initialization of the static variable

int main() {

MyClass::staticVar = 42; // Accessing the static variable using the class name

MyClass obj;

obj.staticVar = 10; // Accessing the static variable through an object

return 0;

}

Static Member Functions:

No Access to Non-Static Members:

Static member functions are associated with the class, not with individual objects, and they can only access static data members and other static member functions.

They do not have access to non-static (instance-specific) members, including instance variables.

Access Without Object Creation:

Static member functions can be called using the class name without the need to create an object.

This makes them useful for utility functions or operations that don't depend on specific object state.

Usage Examples:

Utility functions: Static member functions can be used for operations that do not require object-specific data. For example, a math library class might have static functions for mathematical operations.

Factory methods: Static member functions can be used to create and return instances of the class, often as an alternative to constructors.

Counters and trackers: Static member functions can be used to maintain counts or track certain class-wide statistics.

class MyClass {

public:

static int staticVar;

static void staticFunction() {

staticVar++;

}

};

int MyClass::staticVar = 0;

int main() {

MyClass::staticFunction(); // Calling a static member function

MyClass obj;

obj.staticFunction(); // Also calling the static member function through an object

return 0;

}

1. Data Encapsulation: OOP enhances data security and integrity by encapsulating data within objects.
2. Code Reusability: OOP promotes code reuse through classes and objects.
3. managing Complexity: OOP simplifies complex systems by breaking them into smaller, manageable objects.
4. Flexible Code Modification: OOP makes it easier to extend and modify code without affecting existing functionality.
5. Real-World Modeling: OOP allows developers to model real-world entities and their relationships accurately.
6. Code Organization: OOP reduces clutter and complexity by encapsulating data and behavior within objects.
7. Collaboration: OOP facilitates collaboration among developers by reducing naming conflicts and ensuring data consistency.
8. Maintenance: OOP simplifies code maintenance by focusing on specific objects and interactions.
9. Code Readability: OOP improves code readability and understandability with meaningful object-oriented constructs