```
    };
};
```

Purpose and Usage:

- Encapsulation: Nested classes help in encapsulating related functionality within the scope of the enclosing class, reducing namespace pollution and improving code organization.
- Access to Enclosing Class Members: Nested classes have access to the private members of the enclosing class, allowing them to manipulate the state of the enclosing object.
- Information Hiding: They can be used to hide implementation details and provide a cleaner interface to the users of the enclosing class.

2.9 Local Classes

Local classes in C++ are classes that are defined within the scope of a function or a block. Local classes have limited local scope and are accessible only within the function or block in which they are defined. Local classes can be used to encapsulate functionality that is only relevant within a specific scope.

Example:

```
void myFunction() {
    // Local class definition
    class LocalClass {
    public:
        void localMethod() {
            cout << "Inside local method" << endl;
        }
    };

    LocalClass obj;
    obj.localMethod(); // Accessing method of local class
}
```

NOTES NEO

Purpose and Usage:

- Limited Scope: Local classes have a limited scope and are only accessible within the block or function where they are defined, reducing namespace pollution.
- Encapsulation: They can encapsulate functionality that is only relevant within a specific scope, improving code organization and modularity.
- Information Hiding: Local classes can hide implementation details and provide a cleaner interface to the surrounding code.

2.10 Abstract Classes

Abstract classes in C++ are classes that cannot be instantiated on their own and typically contain at least one pure virtual functions. They serve as base classes or blueprints for other derived classes and define an interface that derived classes must implement.
Pure virtual functions are declared with the virtual keyword and have no implementation (no function body).

Syntax:

```
class AbstractClass {
public:
    virtual void pureVirtualFunction() = 0; // Pure virtual function
};
```

Example:

```
class Shape {
public:
    virtual double area() const = 0; // Pure virtual function
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14159 * radius * radius;
    }
};
```

Purpose and Usage:

- Defining Interfaces: Abstract classes define interfaces for derived classes, specifying a set of methods that must be implemented.
- Enforcing Polymorphism: They provide a way to achieve runtime polymorphism through dynamic binding and function overriding.
- Providing Frameworks: Abstract classes can be used to define frameworks or templates for similar types of objects, providing a common structure and behavior.f

2.11 Container Classes

Container classes in C++ are classes that are used to store and manage collections of other objects, known as elements, in an organized manner. They provide various methods for accessing, inserting, removing, and manipulating elements within the container.
Standard template library (STL) provides a variety of container classes, such as vectors, lists, sets, and maps.

Example of Container Classes:

1. Vector: A dynamic array that can resize itself automatically.
2. List: A doubly linked list that allows for efficient insertion and deletion operations.
3. Stack: A Last-In-First-Out (LIFO) data structure that allows for push and pop operations.
4. Queue: A First-In-First-Out (FIFO) data structure that allows for enqueue and dequeue operations.
5. Map: An associative container that stores key-value pairs and allows for efficient retrieval of values based on keys.
6. Set: A container that stores unique elements in a sorted order.

Example (Using std::vector):

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

int main() {
    // Creating a vector container
    vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);

    // Accessing elements of the vector
    for (int num : numbers) {
        cout << num << " ";
    }
    return 0;
}
```

Purpose and Usage:

- Data Management: Container classes provide efficient ways to store and manage collections of objects, facilitating data organization and manipulation.
- Dynamic Storage: They dynamically resize to accommodate varying numbers of elements, allowing for flexible storage of data.
- Standardized Interfaces: Container classes in the STL provide standardized interfaces and algorithms for working with collections, promoting code reusability and interoperability.

2.12 Bit Field and Classes

Bit fields in C++ are used to represent and manipulate groups of bits within a data structure, allowing for efficient use of memory and bitwise operations. They are typically used within classes to pack multiple boolean or integer flags into a single storage unit.

- **Definition:** Bit fields are declared within a class using integer data types, specifying the number of bits to allocate for each field.
- **Purpose:** Bit fields are used to conserve memory by efficiently packing multiple boolean flags or integer values into a single storage unit, reducing memory overhead.
- **Usage:**
  - Bit fields are accessed and manipulated using bitwise operators such as bitwise AND (&), bitwise OR (|), bitwise XOR (^), and bitwise shifts (<< and >>).
  - They are commonly used to represent sets of boolean flags, where each flag corresponds to a single bit within the bit field.
- **Syntax:**

```
class BitFieldClass {
public:
    unsigned int flag1 : 1; // Bit field with 1 bit
    unsigned int flag2 : 1; // Bit field with 1 bit
    unsigned int value : 8; // Bit field with 8 bits
};
```

- **Example:**

```
class Permissions {
public:
    unsigned int read : 1;    // Bit field for read permission
    unsigned int write : 1;   // Bit field for write permission
    unsigned int execute : 1; // Bit field for execute permission
};

int main() {
    Permissions file1;
    file1.read = 1;
    file1.write = 0;
    file1.execute = 1;

    // Check if read permission is granted
    if (file1.read) {
        std::cout << "Read permission granted." << std::endl;
    }

    return 0;
}
```

Purpose and Usage:

- Memory Efficiency: Bit fields allow for efficient use of memory by packing multiple variables into a single storage unit, reducing memory overhead.
- Compact Representation: They provide a compact representation of data, especially for flags, status bits, or other binary values.
- Improved Readability: Bit fields can improve code readability by providing a concise way to define and access binary data within a class.

MDU PYQs (Short)

1. Explain the following:
    a. Encapsulation
    b. Abstract class
    c. Static Members
    d. Const keyword
    e. Member function
    f. Preprocessor directives
    g. Header files and library files
    h. Access/Visibility Specifiers
    i. Class and object
2. How main() function in C++ is different from main() function in C?
3. What is the need of declaring a member of class static ?
4. Explain controlling access function and utility function with example.
5. What is the use of scope resolution operator?
6. How can we define a class? How can we access class members?
7. What is abstraction? Explain with example.
8. What is an interface? How it is different from class?
9. What is in encapsulation? Explain with example.
10. What are nested classes ? Explain.

MDU PYQs (Long)

1. In what ways Object oriented programming paradigm is better than a structured programming? Explain features of OOPs.
2. Explain what features of C++ makes it different from C.
3. What are preprocessor directives. Why are they used? Explain any two preprocessor directives.
4. Explain the concept of re-usability with example.
5. Differentiate between procedural and object oriented programming language.
6. What do you mean by library files? Explain with suitable examples.