Course Code	**:**	**BCS-031**
Course Title	**:**	**Programming in C++Assignment**
Number	**:**	**BCA(III)031/Assignment/2024-25**
Maximum Marks	**:**	**100**
Weightage	**:**	**25%**
Last Date of Submission	**:**	**31st October,2024(for July session)**
		30th April,2025(for January session)

This assignment has three questions carrying a total of 80 marks. Answer all the questions. Rest 20 marks are for viva-voce. You may use illustrations and diagrams to enhance explanations. Please go through the guidelines regarding assignments given in the Programme Guide for the format of presentation. Wherever required, you may write C++ program and take its output as part of solution

Q1. What is a Virtual Constructor? What are its advantages? Explain with examples. **(30 Marks)**

Q2. What is a Virtual Function? How does it differ from a Pure Virtual Function? Explain with examples. **(30 Marks)**

Q3. What is a Header File? Explain any 5 header files along with their functions. **(20 Marks)**

Q1. **What is a Virtual Constructor? What are its advantages? Explain with examples.**

Introduction: In C++, the concept of a **virtual constructor** does not exist directly, because constructors in C++ cannot be virtual. The reason for this is that constructors are responsible for creating objects, and until the object is created, there is no way to determine which virtual function would be called. However, a "virtual constructor" typically refers to design patterns or mechanisms that provide runtime-based object creation and achieve similar results. This is most commonly implemented using **factory design patterns**.

The goal of simulating a virtual constructor is to provide a way to instantiate objects at runtime based on input or conditions without hardcoding the object type.

Why C++ Doesn't Have Virtual Constructors: Constructors are used to initialize objects, and they are called during object creation. Virtual functions, on the other hand, rely on the existence of an object to resolve which version of the function to call. Since the object is not fully constructed when the constructor is called, it cannot use virtual function tables for polymorphism.

Advantages of Simulating Virtual Constructors:

1. **Dynamic Object Creation:** Virtual constructor-like behavior can enable dynamic selection and creation of different types of objects at runtime, based on input.
2. **Polymorphism:** Allows creation of objects via base class pointers or references, leading to flexible polymorphic behavior.
3. **Loose Coupling:** The client code is not tightly bound to specific object classes; it can create objects based on external information or configuration, promoting flexibility and scalability.
4. **Extendibility:** New derived classes can be added without modifying the factory or base class logic, as long as the same interface is adhered to.

Example: In this example, we use the **Factory Design Pattern** to simulate virtual constructor behavior. We create objects based on user input without the need to modify the core logic when new object types are added.

```cpp
#include <iostream>
#include <memory>

// Base class
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
    virtual ~Shape() {}
};

// Derived class 1
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Circle" << std::endl;
    }
};

// Derived class 2
class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing Square" << std::endl;
    }
};

// Factory method to simulate virtual constructor
std::shared_ptr<Shape> createShape(int shapeType) {
    if (shapeType == 1)
        return std::make_shared<Circle>();
    else if (shapeType == 2)
        return std::make_shared<Square>();
    return nullptr;
}
```

```
int main() {
    std::shared_ptr<Shape> shape1 = createShape(1);
    shape1->draw();  // Output: Drawing Circle

    std::shared_ptr<Shape> shape2 = createShape(2);
    shape2->draw();  // Output: Drawing Square

    return 0;
}
```

Explanation:

- The `createShape` function serves as a **factory method**, deciding which object (Circle or Square) to instantiate based on the runtime input.
- The base class `Shape` has a pure virtual function `draw()`, which is overridden in the derived classes.
- This approach enables creating new `Shape` objects dynamically, simulating the effect of a "virtual constructor" by selecting the object type at runtime.

Conclusion: While C++ does not support virtual constructors natively, design patterns such as the Factory pattern can effectively simulate this behavior. This allows for dynamic and flexible object creation, which is a key component of runtime polymorphism and extendibility in object-oriented design.

Q2. What is a Virtual Function? How does it differ from a Pure Virtual Function? Explain with examples.

Introduction: A **virtual function** is a member function in a base class that can be overridden in derived classes to provide specific functionality. The base class uses the `virtual` keyword, allowing derived classes to modify the behavior of the base class function at runtime, achieving **runtime polymorphism**. When a function is declared as `virtual`, the decision about which function (base or derived) will be called is made during runtime based on the object type.

A **pure virtual function**, on the other hand, is a virtual function that has no implementation in the base class. It must be overridden in any derived class, otherwise, the derived class becomes abstract and cannot be instantiated.

Virtual Function:

- A **virtual function** is declared using the `virtual` keyword in the base class.
- It can have a default implementation in the base class, which derived classes may override if necessary.
- It is optional for derived classes to provide their own version of a virtual function.
- The compiler uses **late binding** or **dynamic dispatch** to decide which version of the function to invoke at runtime.

Pure Virtual Function:

- A **pure virtual function** is a function that is declared in the base class but has no implementation in that class.
- It is specified by assigning `= 0` in the declaration of the function.
- All derived classes must implement the pure virtual function, otherwise, they themselves become abstract classes and cannot be instantiated.
- Pure virtual functions make a class abstract and force derived classes to provide their specific implementation.

Example of Virtual Function:

```cpp
#include <iostream>

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

class Derived : public Base {
public:
    void show() override {  // Overriding the base class function
        std::cout << "Derived class show function" << std::endl;
    }
};
```

```
int main() {
    Base *bptr;
    Derived d;
    bptr = &d;

    // Virtual function, resolved at runtime
    bptr->show();  // Output: Derived class show function

    return 0;
}
```

Explanation:

- In this example, the `show()` function is declared as virtual in the base class.
- Even though `bptr` is a pointer to the base class `Base`, at runtime it invokes the `show()` function from the derived class `Derived`, because the base class function is virtual.

Example of Pure Virtual Function:

```cpp
#include <iostream>

// Abstract base class
class AbstractBase {
public:
    // Pure virtual function
    virtual void show() = 0;
};

class Derived : public AbstractBase {
public:
    void show() override {
        std::cout << "Derived class implementation of pure virtual function"
    }
};

int main() {
    Derived d;
    d.show();  // Output: Derived class implementation of pure virtual functi

    return 0;
}
```

Explanation:

- The base class `AbstractBase` contains a pure virtual function `show()`, which means that the derived class `Derived` must provide an implementation.
- The base class cannot be instantiated directly because it contains a pure virtual function, making it abstract.

Key Differences:

1. **Implementation:**
    - Virtual Function: May have an implementation in the base class.
    - Pure Virtual Function: Has no implementation in the base class; must be implemented by derived classes.
2. **Instantiation:**
    - Virtual Function: The base class can still be instantiated if the virtual function has an implementation.
    - Pure Virtual Function: The base class cannot be instantiated; it becomes an abstract class.
3. **Purpose:**
    - Virtual Function: Provides flexibility, allowing derived classes to override functionality if necessary.
    - Pure Virtual Function: Forces derived classes to provide specific functionality, ensuring a common interface.

Conclusion: Virtual functions provide runtime polymorphism, allowing functions to be overridden by derived classes. Pure virtual functions, on the other hand, enforce the contract that derived classes must implement certain functionalities, making them essential for defining abstract classes and interfaces in object-oriented programming.

Q3. What is a Header File? Explain any 5 header files along with their functions.

Introduction: A **header file** in C++ is a file that contains declarations of functions, macros, classes, and constants that can be shared across multiple source files. It allows code reusability and modularity by separating interface from implementation. Header files usually have the `.h` extension and are included in C++ source files using the `#include` preprocessor directive.

Using header files is an essential part of organizing and maintaining larger C++ projects, as they allow you to declare interfaces in one place and use them across many different parts of your program.

Common Header Files in C++:

1. **iostream:**
   - **Purpose:** Provides functionality for input and output operations.
   - **Classes/Functions:** Contains `istream`, `ostream`, `cin`, `cout`, `cerr`, and `clog` for handling standard input/output streams.
   - **Usage Example:**

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

2. **cmath:**
   - **Purpose:** Contains mathematical functions like trigonometric, exponential, logarithmic, and power functions.
   - **Functions:** Includes functions such as `sin()`, `cos()`, `sqrt()`, and `pow()`.
   - **Usage Example:**

```cpp
#include <cmath>
int main() {
    double result = sqrt(16.0);  // result = 4.0
    return 0;
}
```

3. cstdlib:
   - **Purpose:** Provides utility functions for memory allocation, random number generation, program control, and more.
   - **Functions:** Includes functions such as `malloc()`, `free()`, `rand()`, `srand()`, `exit()`, and `abort()`.
   - **Usage Example:**

```cpp
#include <cstdlib>
int main() {
    int randomNumber = rand();  // Generates a random number
    return 0;
}
```

4. string:
   - **Purpose:** Provides the `std::string` class for handling and manipulating sequences of characters in a convenient way.
   - **Classes/Functions:** Includes the `std::string` class and functions like `length()`, `substr()`, and `append()`.
   - **Usage Example:**

```cpp
#include <string>
int main() {
    std::string str = "Hello";
    str.append(" World");
    std::cout << str << std::endl;  // Output: Hello World
    return 0;
}
```

5. **vector:**
   - **Purpose:** Part of the Standard Template Library (STL), `vector` is a dynamic array that can grow or shrink in size.
   - **Classes/Functions:** Provides the `std::vector` class and functions like `push_back()`, `size()`, and `at()`.
   - **Usage Example:**

```cpp
#include <vector>
int main() {
    std::vector<int> vec = {1, 2, 3};
    vec.push_back(4);  // Adds 4 to the vector
    std::cout << vec.at(0) << std::endl;  // Output: 1
    return 0;
}
```

Conclusion: Header files play a crucial role in C++ by promoting code reusability and modularity. By separating declarations and definitions, header files allow efficient management of larger projects and make it easier to maintain and scale programs. Understanding and using common header files like `iostream`, `cmath`, `cstdlib`, `string`, and `vector` is fundamental for any C++ developer.