
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	:	31stOctober,2024(for July session) 30thApril,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.

Ans

Virtual Constructor: Concept and Explanation

A **Virtual Constructor** is a concept or design pattern rather than an actual feature in programming languages like C++. In object-oriented programming, a virtual constructor is used to create objects of derived classes from base class pointers, dynamically at runtime, similar to how virtual functions work for methods.

Understanding the Concept:

In C++, constructors cannot be virtual because when a constructor is called, the object is not yet fully created, so there is no object on which a virtual function call can operate. However, there are scenarios where you want to create objects of different derived classes depending on some condition at runtime. This is where the "virtual constructor" concept comes in. It is implemented using the **Factory Method** or **Prototype Design Pattern**.

Advantages:

1. **Runtime Flexibility:** The virtual constructor allows creating objects of derived classes at runtime based on dynamic conditions.
2. **Encapsulation:** The details of object creation are hidden, providing a cleaner interface.
3. **Extensibility:** New derived classes can be added with minimal changes to existing code, improving the maintainability and scalability of the system.
4. **Polymorphism:** It supports polymorphism by allowing objects of different derived classes to be treated uniformly.

Example Using the Factory Method:

Here's an example to illustrate the virtual constructor pattern using the Factory Method:

Code in C++

```
#include <iostream>

using namespace std;

// Base class
class Shape {
public:
```

```
// Pure virtual function
virtual void draw() = 0;

// Static function acting as a 'virtual constructor'
static Shape* createShape(int type);
};

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

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

// Factory Method Implementation
Shape* Shape::createShape(int type) {
    if (type == 1) {
        return new Circle();
    } else if (type == 2) {
```

```

        return new Square();
    } else {
        return nullptr;
    }
}

int main() {
    // Creating objects using the 'virtual constructor'

    Shape* shape1 = Shape::createShape(1); // Creates Circle
    Shape* shape2 = Shape::createShape(2); // Creates Square

    // Using polymorphism
    if (shape1) shape1->draw();
    if (shape2) shape2->draw();

    // Clean up
    delete shape1;
    delete shape2;

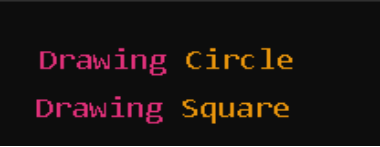
    return 0;
}

```

Explanation:

- **Shape Class:** The base class Shape declares a pure virtual function draw() and a static method createShape() which acts as a virtual constructor.
- **Circle and Square Classes:** These are derived classes implementing the draw() method.
- **Factory Method:** The createShape() method is responsible for deciding which derived class to instantiate based on the input parameter.

Output:



Drawing Circle
Drawing Square

In this example, the 'Shape::createShape()' method is responsible for creating objects of Circle or Square based on the input provided. This demonstrates the concept of a virtual constructor by allowing object creation dynamically at runtime.

Conclusion:

While C++ doesn't support virtual constructors directly, the concept is effectively implemented using design patterns like the Factory Method or Prototype Pattern. These patterns provide flexibility and maintainability by enabling dynamic object creation, which is essential for polymorphism in object-oriented programming.

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

Virtual Function in C++

A **Virtual Function** is a function in a base class that is declared with the virtual keyword. The purpose of a virtual function is to ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call. This is essential in achieving **polymorphism** in C++.

When a derived class overrides a virtual function, and you call that function using a pointer or reference to the base class, the function in the derived class is called. This allows for dynamic binding or late binding.

Key Points:

- **Declared using the virtual keyword** in the base class.
- **Can be overridden** by the derived class.
- **Supports runtime polymorphism**, ensuring the correct function is called based on the actual object type.

Example of a Virtual Function:

Code C++

```
#include <iostream>
```

```
using namespace std;
```

```

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

class Derived : public Base {
public:
    void show() override { // Overriding the virtual function
        cout << "Derived class show function called." << endl;
    }
};

int main() {
    Base* b;    // Base class pointer
    Derived d;  // Derived class object
    b = &d;

    b->show();  // Calls Derived class function due to virtual function mechanism

    return 0;
}

```

Output:

Derived class show function called.

Pure Virtual Function in C++

A **Pure Virtual Function** is a virtual function that has no implementation in the base class and is meant to be overridden in derived classes. It is declared by assigning '0' in

the base class. A class containing at least one pure virtual function is considered an **abstract class** and cannot be instantiated.

Key Points:

- **Declared by assigning = 0** in the base class.
- **Must be overridden** by derived classes; otherwise, the derived class will also become abstract.
- **Makes the base class abstract**, meaning it cannot be instantiated.

Example of a Pure Virtual Function:

Code c++

```
#include <iostream>
```

```
using namespace std;
```

```
class Shape {
```

```
public:
```

```
    virtual void draw() = 0; // Pure virtual function
```

```
};
```

```
class Circle : public Shape {
```

```
public:
```

```
    void draw() override {
```

```
        cout << "Drawing Circle" << endl;
```

```
    }
```

```
};
```

```
class Square : public Shape {
```

```
public:
```

```
    void draw() override {
```

```
        cout << "Drawing Square" << endl;
```

```
    }
```

```
};
```

```
int main() {  
    Shape* s1 = new Circle();  
    Shape* s2 = new Square();  
  
    s1->draw(); // Calls Circle's draw function  
    s2->draw(); // Calls Square's draw function  
  
    delete s1;  
    delete s2;  
  
    return 0;  
}
```

Output:

Drawing Circle

Drawing Square

Differences Between Virtual and Pure Virtual Functions:

Feature	Virtual Function	Pure Virtual Function
Implementation	May have an implementation in the base class.	No implementation in the base class (declared = 0).
Overriding in Derived Class	Can be overridden, but it's optional.	Must be overridden in derived classes.
Class Instantiation	The base class can be instantiated if no other pure virtual functions exist.	The base class cannot be instantiated; it's abstract.

Feature	Virtual Function	Pure Virtual Function
Use Case	Provides a default implementation while allowing overrides.	Enforces derived classes to provide specific implementations.

Summary:

- **Virtual Functions** are used to achieve runtime polymorphism, allowing derived classes to override base class methods.
- **Pure Virtual Functions** make a class abstract, enforcing derived classes to provide their own implementations, ensuring specific behavior for derived class objects.

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

Ans

What is a Header File?

A **Header File** in C and C++ is a file with a '.h' extension (though in C++ they can also have '.hpp' or no extension in some cases) that contains declarations of functions, macros, variables, and other identifiers. These files are included in your source code files using the '#include' preprocessor directive to provide the necessary information for compilation.

Header files allow for the separation of interface and implementation. They enable code reusability and modularity by allowing declarations to be shared across multiple source files, ensuring that the same declarations are available wherever they are needed without duplication.

Commonly Used Header Files in C++:

1. <iostream>

- **Purpose:** Provides input and output stream objects and functions.
- **Common Components:**
 - `std::cin`: Standard input stream (for reading input from the user).
 - `std::cout`: Standard output stream (for printing output to the console).
 - `std::cerr`: Standard error stream (for printing error messages).
 - `std::endl`: Used to insert a newline and flush the output buffer.
- **Example:**

Code C++

```
#include <iostream>
```

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

2. <vector>

- **Purpose:** Provides the `std::vector` container class, which is a dynamic array that can resize automatically.
- **Common Components:**
 - `std::vector<T>`: Template class for creating a dynamic array of type `T`.
 - Methods like `push_back()`, `size()`, `at()`, etc., are used for manipulating the vector.
- **Example:**

Code C++

```
#include <iostream>
```

```
#include <vector>
```

```
int main() {  
    std::vector<int> numbers = {1, 2, 3, 4, 5};  
    numbers.push_back(6);  
  
    for(int num : numbers) {  
        std::cout << num << " ";  
    }  
  
    return 0;
```

}

3. <string>

- **Purpose:** Provides the std::string class, which is used for handling and manipulating sequences of characters.
- **Common Components:**
 - std::string: Class for representing and managing a string of characters.
 - Methods like length(), substr(), find(), append(), etc., are used for string manipulation.
- **Example:**

Code C++

```
#include <iostream>
```

```
#include <string>
```

```
int main() {
```

```
    std::string greeting = "Hello";
```

```
    greeting.append(", World!");
```

```
    std::cout << greeting << std::endl;
```

```
    return 0;
```

```
}
```

4. <cmath>

- **Purpose:** Provides a range of mathematical functions and operations.
- **Common Components:**
 - Functions like sqrt(), pow(), sin(), cos(), log(), etc., are used for performing mathematical computations.
- **Example:**

Code C++

```
#include <iostream>
```

```
#include <cmath>
```

```
int main() {
```

```
    double result = std::sqrt(25.0);
```

```
    std::cout << "The square root of 25 is: " << result << std::endl;
```

```
    return 0;
```

```
}
```

5. <algorithm>

- **Purpose:** Provides a range of functions for performing operations on sequences of elements, such as searching, sorting, and manipulating data.
- **Common Components:**
 - Functions like `sort()`, `find()`, `reverse()`, `max_element()`, `min_element()`, etc., are used for performing operations on containers like vectors and arrays.
- **Example:**

Code C++

```
#include <iostream>
```

```
#include <algorithm>
```

```
#include <vector>
```

```
int main() {
```

```
    std::vector<int> numbers = {3, 1, 4, 1, 5, 9};
```

```
    std::sort(numbers.begin(), numbers.end());
```

```
    for(int num : numbers) {
```

```
        std::cout << num << " ";
```

```
    }
```

```
    return 0;  
}
```

Summary:

Header files are essential components in C and C++ programming, facilitating code organization, modularity, and reusability. Common header files like `<iostream>`, `<vector>`, `<string>`, `<cmath>`, and `<algorithm>` provide a wide range of functionalities that are frequently used in programming tasks, making them indispensable in most C++ programs.