IKGPTU Question Paper Solution
Date: 29-12-2023
Session: July-Dec 2023

Section-A

1. **Explain the briefly:**

a) Float

Float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Syntax of float
The **float keyword** is used to declare the variable as a floating point:

```
float var_name;
```

b) Operator

An operator can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

c) Inline Function

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Syntax:

```
inline return-type function-name(parameters)
{
  // function code
}
```

d) Object

An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the

objects interact by sending messages to one another. Each object contains data and code to manipulate the data.

e) Public

The public keyword is an access specifier. Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are public - which means that they can be accessed and modified from outside the code.

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

f) Destructor

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

Syntax to Define Destructor
The syntax for defining the destructor within the class:

```
~ <class-name>() {
  // some instructions
}
```

g) Inheritance

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object Oriented Programming in C++. In this article, we will learn about inheritance in C++, its modes and types along with the information about how it affects different properties of the class.

Syntax of Inheritance in C++

```
class  derived_class_name : access-specifier  base_class_name
{
    //    body ....
};
```

where,

- **class:** keyword to create a new class
- **derived_class_name**: name of the new class, which will inherit the base class

- **access-specifier**: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.
- **base-class-name**: name of the base class.

h) Catch

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.
Syntax of try-catch in C++

```
try {
   // Code that might throw an exception
  throw SomeExceptionType("Error message");
}
catch( ExceptionName e1 )  {
   // catch block catches the exception that is thrown from try block
}
```

i) Call by Value

In the call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.

- There are two copies of parameters stored in different memory locations.
- One is the original copy and the other is the function copy.
- Any changes made inside functions are not reflected in the actual parameters of the caller.

j) Friend Function

if a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
   friend data_type function_name(argument/s);          // syntax of friend function.
```

```
};
```

Section- B

2. Explain with the help of example how to overload '++' Operator?

n C++, operator overloading allows you to redefine the behavior of operators for user-defined types (such as classes or structs). The ++ operator, which is used to increment a value, can be overloaded in two ways:

1. **Pre-increment (++a)**
2. **Post-increment (a++)**

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


class Counter {
private:
   int value;


public:
   // Constructor to initialize the counter
   Counter(int v = 0) : value(v) {}


   // Pre-increment operator overloading
   Counter& operator++() {
      // Increment value first, then return *this (the current object)
      ++value;
      return *this;
   }


   // Post-increment operator overloading
   Counter operator++(int) {
      // Create a copy of the current object to return
      Counter temp = *this;
      value++;
      return temp;  // Return the original value before increment
```

```
    }

    // Method to print the counter value
    void display() const {
        cout << "Counter value: " << value << endl;
    }
};

int main() {
    Counter counter1(5);
    Counter counter2 = counter1++;

    cout << "After post-increment:" << endl;
    counter1.display();  // Should display 6
    counter2.display();  // Should display 5 (the original value)

    cout << "After pre-increment:" << endl;
    ++counter1;  // Pre-increment, counter1 becomes 7
    counter1.display();  // Should display 7

    return 0;
}
```

3. Differentiate between virtual and pure virtual function with help of an example.

Key Differences Between Virtual and Pure Virtual Functions:

Feature	Virtual Function	Pure Virtual Function
Declaration	Declared with the virtual keyword.	Declared with = 0 at the end of the function signature.
Implementation in Base	May have a default implementation.	Cannot have any implementation in the base class.

Abstract Class	Does not make the class abstract by itself.	Makes the class abstract if it has at least one pure virtual function.
Derived Class Requirement	Derived class may choose to override, but it's not required.	Derived class **must** override the pure virtual function, otherwise, it is also abstract.
Instantiability	Objects of the base class can be created (unless the class is abstract for other reasons).	The class containing a pure virtual function cannot be instantiated.
Purpose	Used to allow polymorphic behavior with a default or optional implementation.	Used to define an interface that must be implemented by derived classes.

```
#include <iostream>

using namespace std;

// Base class with a virtual function and a pure virtual function

class Shape {

public:

    // Virtual function with a default implementation

    virtual void draw() {

        cout << "Drawing a shape (base class version)." << endl;

    }

    // Pure virtual function (no implementation in base class)

    virtual double area() = 0;  // Pure virtual function, making Shape an abstract class

};

// Derived class 1: Circle
```

```
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    // Override the draw function for Circle
    void draw() override {
        cout << "Drawing a Circle." << endl;
    }
    // Implement the pure virtual function for area
    double area() override {
        return 3.14159 * radius * radius;
    }
};
// Derived class 2: Rectangle
class Rectangle : public Shape {
private:
    double length, width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
```

```
    // Override the draw function for Rectangle
    void draw() override {
        cout << "Drawing a Rectangle." << endl;
    }
    // Implement the pure virtual function for area
    double area() override {
        return length * width;
    }
};
int main() {
    // Create objects of derived classes
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    // Create pointers of the base class type
    Shape* shape1 = &circle;
    Shape* shape2 = &rectangle;
    // Demonstrate polymorphism using the virtual function
    cout << "Shape 1: ";
    shape1->draw();  // Calls Circle's draw function
    cout << "Shape 2: ";
```

```
    shape2->draw();  // Calls Rectangle's draw function

    // Demonstrate polymorphism using the pure virtual function (area)

    cout << "Area of Shape 1 (Circle): " << shape1->area() << endl;

    cout << "Area of Shape 2 (Rectangle): " << shape2->area() << endl;

    return 0;

}
```

4. Write a program to read a file and copy character by character into another file.

```
#include <iostream>

#include <fstream>

using namespace std;

int main() {

    // Declare input and output file streams

    ifstream inputFile;

    ofstream outputFile;

    // Open the input file in read mode

    inputFile.open("input.txt");

    // Check if the input file is opened successfully

    if (!inputFile) {

        cerr << "Error opening the input file!" << endl;

        return 1; // Exit with error code
```

```
    }
    // Open the output file in write mode
    outputFile.open("output.txt");
    // Check if the output file is opened successfully
    if (!outputFile) {
        cerr << "Error opening the output file!" << endl;
        return 1; // Exit with error code
    }
    // Variable to store each character
    char ch;
    // Read character by character from the input file and write to the output file
    while (inputFile.get(ch)) {
        outputFile.put(ch);
    }
    cout << "File copied successfully!" << endl;
    // Close both input and output files
    inputFile.close();
    outputFile.close();
    return 0;
}
```

5.Discuss the use of exceptional handling in programming.
Exception Handling in Programming

Exception handling is a mechanism used in programming languages to deal with runtime errors or exceptional situations in a controlled manner. It allows developers to manage errors or unexpected events that might otherwise cause a program to crash or behave unpredictably.

In languages like C++, Java, and Python, exceptions are used to handle errors, and they provide a structured way of responding to errors while maintaining the normal flow of execution.

Key Concepts of Exception Handling:

1. Exception:
    - An exception is an event that disrupts the normal flow of the program's execution. It is typically an error or unexpected condition, such as division by zero, file not found, memory allocation failure, etc.
2. Try Block:
    - The try block contains code that might throw an exception. It is a way of marking code that needs to be checked for errors.
3. Catch Block:
    - The catch block is used to handle the exception. If an exception is thrown within the try block, the control is transferred to the corresponding catch block, where the exception is processed.
4. Throw Statement:
    - The throw keyword is used to explicitly throw an exception. It can be used to throw objects of any type, but typically exceptions are derived from the base exception class (in C++ or other languages).

Why Use Exception Handling?

1. Graceful Error Handling:
    - Without exception handling, errors often cause a program to terminate abruptly. Exception handling allows the program to recover from errors and continue execution or at least terminate gracefully without a crash.
2. Separation of Error-Handling Logic:
    - Exception handling provides a clear separation of the main logic of a program and the error-handling code. This leads to cleaner, more readable code where error-handling does not clutter the regular flow of execution.
3. Error Propagation:
    - Exceptions allow errors to propagate up the call stack, so they can be handled at a higher level in the program. This means that lower-level functions do not have to manage every possible error, and errors can be handled at an appropriate level.
4. Centralized Error Management:
    - By using exception handling, errors can be handled in a centralized manner, improving maintainability. You can define global error-handling routines (e.g., logging or resource cleanup) in one place.
5. Avoiding Unpredictable Behavior:

- Uncaught exceptions can lead to unpredictable behavior. Exception handling ensures that errors are handled explicitly, preventing the program from running into undefined states.

6. Explain the concept of access specifiers with an example.
Access Specifiers in C++

In C++, **access specifiers** are keywords that define the visibility of class members (variables and functions) to other parts of the program. They control the **access level** to the members of a class. C++ provides three primary access specifiers:

1. **public**: Members declared as public can be accessed from anywhere in the program.
2. **private**: Members declared as private can only be accessed by functions within the same class. They are not accessible outside the class.
3. **protected**: Members declared as protected can be accessed by member functions of the class itself and derived classes. However, they are not accessible outside the class.

Access Specifier Usage:

- **public**: Useful for members that need to be accessible from outside the class. Typically used for functions that provide an interface to the user of the class.
- **private**: Useful for members that should not be accessed directly from outside the class, typically to protect data integrity. These are usually data members or helper functions.
- **protected**: Useful for members that should only be accessible within the class and its derived classes. This is typically used in inheritance scenarios where derived classes need access to certain data but should not expose it to other parts of the program.

Section- C

7. Explain different types of inheritance and the concept of ambiguity.
Types of Inheritance in C++

In C++, inheritance is a mechanism that allows a class (called the derived class) to inherit properties and behaviors (i.e., data members and member functions) from another class (called the base class). This promotes code reusability and creates a hierarchy between classes.

There are several types of inheritance in C++:

1. Single Inheritance

In **single inheritance**, a class is derived from only one base class. This is the simplest form of inheritance where a derived class inherits from a single base class.

2. Multiple Inheritance

In **multiple inheritance**, a class is derived from more than one base class. The derived class inherits members from multiple base classes.

3. Multilevel Inheritance

In **multilevel inheritance**, a class is derived from another derived class, creating a chain of inheritance.

4. Hierarchical Inheritance

In **hierarchical inheritance**, multiple classes inherit from a single base class. In this scenario, one base class is shared by multiple derived classes.

5. Hybrid Inheritance

In **hybrid inheritance**, a class is derived from more than one class, and the inheritance forms can be a combination of multiple inheritance, multilevel inheritance, or hierarchical inheritance.

Example:

```
#include <iostream>

using namespace std;

class Animal {

public:

   void speak() {

      cout << "Animal speaks" << endl;

   }

};

class Mammal {

public:

   void walk() {

      cout << "Mammal walks" << endl;

   }

};

class Dog : public Animal, public Mammal {  // Dog inherits from both Animal and Mammal
```

```
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
int main() {
    Dog dog;
    dog.speak(); // Inherited from Animal
    dog.walk();  // Inherited from Mammal
    dog.bark();  // Defined in Dog
    return 0;
}
```

Ambiguity in Inheritance

Ambiguity arises in inheritance when a derived class inherits multiple base classes, and those base classes have methods or members with the same name. This causes confusion about which method or member should be called, especially if the derived class does not provide a clear indication.

Example of Ambiguity:

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

class A {
public:
    void display() {
        cout << "Display of class A" << endl;
```

```
    }
};
class B {
public:
    void display() {
        cout << "Display of class B" << endl;
    }
};
class C : public A, public B {
public:
    // display() function is ambiguous because both A and B have a display function.
};
int main() {
    C obj;
    obj.display();  // Error: Ambiguity between A::display() and B::display()
    return 0;
}
```

8. Differentiate between functions and recursive functions with help of an example.
Difference Between Functions and Recursive Functions

In programming, functions and recursive functions are both mechanisms for performing specific tasks, but they differ in how they operate and solve problems.

1. Function

A function is a block of code that is defined to perform a specific task. It typically takes some input (arguments), processes it, and returns a result. Functions can be called multiple times from different parts of a program.

- Purpose: To perform a single task or a set of tasks that can be reused.

- How it works: A function is executed when it is called. It runs once and completes its task.

Example of a Simple Function:

```
#include <iostream>

using namespace std;

// Function that calculates the sum of two numbers

int add(int a, int b) {

   return a + b;

}

int main() {

   int result = add(5, 3);  // Call the function

   cout << "The sum is: " << result << endl;

   return 0;

}
```

2. Recursive Function

A recursive function is a function that calls itself in order to solve a problem. The key idea is to break a larger problem into smaller, similar subproblems. Recursive functions typically have two key components:

- Base case: The condition that stops the recursion.
- Recursive case: The part of the function where it calls itself with modified parameters.
- Purpose: To solve problems that can be broken down into smaller, similar problems (e.g., calculating factorials, Fibonacci numbers, tree traversals, etc.).
- How it works: A recursive function keeps calling itself with new arguments until it reaches the base case, at which point it stops and begins returning results back up the recursive calls.

Example of a Recursive Function (Factorial):

```
#include <iostream>

using namespace std;
```

```
// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1)  // Base case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive case
}
int main() {
    int result = factorial(5);  // Call the recursive function
    cout << "Factorial of 5 is: " << result << endl;
    return 0;
}
```

Key Differences Between Functions and Recursive Functions

Aspect	Function	Recursive Function
Definition	A function is a block of code that performs a specific task.	A recursive function is a function that calls itself to solve smaller instances of the problem.
Execution	A function executes once when called and performs the task.	A recursive function calls itself repeatedly until a base case is reached.
Base Case	A regular function does not have a base case because it does not call itself.	A recursive function must have a base case to prevent infinite recursion.

Use Case	Used for simple, non-repetitive tasks like addition, subtraction, etc.	Used for problems that can be broken down into smaller subproblems (e.g., calculating factorials, Fibonacci series, traversing trees).
Efficiency	Generally more efficient for simple tasks, since it does not involve function calls within itself.	Recursive functions can be less efficient and lead to stack overflow if not designed properly, especially for deep recursion.
Memory Usage	Standard function calls use stack space for the function call itself.	Recursive function calls accumulate on the stack for each recursive call, potentially leading to high memory usage for deep recursion.

9. Write a detailed note on exceptional handling mechanism.

Exception Handling in C++

Exception handling is a powerful mechanism in C++ (and many other programming languages) that allows a program to deal with unexpected situations (called exceptions) gracefully, without abruptly terminating the program. It provides a way to detect errors, recover from them, and maintain the program's stability and functionality.

In C++, the **exception handling mechanism** is implemented using three main keywords:

- **try**
- **throw**
- **catch**

These keywords provide a structured way to deal with runtime errors (exceptions) in a controlled manner.

Key Concepts of Exception Handling

1. **Exception**: An event that disrupts the normal flow of program execution. It can be a runtime error (e.g., division by zero, invalid input, out-of-memory error), or any other unexpected condition that the program needs to handle gracefully.
2. **try Block**: The code that might generate an exception is placed inside the try block. If an exception occurs within the try block, the control is transferred to the corresponding catch block.

3. **throw Keyword**: When an exception is detected, the throw keyword is used to signal the occurrence of an error. It "throws" the exception to the runtime system, which will search for an appropriate catch block to handle it.
4. **catch Block**: The catch block is responsible for catching the thrown exception. It handles the exception and allows the program to continue executing instead of crashing.

Exception Handling Syntax

```
try {
    // Code that might throw an exception
    // If an exception occurs here, control will be passed to the catch block
}
catch (ExceptionType1 e1) {
    // Handle exception of type ExceptionType1
}
catch (ExceptionType2 e2) {
    // Handle exception of type ExceptionType2
}
catch (...) {
    // Catch-all handler for any other types of exceptions
    // The '...' can catch any type of exception if it doesn't match any specific ones
}
```

Steps in Exception Handling:

1. **Code Execution in the try Block**:
    - Code that might raise an exception is enclosed within the try block.
    - If no exception is thrown, the program proceeds as normal.
2. **Throwing an Exception (throw)**:
    - If an error occurs within the try block, an exception is thrown using the throw keyword.
    - The type of exception thrown can be any object (usually of type std::exception or user-defined types).
3. **Catching the Exception (catch)**:

    - After an exception is thrown, the runtime system looks for an appropriate catch block that can handle the thrown exception.
    - If a matching catch block is found, it handles the exception; if not, the program will terminate.
4. **After the Catch Block**:
    - After handling the exception in the catch block, the program continues its execution after the try-catch block, unless the exception is rethrown or there is a fatal error.

Types of Exceptions:

- **Standard Exceptions**:
    - C++ provides a built-in class std::exception, which is the base class for all standard exceptions. Other standard exceptions, like std::logic_error, std::runtime_error, etc., are derived from this base class.
- **User-defined Exceptions**:
    - You can also define your own exceptions by creating classes that derive from std::exception. This is useful when you want to handle specific error conditions in your application.

Example :

```
#include <iostream>

#include <stdexcept>  // For std::runtime_error

using namespace std;

int divide(int a, int b) {

   if (b == 0) {

      throw runtime_error("Division by zero error!");

   }

   return a / b;

}

int main() {

   try {

      int result = divide(10, 0);  // This will throw an exception

      cout << "Result: " << result << endl;
```

```
    }
    catch (const runtime_error &e) {
        // Catch the exception and print the error message
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```