black N White

NAME	
ROLL NUMBER	
SEMESTER	2nd
COURSE CODE	DCA1210
COURSE NAME	BCA
Subject Name	OBJECT-ORIENTED PROGRAMMING USING C ++

SET - I

Q.1) Describe the main differences between procedural programming in C and object-oriented programming in C++.

Answer .:-

Procedural programming in C and object-oriented programming (OOP) in C++ are two different approaches to writing and organizing computer programs. Although C++ is based on C, it adds several powerful features that support object-oriented programming. Below are the key differences between these two paradigms:

1. Programming Paradigm

- **C** follows a **procedural programming** approach, where the focus is on writing procedures or functions that operate on data.
- **C++** supports **object-oriented programming**, which is centered around **objects** — entities that combine data and behavior.

2. Code Organization

- In **C**, the code is divided into **functions**. These functions take input, process it, and return output.
- In **C++**, the code is organized into **classes** and **objects**. Classes define data and the functions (methods) that operate on that data.

3. Data Handling

- **C** uses **structured data types** like arrays, structures, and pointers to handle data, and data is typically exposed globally.
- **C++** promotes **data encapsulation**, which means internal data is hidden within objects and accessed through public methods.

4. Encapsulation

- **C** has no direct support for encapsulation. Data and functions are generally separate.
- **C++** provides encapsulation through **access specifiers** like private, public, and protected, controlling how data and functions are accessed.

5. Inheritance

- **C** does not support inheritance. Code reuse is managed manually by copying or rewriting code.
- **C++** supports **inheritance**, allowing new classes to inherit properties and behaviors from existing classes, improving code reuse and scalability.

6. Polymorphism

- **C** lacks built-in support for polymorphism. Function behavior is usually fixed at compile time.
- **C++** supports **polymorphism**, especially through function overloading and virtual functions, allowing multiple forms of functions or methods to exist.

7. Function Overloading

- **C** does not allow function overloading. Function names must be unique.
- **C++** allows **function and operator overloading**, enabling the same function name to be used for different types of arguments.

8. Abstraction

- **C** requires manual handling for abstraction using function pointers or separate interfaces.
- **C++** supports abstraction using **abstract classes** and **interfaces**, allowing complex systems to be represented more simply.

9. Code Reusability

- In **C**, reusability depends on writing modular functions.
- In **C++**, classes and inheritance make it easier to reuse and extend code.

10. Memory Management

- Both C and C++ use pointers and dynamic memory, but C++ offers advanced tools like constructors, destructors, and smart pointers for better memory management.

Q.2) Define an inline function and explain its advantages.

Answer .:-

An **inline function** in C++ is a special type of function where the compiler replaces the function call with the actual code of the function during **compilation**, rather than performing a traditional function call at **runtime**.
It is declared using the keyword inline before the function definition. For example:

```
inline int square(int x) {
   return x * x;
}
```

When the above function is called in a program, the compiler may replace the call to square(x) with x * x directly.

How Inline Functions Work:
Normally, when a function is called, the control jumps to the function's memory location, executes the code, and then returns back. This process consumes time due to the overhead of jumping back and forth.
An inline function avoids this jump by **inserting the function code directly** at the point of the call, reducing function call overhead.

Note: The inline keyword is a **request** to the compiler, not a command. The compiler may choose to ignore it, especially for large or complex functions.

Advantages of Inline Functions:

1. **Reduced Function Call Overhead:**
   - Since the function code is inserted directly at the call site, there is no need for a jump to a different memory location.
   - This speeds up execution, especially for small and frequently used functions.
2. **Improved Performance for Small Functions:**
   - When simple functions (like arithmetic operations) are used repeatedly, making them inline can result in faster execution.
3. **Better Code Readability:**
   - Developers can write clean, reusable functions without worrying about performance loss due to function calls.
4. **Helps in Optimization:**
   - Inline functions may help the compiler perform further optimizations like **constant folding** and **loop unrolling** during compilation.
5. **Avoids Multiple Definition Errors:**
   - When inline functions are defined in header files and included in multiple source files, it prevents "multiple definition" linker errors, because inline functions have **internal linkage** by default.

Limitations

- Large or complex functions may not be inlined by the compiler.

- Excessive use of inline functions can lead to **code bloat**, increasing the size of the executable.
- Recursion and functions containing loops or static variables are usually **not inlined**.

Inline functions are useful for reducing the overhead of function calls and improving performance for small, simple functions. While they offer several advantages, they should be used wisely to avoid increasing the size of the compiled code. Ultimately, the decision to inline is made by the compiler based on optimization strategies.

Q.3) Explain the concept of exception handling in C++ and its necessity. Discuss the roles of try, throw, and catch in the exception handling mechanism.

Answer .:-

Exception handling in C++ is a powerful mechanism used to detect and handle **runtime errors** in a structured and safe way. Instead of terminating the program abruptly when an error occurs, C++ allows us to **catch the error, handle it gracefully**, and continue the execution or safely exit.
An **exception** is an event that occurs during program execution that disrupts the normal flow of the program. It can be caused by issues like division by zero, invalid memory access, file not found, etc.

Necessity of Exception Handling

1. **Ensures Program Stability**: Without exception handling, a program may crash unexpectedly. Exceptions allow safe handling of errors without stopping the entire application.
2. **Separates Error-Handling Code**: It separates normal code from error-handling code, making programs cleaner and easier to maintain.
3. **Improves Readability**: By avoiding cluttered if-else checks, exception handling improves code readability.
4. **Allows Recovery from Errors**: In certain cases, a program can recover from the error and continue running.

Key Keywords in Exception Handling
C++ provides three main components for handling exceptions:
1. try block

- A try block contains code that might cause an exception.
- If an exception occurs within this block, it is **thrown** and caught by the corresponding catch block.

```
try {
  // Code that may throw an exception
}
```

2. throw statement

- The throw keyword is used to signal (or "throw") an exception.
- It passes the control to the nearest matching catch block.

```
throw exception_type; // e.g., throw 10; or throw "Error!";
```

3. catch block

- The catch block handles the exception thrown by the try block.
- It specifies the type of exception it can handle.

```
catch (int e) {
    cout << "Caught an exception: " << e << endl;
}
```

Example:

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

int main() {
    int a = 10, b = 0;

    try {
        if (b == 0)
            throw "Division by zero error!";
        cout << "Result: " << a / b << endl;
    }
    catch (const char* msg) {
        cout << "Exception caught: " << msg << endl;
    }

    return 0;
}
```

Output:

```
Exception caught: Division by zero error!
```

Multiple catch Blocks

C++ allows multiple catch blocks to handle different types of exceptions:

```
try {
    // Code
}
catch (int e) { /* handle int */ }
catch (char c) { /* handle char */ }
catch (...) { /* handle any type */ }
```

Exception handling is essential in C++ to make programs more robust, reliable, and user-friendly. The try, throw, and catch blocks work together to detect, report, and recover from errors during runtime. By handling exceptions properly, developers can prevent program crashes and manage errors in a structured manner.

SET - II

Q.4) Describe basic programming using streams in C++. Include the process of creating, connecting, and disconnecting streams, and provide a simple example program.

Answer .:-

Introduction to Streams in C++
In C++, **streams** are used to perform input and output operations. A **stream** is simply a flow of data: either from an input device (like a keyboard or file) **to the program**, or from the program **to an output device** (like the screen or a file).
C++ provides a set of **stream classes** in the <iostream> and <fstream> headers to handle various I/O operations.

Types of Streams in C++

1. **Input Stream** – Used to read data (cin, ifstream)
2. **Output Stream** – Used to write data (cout, ofstream)
3. **Input/Output Stream** – Used for both reading and writing (fstream)

Steps in Stream Programming
1. Creating a Stream Object
You declare a stream object of a suitable class depending on the operation:

- ifstream for reading from a file
- ofstream for writing to a file
- fstream for both reading and writing

2. Connecting the Stream to a Source/Destination
Use .open() to associate the stream with a file.
3. Performing Read/Write Operations
Use << or >> operators (or stream functions) to perform I/O.
4. Disconnecting the Stream
Use .close() to close the file and disconnect the stream.

Example Program: Writing and Reading a File
```
#include <iostream>
#include <fstream> // Required for file streams
using namespace std;

int main() {
   ofstream outFile;       // Step 1: Create output stream
   outFile.open("data.txt"); // Step 2: Connect to file

   if (!outFile) {
      cout << "Error creating file!" << endl;
      return 1;
   }

   // Step 3: Write data
   outFile << "Hello, this is a test file.\n";
   outFile << "C++ stream programming is simple!" << endl;
```

```
    outFile.close(); // Step 4: Disconnect output stream

    // Step 1: Create input stream
    ifstream inFile("data.txt"); // Step 2: Connect directly via constructor

    if (!inFile) {
        cout << "Error opening file!" << endl;
        return 1;
    }

    // Step 3: Read and display content
    string line;
    while (getline(inFile, line)) {
        cout << line << endl;
    }

    inFile.close(); // Step 4: Disconnect input stream

    return 0;
}
```

Explanation of Example:

- ofstream outFile; creates a stream for writing.
- outFile.open("data.txt"); connects it to a file named data.txt.
- Data is written using the << operator.
- The file is closed using outFile.close();.
- Then ifstream inFile("data.txt"); reads the same file.
- getline(inFile, line); reads line-by-line.
- Finally, inFile.close(); disconnects the stream.

Conclusion:
Stream-based programming in C++ simplifies input and output operations. It abstracts the source or destination of data (file, console, etc.) into stream objects. By creating, connecting, using, and disconnecting streams properly, programmers can easily manage file I/O and build efficient, organized programs.

Q.5)What are access specifiers in C++? Provide examples to demonstrate the use of each access specifier in a class.

Answer .:-

Access specifiers in C++ define the **scope (visibility)** of class members (i.e., variables and functions). They control **how and where** class members can be accessed from within or outside the class.
There are **three** main access specifiers in C++:

1. **public**
2. **private**
3. **protected**

1. public:

- Members declared under public can be accessed **from anywhere**, both inside and outside the class.
- Used for interfaces that are meant to be accessible to all.

```
class MyClass {
public:
   int a;  // public data member

   void show() {  // public function
      cout << "Value of a: " << a << endl;
   }
};
```

Usage:

```
MyClass obj;
obj.a = 10;
obj.show();
```

2. private:

- Members declared as private can only be accessed **within the class**.
- They are **not accessible** from outside the class directly.
- Used to implement **data hiding**.

```
class MyClass {
private:
   int secret;

public:
   void setSecret(int s) {
      secret = s;
   }

   void showSecret() {
      cout << "Secret: " << secret << endl;
   }
};
```

Usage:

```
MyClass obj;
obj.setSecret(123);
obj.showSecret();
// obj.secret = 10; // ❌ Not allowed (private access)
```

3. protected:

- Members declared protected are accessible **within the class** and by **derived classes** (through inheritance).
- Not accessible outside the class unless inherited.

```
class Base {
protected:
   int protectedValue;

public:
    void setValue(int v) {
        protectedValue = v;
    }
};

class Derived : public Base {
public:
    void showValue() {
        cout << "Protected Value: " << protectedValue << endl;
    }
};
```

Usage:

```
Derived obj;
obj.setValue(42);
obj.showValue();
// obj.protectedValue = 10; // ✗ Not allowed (protected access)
```

Summary Table:

Access Specifier	Access within class	Access in derived class	Access outside class
public	☑ Yes	☑ Yes	☑ Yes
private	☑ Yes	✗ No	✗ No
protected	☑ Yes	☑ Yes	✗ No

Conclusion:
Access specifiers in C++ help enforce **data encapsulation and abstraction**, which are core principles of object-oriented programming. By using private, protected, and public effectively, developers can build more secure and maintainable programs by controlling how class members are accessed.

Q.6) Explain the concept of operator overloading in C++.

Answer .:-

Definition:
Operator overloading in C++ allows you to **redefine the way operators work** for user-defined types (like classes and structures). It lets you use **standard operators** (+, -, *, =, <<, etc.) to perform custom operations on objects, just like you do with built-in types.
In simple words, **operator overloading** means giving new meaning to an operator when it is used with **class objects**.

Why Use Operator Overloading?

- Makes code more **intuitive and readable**
- Allows object-oriented behavior with common operators
- Enhances **reusability** of classes

Syntax:
Operator overloading is done using a **special function** called an *operator function*, declared using the operator keyword:
return_type operator symbol (arguments)

You can overload both **unary** and **binary** operators.

Example: Overloading the '+' Operator
Let’s say we want to add two complex numbers using +:

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

class Complex {
    float real, imag;

public:
    Complex(float r = 0, float i = 0) {
        real = r;
        imag = i;
    }

    // Overload '+' operator
    Complex operator + (Complex obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(2.5, 3.5), c2(1.5, 4.5), c3;

    c3 = c1 + c2;  // Using overloaded '+' operator

    cout << "Sum of complex numbers: ";
    c3.display();

    return 0;
}
```

Output:

```
Sum of complex numbers: 4 + 8i
```

Important Notes:

- Not all operators can be overloaded (e.g., ::, .*, ., sizeof).
- You **cannot change** the number of operands or operator precedence.
- Operator functions can be **member functions** or **friend functions**.

Commonly Overloaded Operators:

- Arithmetic: +, -, *, /

- Assignment: =
- Comparison: ==, !=, <, >
- Stream: << and >> (usually as friend functions)

Conclusion:
Operator overloading in C++ allows you to use operators with user-defined types, making the code cleaner and more natural. It helps achieve **polymorphism**, one of the core features of object-oriented programming.