Module-3(introduction to oops programming)

1. What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?

Ans->procedural vs. Object-Oriented Programming

Aspect	Procedural Pro	Object-Oriented Programming (OOP)
Approach	Step-by-step, to	Bottom-up, based on real-world entities
Focus	Functions / Proc	Objects and Classes
Data Hand	Data is global/sh secure	Data is encapsulated within objects
Modularity	Code is divided functions	Code is divided into objects/classes
Reusability	Limited, throug reuse	High, using inheritance and polymorphism
Security	Less secure (no control)	More secure (access modifiers: private, protected, public)
Maintainal	Harder for large	Easier due to modularity and abstraction
Real-world Mapping	Does not model entities directly	Models real-world entities naturally
Examples	C, Pascal, Fortra	Java, C++, Python, C#, Ruby

Key Concepts	Functions, control flow, variables		Class, Object, Inheritance, Encapsulation, Polymorphism	

2.List and explain the main advantages of OOP over POP ?

Ans-> Advantages of OOP over POP.

Advantag	**OOP (Object-Or Programming)**	**POP (Procedural-Oriented Programming)**

Modularit	Code is divided into classes/objects — highly modular.	Code is divided into functions — less modular.	
Reusabilit	Supports inheritance and polymorphism — high reusability.	Limited code reuse; often requires rewriting code.	
Data Secu	Uses encapsulation — data is protected via access modifiers.	Data is globally accessible — less secure.	
Maintena	Easier to maintain and update due to modular design.	Harder to maintain, especially in large projects.	
Real-worl Modeling	Objects represent real-world entities — intuitive design.	No direct mapping to real-world concepts.	
Scalability	Easy to scale with added classes and objects.	Scaling becomes difficult as code grows.	
Flexibility	Polymorphism allows dynamic and flexible behavior.	No built-in flexibility like polymorphism.	
Team Collabora	Teams can work on separate classes/modules independently.	Collaboration is harder due to shared global data/functions.	

3.Explain the steps involved in setting up a C++ development environment ?

Ans->step 1: Install C++ Compiler (MinGW)

- MinGW is a popular GCC (GNU Compiler Collection) version for Windows.

Steps:

1. Go to: https://www.mingw-w64.org/
2. Download the **MinGW-w64 installer**. You can also get it directly from: 3. https://sourceforge.net/projects/mingw-w64/ 4. Install it with the following settings:

- Version: `8.1.0` (or latest) Architecture: `x86_64`

- Threads: `posix`
- Exception: `seh`
- Build revision: latest
- Choose an installation folder like: `C:\mingw-w64\`

Step 2: Add MinGW to System PATH

- This lets you use `g++` command from anywhere in the terminal.

Steps:

- Go to **Start > Search > Environment Variables**.
- Click **Environment Variables**.
- Under **System variables**, select `Path` > click **Edit**.
- Click **New** and paste:
- C:\mingw-w64\bin
- (Make sure it's the correct path where MinGW is installed.)
- Click **OK** on all windows.

Step 3: Install Visual Studio Code (VS Code)

- Download from: https://code.visualstudio.com/
- Install it using default settings.

Step 4: Install C++ Extension in VS Code

- Open **VS Code**.
- Go to **Extensions (Ctrl+Shift+X)**. • Search for `C/C++` by Microsoft.
- Click **Install**.
- (Optional but useful) Install `Code Runner` for quick code execution.

Step 5: Write a Sample C++ Program

- Open VS Code.
- Create a folder for your project, open it in VS Code.
- Create a new file: `main.cpp`

- **Example:**

```
#include <iostream> using

namespace std;


int main() {      cout << "Hello,

World!" << endl;      return 0;

}
```

Step 6: Compile and Run the Program

- Option 1: Using VS Code Terminal • Open Terminal in VS Code (`Ctrl + ~`) Type:

- `g++ main.cpp -o main.exe`
- `main.exe`
- `main.exe` will run and show `Hello, World!` in the terminal.

5. What are the main input/output operations in C++? Provide examples.

Ans-> In **C++**, the **main input/output (I/O) operations** are handled using:

- `cin` → for **input** (from keyboard)
- `cout` → for **output** (to screen)
- `cerr` → for **error messages**
- `clog` → for **log messages**

❾ All of these are part of the **`iostream`** header.

1. `cin` – Standard Input

O Used to **read data from the user (keyboard)**.

-> Example:

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

int main() { int age; cout << "Enter
your age: "; cin >> age; // taking input
cout << "You entered: " << age << endl;
return 0;
}
```

2. `cout` –Standard Output

O Used to **display output** to the screen.

-> Example:

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

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

<< is the **insertion operator**.

3. `cerr` – Standard Error Output

- Used to **display error messages**. Unlike `cout`, it does **not get buffered**, meaning it shows up immediately.

->Example:

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

int main() {      cerr << "An error
occurred!" << endl;      return 0;
}
```

4. `clog` – Standard Log Output

- Used for **logging messages**. It **is buffered**, unlike `cerr`.

-> Example:

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

int main() {      clog << "Program
started..." << endl;      return 0;
}
```

1: What are the different data types available in C++? Explain with examples.

Ans>

➢ Variables, Data Types, and Operators

- In C++, data types are used to tell the compiler what kind of data a variable will store. They define size, range, and operations that can be performed on the data.

Types of Data Types in C++

C++ mainly has three categories of data types:

1. Primary (Basic / Fundamental) Data Types
2. Derived Data Types
3. User-Defined Data Types

1. Primary (Basic) Data Types

These are the fundamental building blocks.

Data Type Size (in bytes) Example Description

int 4 int a = 10; Stores integers (whole numbers). float 4 float pi = 3.14; Stores decimal numbers (single precision). double 8 double d = 123.456; Stores decimal numbers (double precision). char 1 char grade = 'A'; Stores a single character. bool 1 bool isOn = true; Stores true (1) or false (0).
void 0 void function(); Means "no value". Used in functions. wchar_t 2 or 4 wchar_t ch = L'Ω'; Stores wide characters (Unicode).

O Example:

```
#include <iostream>
using namespace std; int
main() {
   int age = 20;    float
pi = 3.14;    double
price = 99.99;    char
```

```
grade = 'A';     bool
isPassed = true;


   cout << "Age: " << age << endl;     cout
<< "Pi: " << pi << endl;     cout << "Price:
" << price << endl;     cout << "Grade: "
<< grade << endl;     cout << "Passed: "
<< isPassed << endl;     return 0;
}
```

2. Derived Data Types These are built from the basic types.

Array → Collection of same type of data.

```
int arr[5] = {1, 2, 3, 4, 5};
```

Pointer → Stores memory address of a variable.

```
int x = 10; int*
ptr = &x;
```

Reference → Alternative name for a variable. int

```
a = 5;

int &ref = a;
```

Function → Returns a specific data type.

```
int sum(int a, int b) { return a + b;
}
```

3. User-Defined Data Types

Programmer can create custom types.

Structure (struct) struct Student {

```
   int id;
   char name[50];
};
```

Class (class)

```
class Car {
public:    string
brand;    int
speed;
};
```

Enumeration (enum)

enum Week {Mon, Tue, Wed, Thu, Fri, Sat, Sun}; Typedef / using typedef unsigned int uint; uint age = 25;

2.Explain the difference between implicit and explicit type conversion in C++.

O Difference Between Implicit and Explicit Type Conversion in C++

	Explicit Type Conversion (Type Casting)

Feature	Implicit Type Conversion	Manually done by the programmer
		Type Casting
Definition	Automatical by the comp	Programmer
Also Called	Type Coerci	Uses casting syntax (e.g., `(int)`, `static_cast<int>()`)
Who Perfor It?	Compiler	`int x = (int)10.5;` or `int x = static_cast<int>(10.5);`
Syntax	No special s needed	Yes, but programmer is aware and responsible
Example	`int x =` → x become	Full control over how and when it happens
Risk of Data Loss?	Yes, but less controlled	
Control	Less control how convers happens	

3.What are the different types of operators in C++? Provide examples of each.

Ans->

- **Arithmetic Operators**

Used to perform basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
−	Subtraction	a − b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder	a % b

- **Relational (Comparison) Operators** Used to compare two values.

Operato	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b

Operato	Description	Example
<=	Less than or equal to	a <= b

Logical Operators

Used to combine multiple conditions (expressions).

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
||	Logical or	(a==0 || b>0)
!	Logical NOT	!(a > b)

Assignment Operators

Used to assign values to variables.

Operat	Description	Example
=	Assignment	a = 10
+=	Add and assign	a += 5 (same as a = a + 5)
-=	Subtract and as	a -= 5
*=	Multiply and as	a *= 2
/=	Divide and assig	a /= 2
%=	Modulus and as	a %= 3

Increment/Decrement Operators

Used to increase or decrease a value by 1.

Operator	Description	Example
`++`	Increment	`a++` or `++a`
`--`	Decrement	`a--` or `--a`

- **Bitwise Operators**

Used to perform bit-level operations.

Operator	Description	Example
`&`	AND	`a & b`
`	`	`	OR
`^`	XOR	`a ^ b`
`~`	NOT	`~a`
`<<`	Left shift	`a << 1`
`>>`	Right shift	`a >> 1`

- **Conditional (Ternary) Operator** A shorthand for `if-else`.

Operat	Description	Example
`?:`	Ternary Opera	`result = (a > b) ? a : b;`

- **Special Operators `sizeof`**: Returns the size of a data type.

```
sizeof(int); // returns 4 (usually)
```

`typeid`: Returns the type of a variable (used with RTTI).

```
typeid(a).name();
```

4.Explain the purpose and use of constants and literals in C++.

Constants in C++

Purpose:

- Once a value is assigned, it cannot be changed.
- It makes the code safe and readable.
- A fixed name is given for repeated values.

Use:

const int AGE = 18; // constant #define PI 3.14

// macro constant constexpr int SIZE = 100; //

compile-time constants.

Literals in C++

Purpose:

Fixed values that are written directly in the program. No need for a variable." Use:

int x = 10; // 10 → integer literal float pi

= 3.14; // 3.14 → floating literal char grade

= 'A'; // 'A' → character literal string name =

"C++"; // "C++" → string literal bool flag =

true; // true → boolean literal

➢ control flow statements

1.What are conditional statements in C++? Explain the if-else and switch statements.

Conditional Statements in C++

Conditional statements are used to make decisions in a program.

They check a condition (true/false) and execute code based on that result.

C++ provides: if

if-else if-else

if-else

switch

1. if-else Statement

Used when we want to execute one block if the condition is true, otherwise another block.

Syntax:

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

Example:

```
#include <iostream>
using namespace std;
int main() {     int age;
   cout << "Enter your age: ";
cin >> age;

   if (age >= 18) {
      cout << "You are eligible to vote." << endl;
   } else {
      cout << "You are not eligible to vote." << endl;
```

```
    }
    return 0;
}
```

If age >= 18 → prints eligible

Otherwise → prints not eligible

2. switch Statement

- ❾ Used when we have multiple choices.
- ❾ Instead of many if-else if, we use switch.

Syntax:

```
switch(expression) {
case value1:        //
code        break;
case value2:        //
code        break;
    ...
    default:
        // code if no case matches
}
```

Example: #include

<iostream> using

namespace std; int

main() { int day;

  cout << "Enter day number (1-7): ";

cin >> day; switch(day) { case 1:

cout << "Monday"; break; case 2:

cout << "Tuesday"; break; case 3:

cout << "Wednesday"; break; case 4:

cout << "Thursday"; break; case 5:

cout << "Friday"; break; case 6: cout

<< "Saturday"; break; case 7: cout

<< "Sunday"; break; default: cout <<

"Invalid day!";

  }

  return 0;

}

If day = 3 → prints Wednesday

If input doesn't match any case → default executes.

2.What is the difference between for, while, and do-while loops in C++?

Ans->Difference between `for`, `while`, and `do-while` loops in C++

Feature	`for` Loop	`while` Loop	`do-while` Loop
Conditio Check	Before loc	Before loop s	After loop executes once
Runs at l once?	No	No	Yes
Use Case	When iter are knowr	When condit loop is neede	When loop must run at least once
Syntax Initializa	Inside loo header	Before loop	Before loop
Commor	Counting l	Sentinel-cont loops	Menu-driven programs

3.How are break and continue statements used in loops? Provide examples.

Break and Continue Statements in C++

Both break and continue are jump statements used inside loops (for, while, dowhile).

1. break Statement

->Used to terminate the loop immediately.

->Control moves outside the loop.

❾ Example (using break in for loop):

```
#include <iostream> using
namespace std; int main() {
for (int i = 1; i <= 10; i++) {
if (i == 5) {
        break;  // loop ends when i == 5
    }
    cout << i << " ";
  }
  return 0;
}
```

Output:

1 2 3 4

Loop stops completely when i = 5.

2. continue Statement

-> Used to skip the current iteration and go to the next iteration of the loop.

-> Loop does not terminate, only skips that round.

O Example (using continue in for loop):

```
#include <iostream>
using namespace std; int
main() {
   for (int i = 1; i <= 10; i++) {
if (i == 5) {
          continue;  // skips printing 5
       }
       cout << i << " ";
   }
   return 0;
}
```

Output:

1 2 3 4 6 7 8 9 10

When i = 5, printing is skipped, but loop continues.

4.Explain nested control structures with an example.

Ans-> Nested Control Structures in C++

-> A control structure means statements that control the flow of execution in a program (like if, for, while, switch).

-> When one control structure is placed inside another, it is called a nested control structure.

❾ Types of Nested Control Structures

1. Nested if-else
2. Nested loops (for, while, do-while)
3. Mix of conditions and loops

1. Nested if-else Example.

```
#include <iostream> using
namespace std; int main()
{    int age = 20;    char
gender = 'M';

  if (age >= 18) {       if
(gender == 'M') {
        cout << "You are an adult male." << endl;
    } else {
        cout << "You are an adult female." << endl;
    }
```

```
    } else {
        cout << "You are a minor." << endl;
    }
    return 0;
}
```

2. Nested for Loop Example

```
#include <iostream> using
namespace std; int main() {
    for (int i = 1; i <= 3; i++) {       // outer loop (rows)
for (int j = 1; j <= 3; j++) {    // inner loop (columns)
cout << "(" << i << "," << j << ") ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
```

1. What is a function in C++? Explain the concept of function declaration, definition, and calling.

Function and scope

Ans-> What is a function in C++?

A **function** in C++ is a block of code that performs a specific task and can be reused.

Function Declaration, Definition, and Calling:

Declaration: Tells the compiler about the function's name, return type, and parameters.

Example: `int add(int, int);`

Definition: Contains the actual body (code) of the function. Example:

```
int add(int a, int b) {
    return a + b;
}
```

Calling: Invokes the function to execute its code.

Example: `add(5, 3);`

2.What is the scope of variables in C++? Differentiate between local and global scope.

Ans->What is the scope of variables in C++?

- The **scope** of a variable in C++ refers to the **region of the program** where the variable is **accessible or valid**.

- Difference between Local and Global Scope:

Scope Type	Description	Example Location
Local	Variable declared **ins** or block	Accessible only within that block
Global	Variable declared **ou** functions	Accessible from **anywhere** in the program

3. Explain recursion in C++ with an example.

Ans->Recursion in C++

Recursion is a process where a function **calls itself** to solve a smaller part of a problem.

Example: Factorial using Recursion

```
#include <iostream> using

namespace std;
```

```
int factorial(int n) {        if (n == 0)    // base case
return 1;          else              return n * factorial(n -
1);    // recursive call
}
int main() {       cout << factorial(5);
// Output: 120        return 0;
}
```

4. What are function prototypes in C++? Why are they used?

Ans->

What are function prototypes in C++?

A **function prototype** is a **declaration** of a function that tells the compiler about the function's **name, return type, and parameters**, **before its actual definition**.

Example: `int add(int, int);  // Function prototype`

❾ **Why are they used?**

- To **inform the compiler** about a function **before** it is used.
- To enable **function calls before definitions**.
- Helps in **type checking** of function arguments during compilation.

1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays?

Ans->

➢ Array and strings

❾ What are arrays in C++?

An **array** in C++ is a **collection of elements** of the **same data type** stored in **contiguous memory locations**. Each element can be accessed using an index.

Example:

int `numbers[5] = {1, 2, 3, 4, 5};`

❾ Difference between Single-Dimensional and Multi-Dimensional Arrays in C++

Feature	**Single-Dimensio**	**Multi-Dimensional Array**
Structure	Stores elements **row/line**	Stores elements in **rows and columns** (2D or more)
Declaratic Syntax	`int arr[5];`	`int arr[3][4];`
Accessing Elements	`arr[2]` (access element)	`arr[1][2]` (accesses element in 2nd row, 3rd column)

Feature	Single-Dimensio	Multi-Dimensional Array
Use Case	Useful for **linear** scores, list of nu	Useful for **tabular data** like matrices, tables
Memory L	Stored in **contig linear memory**	Stored in **row-major order** (rows stored one after another)
Complexit	**Simple** to declar	**More complex** to manage and visualize
Dimensio	Only **1 index** nee	Requires **2 or more indices**
Example Initializati	`int arr[3] 20, 30};`	`int arr[2][2] = {{1,2},{3,4}};`
Iteration	Single loop used	Nested loops needed

2. Explain string handling in C++ with examples ?

Ans->

1. Using C-style Strings (Character Arrays)
    - Stored as arrays of characters ending with a **null character (\0)**.
    - Header: `<cstring>` ❾ **Example:**

```
#include <iostream>
#include <cstring> using
namespace std;
 int main() {     char
str1[20] = "Hello";     char
```

```
str2[20];     strcpy(str2,
str1);         // Copying
cout << strlen(str1);
// Length     cout <<
strcmp(str1, str2); //
Comparing     return 0;
}
```

2. **Using C++ `string` Class (Recommended)**

- Easier and safer to use.
- Header: `<string>` **Example:**

```
#include <iostream>
#include <string> using
namespace std;
 int main() {     string
str1 = "Hello";     string
str2 = "World";
     string str3 = str1 + " " + str2; //
Concatenation
    cout << str3 << endl;             // Output: Hello
World     cout << str1.length() << endl;  // Length of
string     cout << str2.substr(1, 3);      // Substring:
"orl"     return 0;
}
```

3 **.** How are arrays initialized in C++? Provide examples of both 1D and 2D arrays?

Ans->arrays in C++ can be initialized **at the time of declaration** using curly braces `{}`.

1. One-Dimensional (1D) Array Initialization ->

Full Initialization

```
int arr[5] = {10, 20, 30, 40, 50};
```

All 5 elements are given values.

-> Partial Initialization

```
int arr[5] = {10, 20};
```

First two elements get values, rest are automatically set to `0` (only if global/static).

-> Size Automatically Detected

```
int arr[] = {1, 2, 3};
```

Compiler sets the size to 3.

Using a Loop

```
int arr[5]; for(int i = 0; i
< 5; i++) {      arr[i] =
i * 10;
}
```

2. Two-Dimensional (2D) Array Initialization ->

Row-wise Initialization

```
int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Each row is defined separately.

-> Flat Initialization

```
int arr[2][3] = {1, 2, 3, 4, 5, 6};
```

Values are filled row-by-row.

->Partial Initialization

int `arr[2][3] = { {1}, {4, 5} };`

4.Explain string operations and functions in C++ ?

Ans->String Operations and Functions in C++

In C++, strings are commonly handled using the **`string` class** from the `<string>` header. It provides many **built-in functions** to perform various operations.

-> Common String Operations and Functions

Operation	Function/Operator	Example
Length	`.length()`	`str.length();`
Concaten	+ operator	`str1 + str2;`
Access character	`[]` operator	`str[0];`
Substring	`.substr(star`	`str.substr(2, 3);`
Comparis	`==, !=, >, <`	`str1 == str2;`
Find subst	`.find("text"`	`str.find("lo");`
Replace	`.replace(pos str)`	`str.replace(0, 3, "Hi");`

Operation	Function/Operator	Example
Insert	`.insert(pos,`	`str.insert(5, "World");`
Erase	`.erase(pos,`	`str.erase(0, 2);`
Clear	`.clear()`	`str.clear();`
Empty che	`.empty()`	`str.empty();`

Example

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

int main() {      string
str1 = "Hello";      string
str2 = "World";
     string result = str1 + " " + str2;      //
Concatenation      cout << "Combined: " << result << endl;
cout << "Length: " << result.length() << endl;      cout <<
"Substring: " << result.substr(0, 5) << endl;      cout <<
"Find: " << result.find("World") << endl;
```

➢ . Introduction to Object-Oriented Programming

666. Introduction to Object-Oriented Programming

1. Explain the key concepts of Object-Oriented Programming (OOP) ?

Ans->Key Concepts of Object-Oriented Programming (OOP)

1. **Class** ○ A blueprint or template for creating objects.
   ○ Defines data (attributes) and functions (methods) related to an object.
2. **Object** ○ An instance of a class.
   ○ Contains actual data and can perform behaviors defined by the class.
3. **Encapsulation** ○ Wrapping data and methods into a single unit (class). ○ Controls access to data by using access specifiers (`private`, `public`, `protected`).
4. **Inheritance** ○ Mechanism where one class (derived class) inherits properties and behaviors from another class (base class).
   ○ Promotes code reusability.
5. **Polymorphism** ○ Ability of functions or methods to behave differently based on the object calling them.
   ○ Two types: **Compile-time** (function overloading, operator overloading) and **Run-time** (virtual functions).
6. **Abstraction** ○ Hiding complex implementation details and showing only the necessary features to the user. ○ Achieved using abstract classes and interfaces.

2. What are classes and objects in C++? Provide an example.?

Ans->Classes and Objects in C++

- **Class:**
  A **blueprint** or **template** for creating objects. It defines **data members** (variables) and **member functions** (methods) that operate on the data.
- **Object:**

An **instance** of a class. It holds actual values for the data members defined in the class.

Example:

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

// Class definition
class Car { public:
    string brand;
int year;

    void display() {
        cout << "Brand: " << brand << ", Year: " << year
<< endl;
    }
};
int main() {
    Car car1;       // Creating an object of class Car
car1.brand = "Toyota";     car1.year = 2020;
     car1.display(); // Output: Brand: Toyota,
Year:
2020
    return 0;
}
```

3.What is inheritance in C++? Explain with an example.

Ans->**What is Inheritance in C++?**

Inheritance is a feature of Object-Oriented Programming where a **new class (derived class)** acquires the properties (data members) and behaviors (member functions) of an existing class (base class). It promotes **code reusability**.

- Example:

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

// Base class class
Animal { public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};

// Derived class inherits from Animal
class Dog : public Animal { public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};  int main()
{     Dog
myDog;

    myDog.eat();   // Inherited function from Animal
class
    myDog.bark();  // Function of Dog class

    return 0;
}
```

4.What is encapsulation in C++? How is it achieved in classes?

Ans->What is Encapsulation in C++?

Encapsulation is the concept of **bundling data (variables) and methods (functions)** that operate on the data into a single unit called a **class**. It **restricts direct access** to some of the object's components, which helps to **protect the data** and **maintain integrity**.

How is Encapsulation Achieved in Classes?

- By using **access specifiers**:
  - `private`: Members are accessible **only within the class**.
  - `public`: Members are accessible **from outside the class**.
  - `protected`: Accessible within the class and by derived classes.
- Typically, **data members are made private** and accessed or modified using **public getter and setter functions**.

Example:

```
#include <iostream> using
namespace std;
 class BankAccount
{ private:
    double balance;  // Private data member
 public:
    // Setter function to update balance safely
void setBalance(double amount) {         if
(amount >= 0)             balance = amount;
else
            cout << "Invalid amount!" << endl;
    }

    // Getter function to access balance
double getBalance() {         return
balance;
    }
};
int main() {
BankAccount acc;
    acc.setBalance(1000.50);          // Set balance
cout << "Balance: " << acc.getBalance() << endl;
```

```
// Get balance

    acc.setBalance(-500);              // Invalid amount

    return 0;
}
```