Module-3 Introduction to OOPS Programming

(1).What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

Ans:

PROCEDURAL PROGRAMMING	OBJECT-ORIENTED PROGRAMMING
Program is divided into functions	Program is divided into object
Less secure (no data hiding)	More secure (supports data hiding)
Best for small to medium programs	Ideal for large, complex applications
Difficult to manage large codebases due to lack of modularity	Easier to maintain and extend due to modular design.

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

ANS.

ADVANTAGE OF OOP:

- Code is organized into classes and objects, making it modular and easier to understand, debug, and maintain.
- Classes allow code reuse through *inheritance*. A new class can use properties and behaviours of an existing one without rewriting.
- Changes are easier—modify a class and all objects using it benefit automatically. Large projects are more manageable.
- Supports polymorphism (same function/method name with different behaviours). This improves code flexibility and reduces complexity.

ADVANTAGE OF OOP:

- Code is split into functions, but managing large projects becomes messy as functions interact with global data.
- Reusability is limited—you often need to copy and paste functions, which leads to redundancy.

- Global data is often shared across functions, so accidental modifications are common, reducing security.
- With increasing project size, dependencies between functions and global data make modifications error-prone.

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

Ans: 1. Install a C++ Compiler

C++ programs need to be compiled into machine code before execution. Some common compilers are:

- GCC (GNU Compiler Collection) – Popular on Linux/Unix and also available on Windows (via MinGW or Cygwin).
- Clang – Lightweight and modern, often used on macOS/Linux.
- MSVC (Microsoft Visual C++ Compiler) – Comes with Microsoft Visual Studio (Windows).

👉 Choose a compiler depending on your operating system.

2. Install an IDE or Code Editor

An IDE (Integrated Development Environment) makes coding easier with features like debugging, syntax highlighting, and auto-completion. Popular choices:

- Visual Studio (Windows) – Full-featured IDE with MSVC compiler.
- Code::Blocks (Windows/Linux) – Lightweight and beginner-friendly.
- Dev-C++ (Windows) – Simple IDE for learning.
- CLion (cross-platform) – Powerful but paid (by JetBrains).
- VS Code (cross-platform) – Text editor with C++ extensions.

Beginners often start with Code::Blocks or Visual Studio.

3. Configure the Compiler with the IDE

After installation:

- If you use Visual Studio, it comes pre-configured with MSVC.
- In Code::Blocks, link it with MinGW (if on Windows).
- In VS Code, install the C++ extension and configure tasks.json to tell the editor which compiler to use.

4. Verify the Installation

- Open a terminal/command prompt.
- Type the command to check version:
    - For GCC: g++ --version
    - For Clang: clang++ --version
    - For MSVC: cl
      If you see version details, the compiler is working.

5. Write Your First Program

Example (hello.cpp):

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

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

6. Compile and Run

- **GCC/Clang (Linux/macOS/Windows with MinGW):**

```
g++ hello.cpp -o hello
./hello   # run
```

(4). What are the main input/output operations in C++? Provide examples.

Ans: In C++, **input/output (I/O) operations** are mainly handled through the **iostream** library using **streams**.
A *stream* is like a flow of data:

- **Input stream** → data flows from input devices (keyboard, files) into the program.
- **Output stream** → data flows from the program to output devices (screen, files).

1. Console Input (cin)

Used to take input from the keyboard.

Example:

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

int main() {
    int age;
    cout << "Enter your age: ";   // Output to user
    cin >> age;                   // Input from user
    cout << "You entered: " << age << endl;
    return 0;
}
```

cin >> variable; extracts input from the user.

2. Console Output (cout)

Used to display information on the screen.

Example:

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

int main() {
    cout << "Hello, World!" << endl;      // Print message
    cout << "Sum of 5 + 10 = " << 5 + 10 << endl;  // Output expression result
    return 0;
}
```

<< is the *insertion operator*, used to send data to output.

3. Character Input (get(), getline())

- cin.get() → reads a single character (including spaces).
- getline() → reads an entire line of text (including spaces).

Example:

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

int main() {
    char ch;
    string name;

    cout << "Enter a character: ";
    cin.get(ch);       // reads one character
    cout << "You entered: " << ch << endl;

    cin.ignore();      // ignore leftover newline character

    cout << "Enter your full name: ";
    getline(cin, name); // reads full line with spaces
    cout << "Hello, " << name << "!" << endl;

    return 0;
}
```

4. File Input/Output (ifstream, ofstream, fstream)

Used for reading from and writing to files.

Example:

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

int main() {
    // Write to file
    ofstream outFile("example.txt");
    outFile << "Hello, File!" << endl;
    outFile.close();

    // Read from file
    ifstream inFile("example.txt");
    string line;
    getline(inFile, line);
    cout << "File says: " << line << endl;
    inFile.close();

    return 0;
}
```

Main I/O Operations

- cin → console input
- cout → console output
- cin.get(), getline() → character/string input
- ifstream, ofstream, fstream → file input/output.

(5). What are the different data types available in C++? Explain with examples.

Ans: Basic (Fundamental) Data Types:

These are the primary building blocks.

Data Type	Description	Example
int	Stores integers (whole numbers)	int age = 25;
float	Stores decimal numbers (single precision, ~6 digits)	float price = 19.99;
double	Stores decimal numbers (double precision, ~15 digits)	double pi = 3.1415926535;
char	Stores a single character (1 byte)	char grade = 'A';
bool	Stores true or false (1 byte)	bool isPassed = true;
void	Represents no value (used in functions)	void greet() { cout << "Hello"; }

2. Derived Data Types

Built from basic data types.

Data Type	Description	Example
Array	Collection of elements of the same type	int marks[5] = {90, 85, 88, 92, 75};
Pointer	Stores memory address of another variable	int x = 10; int *ptr = &x;

Data Type	Description	Example
Reference	Alias for another variable	int a = 5; int &ref = a;
Function	A block of code returning a value	int add(int a, int b) { return a+b; }

3. User-defined Data Types:

Created by programmers.

Data Type	Description	Example
struct	Groups related variables (like records)	struct Student { int id; string name; };
class	OOP feature that combines data + functions	class Car { public: void drive() { cout << "Driving"; } };
enum	Defines a set of named constants	enum Color { RED, GREEN, BLUE };
typedef / using	Creates an alias for data types	typedef unsigned int uint;

Example:

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

// User-defined type
struct Student {
    int rollNo;
    string name;
};

int main() {
    // Basic data types
    int age = 20;
    float weight = 55.5;
    double pi = 3.14159;
    char grade = 'A';
    bool isPassed = true;

    // Derived data types
    int marks[3] = {85, 90, 95};
    int x = 10;
    int *ptr = &x;
```

```
    // User-defined data type
    Student s1 = {101, "Harshil"};

    // Output
    cout << "Age: " << age << endl;
    cout << "Weight: " << weight << endl;
    cout << "Pi: " << pi << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << isPassed << endl;
    cout << "Marks[1]: " << marks[1] << endl;
    cout << "Pointer value: " << *ptr << endl;
    cout << "Student Roll No: " << s1.rollNo << ", Name: " << s1.name << endl;

    return 0;
}
```

Basic → int, float, double, char, bool, void

Derived → arrays, pointers, references, functions

User-defined → struct, class, enum, typedef

(6). Explain the difference between implicit and explicit type conversion in C++.

Ans: 1. Implicit Type Conversion (Type Promotion / Type Casting by Compiler)

- Done automatically by the compiler.
- Happens when you assign a smaller data type to a larger one, or mix different types in an expression.
- No data loss in most cases (but can sometimes lose precision when converting from double → int).

Example:

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

int main() {
    int a = 10;
    double b = 5.5;

    double result = a + b;  // 'a' is implicitly converted to double
    cout << "Result = " << result << endl;

    char ch = 'A';   // ASCII value of 'A' = 65
    int x = ch;      // char is implicitly converted to int
    cout << "Character 'A' as int = " << x << endl;

    return 0;
}
```

int → double and char → int are done automatically.

2. Explicit Type Conversion (Type Casting by Programmer)

- Done manually by the programmer.
- Used when you want control over how conversion happens.
- Two main ways:
    1. C-style cast: (type)expression
    2. C++ cast operators:
        - static_cast<type>(expression)
        - dynamic_cast<type>(expression) (used with inheritance)
        - const_cast<type>(expression) (remove const)
        - reinterpret_cast<type>(expression) (low-level bit conversion)

Example:

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

int main() {
    double pi = 3.14159;

    int intPi1 = (int)pi;              // C-style explicit cast
    int intPi2 = static_cast<int>(pi);  // C++ style explicit cast

    cout << "Original double: " << pi << endl;
    cout << "After C-style cast: " << intPi1 << endl;
    cout << "After static_cast: " << intPi2 << endl;

    return 0;
}
```

double → int is **forced** (fractional part is lost).

Example in one line:

- **Implicit:** int x = 10; double y = x;
- **Explicit:** double pi = 3.14; int n = (int)pi;

(7). What are the different types of operators in C++? Provide examples of each

Ans: **1. Arithmetic Operators**

Used for mathematical operations.

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

```
int a = 10, b = 3;
cout << a + b << endl;  // 13
cout << a / b << endl;  // 3 (integer division)
cout << a % b << endl;  // 1
```

2. Relational (Comparison) Operators

Used to compare values. Returns true (1) or false (0).

Operator	Meaning	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

```
int x = 5, y = 10;
cout << (x < y) << endl;  // 1 (true)
cout << (x == y) << endl; // 0 (false)
```

3. Logical Operators

Used in decision making.

Operator	Meaning	Example
&&	Logical AND	(a > 0 && b > 0)
`	`	`
!	Logical NOT	!(a > b)

```cpp
bool a = true, b = false;
cout << (a && b) << endl;  // 0
cout << (a || b) << endl;  // 1
cout << (!a) << endl;      // 0
```

4. Assignment Operators

Used to assign values to variables.

Operator	Example	Meaning
=	x = 5	Assigns 5 to x
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2
%=	x %= 2	x = x % 2

```cpp
int x = 10;
x += 5;   // x = 15
x *= 2;   // x = 30
cout << x;
```

5. Bitwise Operators

Work on bits (0s and 1s).

Operator	Meaning	Example
&	Bitwise AND	a & b
||	Bitwise or ||	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

```cpp
int a = 5, b = 3;  // (0101, 0011)
cout << (a & b) << endl; // 1
cout << (a | b) << endl; // 7
cout << (a ^ b) << endl; // 6
```

(8). Explain the purpose and use of constants and literals in C++.

Ans: C++, both **constants** and **literals** are used to represent fixed values, but they serve slightly different purposes.

1. Constants in C++

A **constant** is a variable whose value cannot be changed after initialization.

They are useful when you want to protect certain values from modification (e.g., mathematical constants like π, configuration values like MAX_SIZE).

Ways to Define Constants

1. **Using const keyword**

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

int main() {
    const double PI = 3.14159;  // constant variable
    int radius = 5;
    double area = PI * radius * radius;

    cout << "Area = " << area << endl;
    // PI = 3.14;  // ❌ Error: cannot modify constant
    return 0;
}
```

2. **Using #define (Preprocessor Directive)**

```cpp
#include <iostream>
#define MAX 100   // constant macro

int main() {
    cout << "Maximum value = " << MAX << endl;
    return 0;
}
```

3. **Using constexpr (C++11 and later)**

- Evaluated at compile time.
- Useful for performance optimization.

```cpp
constexpr int square(int x) { return x * x; }

int main() {
    int arr[square(3)];  // size = 9, known at compile time
    cout << "Array size = " << sizeof(arr)/sizeof(int) << endl;
}
```

2. Literals in C++

A **literal** is a fixed value that directly appears in the program.

They represent constant values **directly written in code** (numbers, characters, strings, etc.).

Type	Example	Description
Integer	10, -25, 0x1A	Decimal, octal, or hexadecimal integers
Floating-point	3.14, -0.001, 2.5e3	Real numbers (with decimal or scientific notation)
Character	'A', '9', '\n'	Single characters (enclosed in single quotes)
String	"Hello", "C++ is fun!"	Sequence of characters (double quotes)
Boolean	true, false	Logical values
Null Pointer	nullptr	Represents a null pointer (C++11 and later)

Example Program (Constants + Literals):

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

int main() {
    const double PI = 3.14159;   // constant
    int radius = 5;              // variable
    double area = PI * radius * radius;  // literal values used in expressions

    cout << "Radius = " << radius << endl;
    cout << "Area = " << area << endl;

    cout << "Character literal: " << 'A' << endl;
    cout << "String literal: " << "Hello, World!" << endl;
    cout << "Boolean literal: " << true << endl;

    return 0;
}
```

In short:

- **Constants** = Named fixed values.
- **Literals** = Raw values directly written in code.

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

Ans: The two most common conditional statements are:

- **if-else**
- **switch**

1. if-else Statement

- Executes a block of code if a condition is true.
- You can also add an else block to run code when the condition is false.
- Multiple conditions can be handled using else if.

Syntax:

```
if (condition) {
    // executes if condition is true
} else {
    // executes 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;
}
```

Here:

- If age >= 18 → first block runs.
- Otherwise → else block runs.

2. if-else-if Ladder

Used when there are multiple conditions.

```
int marks;
cin >> marks;

if (marks >= 90) {
    cout << "Grade: A";
} else if (marks >= 75) {
    cout << "Grade: B";
} else if (marks >= 50) {
    cout << "Grade: C";
} else {
    cout << "Fail";
}
```

3. switch Statement

- Used when you want to test a variable against multiple fixed values (called *cases*).
- Works only with **integral/enum/char types** (not with floating-point or strings in traditional C++).

Syntax:

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

Example:

```cpp
#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;
}
```

Here:

- If day == 1, prints **Monday**.
- If no case matches, default runs.
- break prevents execution from *falling through* to the next case.

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

Ans: In C++, **loops** are used to repeatedly execute a block of code until a condition is satisfied. The three main loops are: **for**, **while**, and **do-while**.

1. for Loop

- Used when the **number of iterations is known** in advance.
- Initialization, condition, and update are written together.

Syntax:

```cpp
for (initialization; condition; update) {
    // code
}
```

Example:

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

int main() {
    for (int i = 1; i <= 5; i++) {
        cout << i << " ";
    }
    return 0;
}
```

Output: 1 2 3 4 5

Runs from i = 1 to 5, updating i++ each time

2. while Loop

- Used when the **number of iterations is not known** in advance.
- Checks the condition **before** entering the loop.

Syntax:

```cpp
while (condition) {
    // code
}
```

Example:

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

int main() {
    int i = 1;
    while (i <= 5) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```

Output: 1 2 3 4 5

Condition checked before execution → if false at start, loop may not run at all.

3. do-while Loop

- Similar to while, but condition is checked **after** executing the loop.
- Ensures the loop runs **at least once**, even if the condition is false.

Syntax:

```
do {
    // code
} while (condition);
```

Example:

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

int main() {
    int i = 1;
    do {
        cout << i << " ";
        i++;
    } while (i <= 5);
    return 0;
}
```

Output: 1 2 3 4 5

Even if i = 10 at start, loop runs once.

Quick Example (all three doing the same task):

Print numbers 1 to 5:

```
// for loop
for (int i = 1; i <= 5; i++) cout << i << " ";

// while loop
int j = 1;
while (j <= 5) {
    cout << j << " ";
    j++;
}

// do-while loop
int k = 1;
do {
    cout << k << " ";
    k++;
} while (k <= 5);
```

In summary:

- **Use for when you know how many times to repeat.**
- **Use while when repeating until a condition becomes false.**
- **Use do-while when the loop must run at least once.**

(11). How are break and continue statements used in loops? Provide examples.

Ans: 1. break Statement

- Purpose: Immediately terminates the loop (or switch) where it appears.
- The control jumps to the statement after the loop.

Example: Using break

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            cout << "Breaking the loop at i = " << i << endl;
            break;  // Exit the loop completely
        }
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
1 2 3 4 Breaking the loop at i = 5
```

When i becomes 5, the loop **ends completely**.

2. continue Statement

- **Purpose**: Skips the **current iteration** and jumps to the **next iteration** of the loop.
- The loop does **not terminate**, only that iteration is skipped.

Example: Using continue

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            cout << "Skipping i = " << i << endl;
            continue;  // Skip this iteration
        }
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
1 2 3 4 Skipping i = 5
6 7 8 9 10
```

Key Difference

- break: **Stops** the loop completely.
- continue: **Skips one iteration**, but keeps the loop running.

(12). Explain nested control structures with an example.

Ans: What are Nested Control Structures?

- A nested control structure means placing one control structure (like a loop, if, or switch) inside another.
- This allows solving more complex problems (like patterns, decision-making inside loops, etc.).

◇ Example 1: Nested if statements

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

int main() {
    int age = 20;
    char gender = 'M';

    if (age >= 18) {                // Outer if
        if (gender == 'M') {        // Inner if
            cout << "Adult Male" << endl;
        } else {
            cout << "Adult Female" << endl;
        }
    } else {
        cout << "Minor" << endl;
    }

    return 0;
}
```

Output:

```
Adult Male
```

Here, the **inner if** executes only if the **outer if condition** is true.

- **Example 2: Nested for loops (Pattern Printing)**

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

int main() {
    for (int i = 1; i <= 3; i++) {          // Outer loop (rows)
        for (int j = 1; j <= 5; j++) {      // Inner loop (columns)
            cout << "* ";
        }
        cout << endl;   // Move to next line
    }
    return 0;
}
```

Output:

```
* * * * *
* * * * *
* * * * *
```

Nested control structures = one control inside another.

Common examples:

- **if inside if**
- **for inside for**
- **loop inside if**

Useful for decision making and repeated patterns.

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

Ans: What is a Function?

A function in C++ is a block of code that performs a specific task.

- It helps in reusability (write once, use many times).
- Makes the program modular and easy to read.

◇ Types of Functions

1. Library (built-in) functions → e.g., sqrt(), pow(), printf(), cin, etc.
2. User-defined functions → Functions that you create yourself.

◇ Parts of a Function

1. Function Declaration (Prototype)

- Tells the compiler the name, return type, and parameters of the function.
- Ends with a semicolon (;).

```
return_type function_name(parameter_list);
```

Example:

```
int add(int a, int b);  // declaration
```

2. Function Definition

- Contains the **actual body** (code) of the function.

```
return_type function_name(parameter_list) {
    // function body
    return value;
}
```

Example:

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

3. Function Call

- Tells the program to **execute** the function.

```
function_name(arguments);
```

Example:

```
int sum = add(5, 10);  // calling function
```

Complete Example:

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

// 1. Function Declaration
int add(int a, int b);

int main() {
    int x = 5, y = 10;

    // 3. Function Call
    int result = add(x, y);

    cout << "Sum = " << result << endl;
    return 0;
}

// 2. Function Definition
int add(int a, int b) {
    return a + b;
}
```

Output:

```
Sum = 15
```

Declaration: Tells compiler about the function (name, return type, parameters).

Definition: Actual implementation (code inside function).

Calling: Executes the function from main() or another function.

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

Ans: Scope of Variables in C++

The scope of a variable in C++ defines the region of the program where the variable can be accessed or used.

It depends on where the variable is declared in the program.

In C++, variable scope can mainly be:

- Local Scope
- Global Scope

1. Local Scope

- A variable declared inside a function, block, or loop is called a local variable.
- It can only be accessed within that block/function.
- It is created when the block is entered and destroyed when the block ends (lifetime is limited).
- Memory is allocated on the stack.

Example:

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

void display() {
    int x = 10;  // local variable
    cout << "Local variable x = " << x << endl;
}

int main() {
    display();
    // cout << x;  // ❌ Error: x is not accessible here
    return 0;
}
```

2. Global Scope

- A variable declared outside all functions is called a global variable.
- It can be accessed by all functions in the program (unless shadowed by a local variable of the same name).
- It exists for the entire lifetime of the program.
- Memory is allocated in the data segment.

Example:

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

int g = 100;  // global variable

void display() {
    cout << "Global variable g = " << g << endl;
}

int main() {
    cout << "Accessing global variable g = " << g << endl;
    display();
    return 0;
}
```

(15). Explain recursion in C++ with an example.

Ans: Recursion in C++

- Definition:
  Recursion is a programming technique in which a function calls itself either directly or indirectly until a base condition is met.
- Every recursive function must have:
  1. Base Case: The condition where the function stops calling itself.
  2. Recursive Case: The part where the function calls itself with modified arguments.

General Syntax:

```cpp
returnType functionName(parameters) {
    if (base_condition) {
        // stopping point
        return some_value;
    } else {
        // recursive call
        return functionName(modified_parameters);
    }
}
```

Example 1: Factorial using Recursion

The factorial of a number n is defined as:

```
n! = n × (n-1) × (n-2) × ... × 1
```

With recursion:

- **Base case:** 0! = 1
- **Recursive case:** n! = n × (n-1)!

```cpp
#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() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Factorial of " << num << " is " << factorial(num) << endl;
    return 0;
}
```

Sample Output:

```
Enter a number: 5
Factorial of 5 is 120
```

Example 2: Fibonacci Series using Recursion

Fibonacci sequence:

```
0, 1, 1, 2, 3, 5, 8, ...
```

Base cases:

F(0) = 0, F(1) = 1

Recursive case:

F(n) = F(n-1) + F(n-2)

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

int fibonacci(int n) {
    if (n == 0) return 0;      // base case 1
    if (n == 1) return 1;      // base case 2
    return fibonacci(n-1) + fibonacci(n-2);  // recursive call
}

int main() {
    int terms;
    cout << "Enter number of terms: ";
    cin >> terms;

    cout << "Fibonacci series: ";
    for (int i = 0; i < terms; i++) {
        cout << fibonacci(i) << " ";
    }
    cout << endl;
    return 0;
}
```

Key Points about Recursion

- **Uses function call stack → Each recursive call is pushed onto the stack until the base case is reached.**
- **May cause stack overflow if base case is missing or incorrect.**
- **Sometimes less efficient than loops (due to overhead of repeated function calls), but makes code shorter and easier to understand.**

(16). What are arrays in C++? Explain the difference between single-dimensional and multidimensional arrays.

Ans: Arrays in C++

- An array is a collection of elements of the same data type stored in contiguous memory locations.
- Each element in an array is accessed using an index (subscript).
- In C++, arrays have fixed size (declared at compile time).

Syntax:

```
data_type array_name[size];
```

Example:

```
int numbers[5] = {10, 20, 30, 40, 50};
cout << numbers[0]; // prints 10
cout << numbers[4]; // prints 50
```

Types of Arrays

1. Single-Dimensional Array

- Represents a **linear list** of elements.
- Elements are accessed with a **single index**.

Declaration:

```
int arr[5];   // array of 5 integers
```

Example:

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

int main() {
    int marks[5] = {90, 85, 78, 92, 88};

    cout << "Student Marks: ";
    for (int i = 0; i < 5; i++) {
        cout << marks[i] << " ";
    }
    return 0;
}
```

Output:

```
Student Marks: 90 85 78 92 88
```

2. Multidimensional Array

- An array of arrays.
- Common type: **2D array**, which can be visualized as a **table (rows & columns)**.
- Elements are accessed with **multiple indices**.

Declaration:

```
int arr[3][3];   // 2D array with 3 rows and 3 columns
```

Example:

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

int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };

    cout << "2D Matrix: \n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
2D Matrix:
1 2 3
4 5 6
```

(17). How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Ans: **Array Initialization in C++**

In C++, arrays can be initialized in several ways at the time of declaration.

1. Initialization of a 1D Array

Syntax:

```
data_type array_name[size] = {values};
```

Examples:

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

int main() {
    // Method 1: Full initialization
    int arr1[5] = {10, 20, 30, 40, 50};

    // Method 2: Partial initialization (rest are set to 0)
    int arr2[5] = {1, 2};   // becomes {1, 2, 0, 0, 0}

    // Method 3: Compiler deduces size
    int arr3[] = {5, 10, 15, 20}; // size = 4

    // Displaying arr1
    cout << "arr1: ";
    for (int i = 0; i < 5; i++) {
        cout << arr1[i] << " ";
    }
    cout << endl;

    // Displaying arr2
    cout << "arr2: ";
    for (int i = 0; i < 5; i++) {
        cout << arr2[i] << " ";
    }
    cout << endl;

    return 0;
}
```

2. Initialization of a 2D Array

Syntax:

```
data_type array_name[rows][cols] = { {row1}, {row2}, ... };
```

Examples:

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

int main() {
    // Method 1: Row-wise initialization
    int matrix1[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Method 2: Single list initialization (row by row)
    int matrix2[2][3] = {1, 2, 3, 4, 5, 6};

    // Method 3: Partial initialization (remaining = 0)
    int matrix3[2][3] = { {7}, {8, 9} }; // { {7,0,0}, {8,9,0} }

    // Display matrix1
    cout << "matrix1:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix1[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output (matrix1):

```
matrix1:
1 2 3
4 5 6
```

Key Points:

- **If you don’t initialize, array elements contain garbage values (for local arrays).**
- **If partially initialized, the remaining elements are set to 0.**
- **For 2D arrays, values are filled row by row.**

(18). Explain string operations and functions in C++.

Ans:

1. C-style strings (character arrays)
2. C++ string class (from <string> library)

1. C-Style Strings (Character Arrays)

- Strings are stored as arrays of characters, ending with a null character '\0'.
- Declared like this:

```
char str[20] = "Hello";
```

Example (C-Style):

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    cout << "Length of str1: " << strlen(str1) << endl;

    strcat(str1, str2); // concatenate
    cout << "Concatenated: " << str1 << endl;

    strcpy(str2, "C++");
    cout << "Copied str2: " << str2 << endl;

    cout << "Comparison: " << strcmp("abc", "abd") << endl;

    return 0;
}
```

2. C++ string Class (Modern Strings)

- C++ provides a **string class** in the <string> header.
- Much easier and safer than C-style strings.

Declaration:

```cpp
#include <string>
string s1 = "Hello";
```

Example (C++ string):

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

int main() {
    string str1 = "Hello";
    string str2 = "World";

    cout << "Length: " << str1.length() << endl;

    string result = str1 + " " + str2; // concatenation
    cout << "Concatenated: " << result << endl;

    cout << "Substring (0,5): " << result.substr(0,5) << endl;

    result.replace(6, 5, "C++"); // replace "World" with "C++"
    cout << "After replace: " << result << endl;

    cout << "Find 'C++': " << result.find("C++") << endl;

    return 0;
```

Output:

```
Length: 5
Concatenated: Hello World
Substring (0,5): Hello
After replace: Hello C++
Find 'C++': 6
```

Key Difference

- **C-style strings → Character arrays, use <cstring> functions.**
- **C++ string class → Easier, more powerful, object-oriented with built-in methods.**

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

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

OOP is a programming paradigm that organizes software around objects (real-world entities), rather than just functions and logic.
C++ is one of the most widely used OOP languages.

The main concepts of OOP are:

1. Class and Object

- Class → A blueprint or template that defines properties (data members) and behaviors (member functions).
- Object → A real instance of a class.

Example:

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

    void drive() {
        cout << brand << " is driving at " << speed << " km/h" << endl;
    }
};

int main() {
    Car c1;  // object
    c1.brand = "BMW";
    c1.speed = 120;
    c1.drive();
    return 0;
}
```

2. Encapsulation

- **Wrapping data and functions into a single unit (class).**
- Data is usually made **private**, and accessed via **public methods** (getters/setters).
- Provides **data hiding** and **security**.

Example:

```
class Student {
private:
    int marks;

public:
    void setMarks(int m) { marks = m; }  // setter
    int getMarks() { return marks; }     // getter
};
```

3. Abstraction

- **Hiding internal details** and showing only the necessary functionality.
- Achieved using **abstract classes** or **interfaces (pure virtual functions)** in C++.

Example:

```
class Shape {
public:
    virtual void draw() = 0;  // pure virtual function (abstract method)
};

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

4. Inheritance

- **One class acquires properties and behaviors of another class.**
- Promotes **code reusability**.

Types in C++:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Example:

```
class Animal {
public:
    void eat() { cout << "Eating..." << endl; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Barking..." << endl; }
};
```

5. Polymorphism

- **Polymorphism = Many forms.**
- Allows same function name or operator to have different meanings.

Types:

1. **Compile-time (Static) Polymorphism** → Function overloading, Operator overloading.
2. **Runtime (Dynamic) Polymorphism** → Function overriding (using virtual functions).

Example (Overriding):

```cpp
class Animal {
public:
    virtual void sound() {
        cout << "Animal makes sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Dog barks" << endl;
    }
};
```

(20). What are classes and objects in C++? Provide an example.

Ans: **Classes and Objects in C++**

Class

- A **class** is a **user-defined data type** in C++.
- It acts as a **blueprint/template** for creating objects.
- It groups together **data (variables)** and **functions (methods)** into a single unit.

Syntax:

```cpp
class ClassName {
    // Access specifiers: public, private, protected
    data_members;   // variables
    member_functions; // methods
};
```

Object

- An **object** is an **instance of a class**.
- When a class is defined, no memory is allocated.
- When an object is created, memory is allocated for the data members.

Syntax:

```
ClassName objectName;
```

Example: Class and Object in C++

```
// Define a class
class Car {
public:  // access specifier
    string brand;
    int speed;

    // Member function
    void drive() {
        cout << brand << " is driving at " << speed << " km/h" << endl;
    }
};

int main() {
    // Create objects of the class
    Car car1;
    Car car2;

    // Assign values to object members
    car1.brand = "BMW";
    car1.speed = 120;

    car2.brand = "Tesla";
    car2.speed = 150;

    // Call member functions
    car1.drive();
    car2.drive();

    return 0;
```

Output:

```
BMW is driving at 120 km/h
Tesla is driving at 150 km/h
```

Class = Blueprint (like a design of a car).

Object = Real instance (like an actual BMW or Tesla car).

(21). What is inheritance in C++? Explain with an example.

Ans: **Inheritance in C++**

Definition

Inheritance is an **OOP (Object-Oriented Programming)** feature in C++ that allows a **class (derived/child)** to acquire the **properties and behaviors** of another class (**base/parent**).

It helps in:

- **Code reusability** (avoid rewriting the same code).
- **Establishing relationships** between classes.
- **Polymorphism** (runtime behavior).

Syntax:

```
class DerivedClass : accessSpecifier BaseClass {
    // new members of derived class
};
```

- accessSpecifier → public, protected, or private.

Example: Single Inheritance:

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

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

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

int main() {
    Dog d1;

    // Inherited method
    d1.eat();    // from Animal
    // Derived class method
    d1.bark();   // from Dog

    return 0;
}
```

(22). What is encapsulation in C++? How is it achieved in classes?

Ans: **Encapsulation in C++:**

Definition

- **Encapsulation** is one of the fundamental concepts of **Object-Oriented Programming (OOP)**.
- It means **binding data (variables) and methods (functions) into a single unit (class)**.
- It also means **restricting direct access to data** and controlling it through **public methods** (getters/setters).

In simple words:
Encapsulation = **Data Hiding + Controlled Access**

How is Encapsulation Achieved in Classes?

1. **Use of Access Specifiers:**
    - **private** → Data members are hidden from outside.
    - **public** → Functions (methods) provide controlled access.
2. **Getters and Setters:**
    - Public functions that allow reading/writing private data safely.

Example: Encapsulation in C++:

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

class Student {
private:
    int rollNumber;   // private data (hidden)
    int marks;

public:
    // Setter methods
    void setRollNumber(int r) {
        rollNumber = r;
    }

    void setMarks(int m) {
        if (m >= 0 && m <= 100)   // validation (controlled access)
            marks = m;
        else
            cout << "Invalid marks!" << endl;
    }

    // Getter methods
    int getRollNumber() {
        return rollNumber;
    }

    int getMarks() {
        return marks;
    }
};

int main() {
    Student s1;

    // Access data only via public methods
    s1.setRollNumber(101);
    s1.setMarks(95);

    cout << "Roll Number: " << s1.getRollNumber() << endl;
    cout << "Marks: " << s1.getMarks() << endl;

    return 0;
}
```

Output:

```
Roll Number: 101
Marks: 95
```

Encapsulation = Wrapping data + functions inside a class.

Protects data from **unauthorized access**.

Achieved using **private variables + public methods**.

Allows adding **validation and restrictions** while accessing data.