

# Modul 3

## **(1) - Key Differences between Procedural Programming and Object-Oriented Programming (OOP):**

### **1. Approach:**

Procedural Programming follows a top-down approach.

OOP follows a bottom-up approach.

### **2. Focus:**

Procedural Programming focuses on functions and procedures.

OOP focuses on objects that combine data and behavior.

### **3. Data Security:**

In Procedural Programming, data is usually global and less secure.

In OOP, data is hidden and protected using access modifiers (encapsulation).

### **4. Code Reusability:**

Procedural Programming has limited code reusability.

OOP supports reusability using inheritance.

## **5. Modularity:**

Procedural code is divided into functions.

OOP code is divided into classes and objects.

## **6. Example Languages:**

Procedural: C, Pascal

OOP: C++, Java, Python (OOP-based)

## **2. Main Advantages of OOP over POP:**

### **1. Encapsulation:**

It hides internal details and protects data using classes and objects.

### **2. Inheritance:**

Enables reuse of existing code by creating new classes from old ones.

### **3. Polymorphism:**

Allows functions or operators to behave differently based on input or context.

### **4. Abstraction:**

Only essential features are shown to the user; internal details are hidden.

### 5. **Modularity:**

Programs are divided into small parts (objects), which makes debugging and maintenance easier.

### 6. **Maintainability:**

OOP programs are easier to maintain and extend due to their organized structure.

## 3. **Steps Involved in Setting Up a C++ Development Environment:**

### 1. **Install a C++ Compiler:**

Example: g++ (MinGW for Windows, GCC for Linux)

### 2. **Choose an Editor or IDE:**

Lightweight: Visual Studio Code, Sublime Text

Full IDE: Code::Blocks, Dev C++, Visual Studio

### 3. **Write C++ Code:**

Save your file with .cpp extension (e.g., program.cpp)

### 4. **Compile the Code:**

Using terminal or command prompt:

## 4. Main Input/Output Operations in C++ (with Examples):

### Input - > cin

Used to take input from the user.

```
#include <iostream>

using namespace std;

int main() {

    int age;

    cout << "Enter your age: ";

    cin >> age;

    cout << "You are " << age << " years old.";

    return 0;

}
```

## (2) - 1. Different Data Types in C++

### (a). Integer (int)

Used to store whole numbers.

Example: `int age = 20;`

### (b). Floating-point (float)

Used to store decimal numbers (single precision).

Example: `float pi = 3.14;`

### (c). Double (double)

Stores decimal numbers with more precision (double precision).

Example: `double distance = 123.456789;`

### (d). Character (char)

Stores a single character.

Example: `char grade = 'A';`

**(e). Boolean (bool)**

Stores true or false values.

Example: `bool passed = true;`

**(f). Void (void)**

Used for functions that return no value.

Example: `void show();`

**(g). Derived types**

Includes arrays, pointers, and functions.

Example: `int marks[5];, int *ptr;`

**(h). User-defined types**

Includes structures, unions, and classes.

Example:

```
struct Student {  
  
    int roll;  
  
    char name[20];  
  
};
```

## **2. Implicit vs Explicit Type Conversion in C++**

### **Implicit Conversion (Type Promotion):**

- Automatically done by the compiler.
- Converts smaller data types to larger ones.

Example:

```
int a = 5;
```

```
float b = a;
```

### **3. Types of Operators in C++**

#### **(a). Arithmetic Operators**

Used for mathematical operations.

Example: +, -, \*, /, %

```
int sum = a + b;
```

#### **(b). Relational Operators**

Used to compare values.

Example: ==, !=, >, <, >=, <=

```
if (a > b)
```

#### **(c). Logical Operators**

Used for logical conditions.

Example: &&, ||, !

```
if (a > 0 && b > 0)
```

#### **(d). Assignment Operators**

Used to assign values.

Example: =, +=, -=, \*=, /=

```
a += 10;
```

#### **(e). Increment/Decrement Operators**

Increase or decrease value by 1.

Example: ++, --

```
I++;
```

#### **(f). Bitwise Operators**

Used for bit-level operations.

Example: `&, |, ^, ~, <<, >>`

#### **(g). Ternary Operator**

Used for shorthand if-else.

Syntax: `condition ? value1 : value2;`

```
int max = (a > b) ? a : b;
```

### **4. Constants and Literals in C++**

#### **Constants:**

- Values that do not change during program execution.
- Declared using `const` keyword.

```
const int PI = 3.14;
```

#### **Literals:**

- Fixed values used directly in the program.
- Examples:

Integer literal: `10`

Float literal: `3.14`

Char literal: `'A'`

String literal: `"Hello"`

Purpose of constants and literals:

- Make the program more readable and reliable.
- Prevent accidental changes to important values.

### **(3) - 1. Conditional Statements in C++**

**Conditional statements** allow a program to make decisions based on conditions. The main types are:

**if-else statement:**

Used to execute one block of code if a condition is true, and another if it's false.

**Syntax:**

```
int age = 18;

if (age >= 18) {
    cout << "Eligible to vote";
} else {
    cout << "Not eligible";
}

int day = 2;

switch (day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    default: cout << "Other day";
}
```



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

- **for loop:** Used when the number of iterations is known. It has initialization, condition, and increment/decrement in a single line.
- **while loop:** Used when the condition is checked before entering the loop. Executes only if the condition is true.
- **do-while loop:** Executes the loop body once before checking the condition. It runs at least one time.

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

**break statement:** Immediately exits the loop, even if the condition is still true. It is used to stop the loop early.

**continue statement:** Skips the current iteration and continues with the next iteration of the loop. It is used to skip specific values or conditions.

```
for (int i = 1; i <= 5; i++) {  
    if (i == 3) break;  
    cout << i;  
}
```

#### 4. Explain nested control structures with an example.

Nested control structures occur when one control structure (like if, for, while, switch) is placed inside another. This allows for more complex decision-making and repeated actions. It is commonly used when there are multiple conditions or loops working together.

```
for (int i = 1; i <= 3; i++) {  
    if (i % 2 == 0) {  
        cout << i << " is even";  
    } else {  
        cout << i << " is odd";  
    }  
}
```

## (4) -

### 1. What is a function in C++?

A **function** in C++ is a block of code that performs a specific task. It allows code reusability and makes programs modular.

#### ◆ Function Declaration (Prototype):

Tells the compiler about the function name, return type, and parameters before it's used.

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

#### Function Definition:

Contains the actual body/code of the function.

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

#### Function Call:

```
int result = add(5, 10);
```

## 2. What is the scope of variables in C++?

**Scope** means where a variable is accessible in the program.

### **Local Variable:**

- Declared inside a function or block.
- Can be used only within that block.

```
void show() {  
    int a = 10; // local variable  
}
```

### **Global Variable:**

- Declared outside all functions.
- Can be used anywhere in the program.

```
int a = 10;  
  
void show() {  
    cout << a;  
}
```

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

**Recursion** is when a function calls itself to solve a smaller part of the problem.

**Example:** Factorial using recursion

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}
```

**Explanation:**

This function calls itself until n becomes 0, then multiplies all values back up.

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

A **function prototype** is the declaration of a function that tells the compiler:

- Function name
- Return type
- Number and type of parameters

```
int sum(int, int);
```

**Why are prototypes used?**

- So the compiler knows about the function before it is used.
- Allows the function to be defined after `main()`.

Without a prototype, calling a function before defining it would cause a compiler error.

## (5) - 1. What are arrays in C++?

An **array** is a collection of elements of the same data type stored at contiguous memory locations. Arrays allow storing multiple values using a single variable name with an index.

### ♦ Difference Between Single-Dimensional and Multi-Dimensional Arrays:

Type	Description	Example
<b>1D Array</b>	Stores elements in a single row (linear)	<code>int num[5];</code>
<b>2D Array</b>	Stores elements in rows and columns (matrix)	<code>int matrix[3][3];</code>
<b>Multi-Dimensional</b>	More than 2 dimensions (rarely used)	<code>int cube[2][2][2] ;</code>

## 2. Explain String Handling in C++ with Examples

C++ supports strings in two ways:

### Using Character Arrays:

- Strings are handled as arrays of characters, ending with a null character (`'\0'`).

### Example:

```
#include <string>
```

```
string name = "Shubham";
```

### 3. How are arrays initialized in C++?

- ♦ One-Dimensional Array (1D):

```
int arr1[5] = {1, 2, 3, 4, 5};
```

Or

```
int arr2[] = {10, 20, 30};
```

Two-Dimensional Array (2D):

```
int matrix[2][3] = {
```

```
    {1, 2, 3},
```

```
    {4, 5, 6}
```

```
};
```

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

In C++, string operations can be performed using the **string class** provided in the **<string>** header file. This class offers several built-in functions to manipulate and handle strings efficiently.

## **Common String Operations and Their Use:**

### **1. length()**

Returns the total number of characters in the string.

Used to find the size of a string.

### **2. append()**

Adds one string to the end of another.

It is useful for combining strings.

### **3. substr(position, length)**

Returns a part (substring) of the string starting from a given position and up to the given length.

### **4. compare()**

Compares two strings.

Returns 0 if strings are equal, a positive or negative value otherwise.

### **5. find()**

Searches for a substring inside the string and returns its index.

If not found, it returns -1.

## 6. **replace(position, length, new\_string)**

Replaces a part of the string with a new substring from a given position.

## 7. **Concatenation using + operator**

Joins two or more strings using the + operator.

# **(6) - 1. Explain the key concepts of Object-Oriented Programming (OOP)**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects** and **classes**. The key concepts of OOP are:

### **1. Class and Object**

A class is a blueprint. An object is an instance of a class.

### **2. Encapsulation**

Wrapping data and code together into a single unit (class). It hides internal details from the user.

### **3. Inheritance**

One class can inherit properties and behaviors from another class. Promotes code reusability.

### **4. Polymorphism**

One function or operator can work in different ways (like overloading and overriding).

### **5. Abstraction**

Showing only the essential features and hiding the complex details from the user.



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

### **Class**

A class is a user-defined data type that contains variables (data) and functions (behavior).

### **Object**

An object is an instance of a class. It is used to access the class members.

**Example:**

```
class Car {  
    public:  
        string brand;  
        void drive() {  
            cout << "Driving...";  
        }  
};
```

```
Car myCar; // object creation
```

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

Inheritance is the process by which one class (child) can acquire the properties and methods of another class (parent). It allows code reuse and makes programs easier to maintain.

**Types of Inheritance in C++:**

- **Single Inheritance**

- **Multiple Inheritance**
- **Multilevel Inheritance**
- **Hierarchical Inheritance**

**Example:**

```
class Animal {
```

```
    public:
```

```
        void eat() {
```

```
            cout << "Eating...";
```

```
        }
```

```
};
```

```
class Dog : public Animal {
```

```
    public:
```

```
        void bark() {
```

```
            cout << "Barking...";
```

```
        }
```

```
};
```

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

**Encapsulation** means combining data and functions into a single unit (a class) and restricting access to some parts of the object.

It is achieved using:

- **Access specifiers:** private, protected, and public.
- By keeping variables **private** and providing **public** getter and setter functions.

##### **Benefits of Encapsulation:**

- Data protection
- Better code structure
- Prevents direct access to internal data

##### **Example:**

```
class Student {  
  
    private:  
  
        int marks;  
  
    public:  
  
        void setMarks(int m) {  
            marks = m;  
        }  
  
        int getMarks() {  
            return marks;  
        }  
};
```

