# Module 3: Introduction to OOPS Programming

## 1. Key Differences Between Procedural Programming (POP) and Object-Oriented Programming (OOP)

| Feature | Procedural Programming (POP) | Object-Oriented Programming (OOP) |
|---|---|---|
| Approach | Focuses on functions and procedures to operate on data. | Focuses on objects which combine data and functions. |
| Data Handling | Data is usually separate from functions. | Data and functions are encapsulated together in objects. |
| Reusability | Limited code reusability; mostly through functions. | High code reusability through inheritance and polymorphism. |
| Security | Less secure; data can be accessed from any part of the program. | More secure; encapsulation allows controlled access to data. |
| Maintenance | Difficult to maintain large programs because code and data are separate. | Easier to maintain large programs because objects are modular and self-contained. |
| Examples | C, early versions of Pascal | C++, Java, Python (supports OOP) |

## 2. Main Advantages of OOP over POP

1. **Modularity:** Code is organized into objects, making it clean and structured.
2. **Reusability:** Objects and classes can be reused in different programs.
3. **Data Security:** Data is hidden and safe from unauthorized access (Encapsulation).

4. **Maintainability:** Easy to update and modify specific parts without affecting the whole program.
5. **Polymorphism:** The same action can be performed in different ways.
6. **Real-world Modeling:** It models the real world better, making it easier to understand.

## 3. Steps to Set Up a C++ Development Environment

1. **Install a Compiler** (like MinGW for Windows, g++ for Linux, or Xcode for Mac).
2. **Install an IDE** (like Code::Blocks, Visual Studio, or CLion).
3. **Configure Compiler Path** in the IDE's settings.
4. **Write a Test Program** to check if everything works.
5. **Compile and Run** the program.

**Example Test Program:**

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

int main() {
    cout << "C++ setup is successful!" << endl;
    return 0;
}
```

## 4. Main Input/Output Operations in C++

- **Input:** Use `cin` to get data from the user.
- **Output:** Use `cout` to display data on the screen.

**Example:**

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

int main() {
    int age;
    string name;

    cout << "Enter your name: ";
    cin >> name;
    cout << "Enter your age: ";
    cin >> age;
```

```
    cout << "Hello " << name << "! You are " << age << " years old." << endl;
    return 0;
}
```

## 5. Data Types in C++

- **Basic Types:** `int`, `float`, `double`, `char`, `bool`
- **Derived Types:** arrays, pointers
- **User-defined Types:** `struct`, `class`, `enum`

**Example:**

```
int age = 25;
float pi = 3.14;
double salary = 5000.75;
char grade = 'A';
bool isPassed = true;
```

## 6. Implicit vs Explicit Type Conversion

- **Implicit:** Done automatically by the compiler.

```
int a = 10;
double b = a; // Compiler converts 'a' to double automatically
```

- **Explicit:** Done manually by the programmer (also called type casting).

```
double x = 9.5;
int y = (int)x; // Programmer forces conversion to int
```

## 7. Types of Operators in C++

- **Arithmetic:** +, -, *, /, %
- **Relational:** ==, !=, >, <, >=, <=
- **Logical:** && (AND), || (OR), ! (NOT)
- **Assignment:** =, +=, -=, *=, /=, %=
- **Increment/Decrement:** ++, --
- **Bitwise:** &, |, ^, <<, >>, ~
- **Conditional (Ternary):** ? :

**Example:**

```
int a = 5, b = 2;
cout << (a + b); // 7 (Arithmetic)
cout << (a > b); // 1 (true) (Relational)
cout << (a && b); // 1 (true) (Logical)
```

## 8. Constants and Literals

- **Constants:** A variable whose value cannot be changed once set.

```
const float PI = 3.14;
```

- **Literals:** The actual fixed values themselves.

```
10        // integer literal
3.14      // floating-point literal
'A'       // character literal
"Hello"   // string literal
```

## 9. Conditional Statements

- **If-Else:** For decisions based on conditions.

```
if (num > 0) {
    cout << "Positive";
} else {
    cout << "Non-positive";
}
```

- **Switch:** For selecting one of many code blocks to execute.

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

## 10. For, While, and Do-While Loops

- **For Loop:** Use when you know how many times to repeat.

```
for (int i = 0; i < 5; i++) {
```

```
    cout << i;
}
```

- **While Loop:** Repeats as long as the condition is true. Checks condition first.

```
while (i < 5) {
    cout << i;
    i++;
}
```

- **Do-While Loop:** Repeats as long as the condition is true. Executes the code block at least once.

```
do {
    cout << i;
    i++;
} while (i < 5);
```

## 11. Break and Continue Statements

- **Break:** Immediately exits the loop.
- **Continue:** Skips the rest of the current iteration and moves to the next one.

**Example:**

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break; // Loop stops when i is 3
    }
    cout << i; // Output will be: 0 1 2
}
```

## 12. Nested Control Structures

Putting loops inside loops or conditionals inside loops.

**Example:**

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

```
    }
}
```

## 13. Functions in C++

- **Declaration:** Telling the compiler about a function's name, return type, and parameters.
  `int add(int, int);`
- **Definition:** The actual body of the function.

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

- **Call:** Using the function to perform a task.
  `int sum = add(5, 3); // sum becomes 8`

## 14. Scope of Variables

- **Local Variable:** Declared inside a function. Can only be used there.
- **Global Variable:** Declared outside all functions. Can be used anywhere in the program.

**Example:**

```
int globalVar; // Global Variable

void demo() {
    int localVar; // Local Variable
}
```

## 15. Recursion in C++

A function that calls itself to solve a problem.

**Example (Factorial):**

```
int factorial(int n) {
    if (n == 0) { // Base case to stop recursion
        return 1;
    }
    return n * factorial(n - 1); // Function calls itself
}
```

## 16. Function Prototypes

A declaration of a function that tells the compiler about the function's name, return type, and parameters *before* the function is defined. This prevents errors and allows you to define the function later.

```
// Function Prototype
int multiply(int, int);

int main() {
    int result = multiply(5, 4); // Works because of the prototype
    return 0;
}

// Function Definition
int multiply(int a, int b) {
    return a * b;
}
```

## 17. Arrays in C++

A collection of elements of the same type stored in contiguous memory locations.

- **1D Array:** A list of elements.
  `int arr[5] = {1, 2, 3, 4, 5};`
- **2D Array:** A table of elements (rows and columns).
  `int mat[2][2] = {{1, 2}, {3, 4}};`

## 18. String Handling

In C++, you can use the `string` class from the `<string>` library for easy text handling.

**Example:**

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

int main() {
    string name = "Karan";
    cout << name.length(); // Outputs 5
    cout << name + " Patel"; // Outputs "Karan Patel"
```

```
    return 0;
}
```

## 19. Array Initialization

- **1D Array Initialization:**
  `int arr[3] = {1, 2, 3};`
- **2D Array Initialization:**
  `int mat[2][2] = {{1, 2}, {3, 4}};`

## 20. String Operations

- **Concatenation:** `s1 + s2` (Joining two strings)
- **Length:** `s.length()` (Number of characters)
- **Compare:** `s1 == s2` (Check if two strings are identical)
- **Substring:** `s.substr(0, 3)` (Get part of a string, starting at index 0, taking 3 characters)

## 21. Key Concepts of OOP

1. **Class & Object:** A class is a blueprint. An object is an instance of that class.
2. **Encapsulation:** Bundling data and methods together, hiding internal details.
3. **Inheritance:** A class can inherit properties and methods from another class.
4. **Polymorphism:** The ability to present the same interface for different underlying forms.
5. **Abstraction:** Hiding complex implementation details and showing only essential features.

## 22. Classes and Objects

A **class** is a user-defined data type. An **object** is a variable of that class type.

**Example:**

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

// Class (Blueprint)
class Car {
public:
    string color;
    void display() {
```

```
        cout << "Car color: " << color;
    }
};

int main() {
    Car myCar; // Object (Instance)
    myCar.color = "Red";
    myCar.display(); // Outputs: Car color: Red
    return 0;
}
```

## 23. Inheritance

Allows a new class (derived class) to inherit attributes and methods from an existing class (base class).

**Example:**

```
// Base Class (Parent)
class Vehicle {
public:
    int wheels;
};

// Derived Class (Child)
class Bike : public Vehicle {
public:
    Bike() {
        wheels = 2; // Bike inherits 'wheels' from Vehicle
    }
};

int main() {
    Bike b;
    cout << b.wheels; // Outputs: 2
    return 0;
}
```

## 24. Encapsulation

The concept of bundling data (variables) and methods (functions) that work on the data into a single unit (a class) and restricting direct access to some of the object's components. This is done using `private` and `public` access specifiers.

**Example:**

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

class Bank {
private: // Data is hidden and private
    int balance;

public: // Functions to control access are public
    void setBalance(int b) {
        balance = b;
    }
    int getBalance() {
        return balance;
    }
};

int main() {
    Bank account;
    account.setBalance(1000);
    // account.balance = 1000; // ERROR: 'balance' is private
    cout << account.getBalance(); // This is the safe way
    return 0;
}
```