
1010101010
0101010101
1010101010
0101010101
Notes
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
Good Notes
Better Understanding
Better Results


1010101010
0101010101
1010101010
0101010101
Unit-I
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

OOP PARADIGM: BASIC CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code to manipulate that data.

BASIC CONCEPTS OF OOP

① CLASS

- A class is a blueprint or template used to create objects.
- It defines the properties (data) and behaviors (methods) that objects of this type will have.

Example: Class Student
Data : name, rollNo, marks
Methods : display(), calculate()

② OBJECT

- An object is an instance of a class.
- It is a real-world entity that has state (data) and behavior (methods).
- Memory is allocated when an object is created.

Example:

s1	s2
name = Amit	name = Neha
rollNo = 101	rollNo = 102
marks = 85	marks = 92

③ ENCAPSULATION

- Encapsulation is the process of binding data (variables) and methods (functions) together in a single unit (class) and restricting direct access to some of an object's components.
- It protects data from unintended access or modification.

Example:

Private data → hidden from outside
Public methods → access & modify private data

④ ABSTRACTION

- Abstraction is the process of hiding unnecessary implementation details and showing only the essential features of an object.
- It helps to reduce complexity.

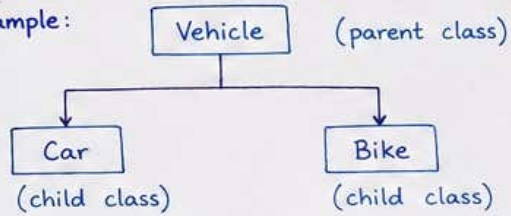
Example:

Car
We drive the car (use features)
- We don't need to know how engine works internally.

⑤ INHERITANCE

- Inheritance is the mechanism in which one class acquires the properties and behaviors of another class.
- It promotes code reusability.

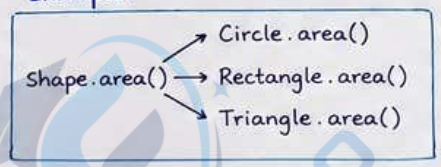
Example:



⑥ POLYMORPHISM

- Polymorphism means "many forms".
- It allows objects of different classes to be treated as objects of a common super class.
- It is achieved through method overloading and method overriding.

Example:



⑦ DYNAMIC BINDING

- Dynamic binding (late binding) is the process in which a call to an overridden method is resolved at run time rather than at compile time.
- It is a key feature that supports polymorphism.

Example:

```
Shape s;
s = new Circle();
s.draw(); // calls Circle's
          draw() method
          at run time
```

Summary: OOP makes programs easier to design, understand, maintain and extend by using concepts like Class, Object, Encapsulation, Abstraction, Inheritance, Polymorphism and Dynamic Binding.

OOP PARADIGM: FEATURES AND ADVANTAGES

Object-Oriented Programming (OOP) provides several key features that help in building robust, flexible, reusable and maintainable software.

FEATURES OF OOP

Feature	Description
Class	• A class is a blueprint or template used to create objects. It defines data and methods that objects will have.
Object	• An object is an instance of a class. It has state (data) and behavior (methods).
Encapsulation	• Binding data and methods together in a class and restricting direct access to some of the object's components.
Abstraction	• Hiding unnecessary implementation details and showing only the essential features of an object.
Inheritance	• Mechanism in which one class inherits the properties and behaviors of another class. Promotes code reusability.
Polymorphism	• Ability to take many forms. The same method or operator behaves differently for different objects.
Dynamic Binding	• Also called late binding. The method to be executed is decided at run time.

ADVANTAGES OF OOP

- ☆ **Modularity** : Program is divided into classes and objects. Each class handles a specific task, making the program well organized.
- ☆ **Reusability** : Classes and inheritance allow code reusability. Existing classes can be used to create new classes.
- ★ **Maintainability** : Easy to modify and maintain as the code is well structured in classes and methods.
- ☆ **Scalability** : New features can be added by creating new classes without affecting existing code. Suitable for large projects.
- ☆ **Data Security** : Encapsulation and data hiding protect data from unauthorized access and misuse.
- ☆ **Flexibility** : Polymorphism and dynamic binding make the program flexible. New behaviors can be added without changing existing code.
- ★ **Real-world Modeling** : OOP models real-world entities like objects, classes, inheritance etc., making the program easy to understand.

Summary : OOP combines the above features to make software easier to develop, reuse, maintain and extend. It leads to better productivity, reduced development time and high quality software.

APPLICATIONS OF OOP

Object-Oriented Programming (OOP) is used in a wide range of real-world applications. Its concepts like **class**, **object**, **inheritance**, **polymorphism**, **encapsulation** and **abstraction** help to build robust, scalable and maintainable software.

Applications of OOP

1. Real-time Systems

- OOP helps in modeling real-world entities and their behaviors.

Example: Traffic control systems, ATM systems, Railway reservation systems.

2. Simulation and Modeling

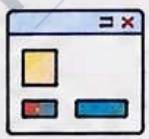
- OOP allows creating models of complex systems that can be easily updated and maintained.

Example: Flight simulators, Weather systems, Scientific simulations.

3. Game Development

- OOP is widely used to develop interactive games with rich graphics and reusability.

Example: 2D/3D games, Mobile games, Online multiplayer games.



4. GUI Applications

- OOP helps in designing user interfaces using objects like buttons, menus, text boxes.

Example: Desktop applications, Media players, Image editors.

5. Database Applications

- OOP helps in designing secure, robust and easy-to-maintain database applications.

Example: Library management systems, Banking systems, ERP systems.

6. Web Applications

- OOP improves code reusability and maintainability in web development.

Example: E-commerce sites, Social media platforms, Content management systems.

7. Enterprise Applications

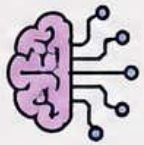
- OOP is used to develop large-scale, reliable and distributed applications.

Example: HR systems, Payroll systems, Inventory management systems.

8. Mobile Applications

- OOP is used in Android and iOS app development for better code organization and reusability.

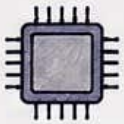
Example: Mobile banking apps, Shopping apps, Health and fitness apps.



9. Artificial Intelligence and Machine Learning

- OOP helps in creating modular and reusable code for AI/ML models and data processing.

Example: Chatbots, Recommendation systems, Image recognition systems.



10. Embedded Systems

- OOP is used to design embedded software that controls real-world devices.

Example: Home appliances, Automotive systems, Medical devices.

Key Takeaway : OOP is used everywhere – from small applications to large enterprise systems. It makes software more **organized**, **reusable**, **scalable** and **easy to maintain**.

STRUCTURED Vs OOP

Structured Programming is a procedure-oriented approach in which a program is divided into functions.
Object-Oriented Programming (OOP) is an object-based approach in which a program is divided into objects combine data and behavior.

Aspect	Structured Programming	OOP (Object-Oriented Programming)
1. Basic Approach	• Procedure-oriented approach. • Program is divided into functions.	• Object-oriented approach. • Program is divided into objects.
2. Focus	• Focus on logic or procedures.	• Focus on data (objects) and behavior.
3. Program Structure	• Top-down approach. • Program is a collection of functions.	• Bottom-up approach. • Program is a collection of interacting objects.
4. Data and Functions	• Data and functions are separate. • Data is usually global and accessible throughout the program.	• Data and functions are combined in objects. • Data is hidden and accessed only through methods.
5. Data Security	• Less secure. Data can be accessed by any function.	• More secure. Data is hidden using encapsulation.
6. Reusability	• Limited reusability. Functions can be reused with some limitations.	• High reusability. Classes and objects can be easily reused.
7. Modularity	• Program is divided into functions (modules).	• Program is divided into classes and objects. More modular.
8. Maintainability	• Difficult to maintain as the program grows.	• Easier to maintain due to data hiding, modularity and reusability.
9. Extensibility	• Difficult to extend the program.	• Easy to extend by adding new classes/objects.
10. Examples of Languages	• C, Pascal, Fortran, Cobol etc.	• C++, Java, Python, C#, PHP, Ruby etc.
11. Real-World Modeling	• Does not model real-world entities directly.	• Models real-world entities naturally using objects.
12. Suitability	• Suitable for small and simple programs.	• Suitable for large, complex and long-term projects.

Example

Structured Approach

Program to calculate area of rectangle.

Functions:
read_length(), read_breadth(), calculate_area(), display()

Data (length, breadth) is passed from one function to another.

OOP Approach

Create a class 'Rectangle' with:
Data: length, breadth
Methods: read(), calculateArea(), display()

Create objects of Rectangle class and call methods.

Summary:
→ Structured Programming is procedure-oriented and function-based.
→ OOP is object-oriented and class/object-based.

OOP is better for:
✓ Large and complex projects
✓ Code reusability
✓ Easy maintenance
✓ Real-world modeling

TRENDING OOP LANGUAGES

Object-Oriented Programming (OOP) languages organize code using classes and objects. These languages are widely used in industry for building scalable, maintainable and robust applications.

No.	Language	Key Features (OOP Support)	Why it's Trending	Common Applications
1.	Java	• Pure OOP language • Platform independent (JVM) • Strong security and robustness	• Widely used in enterprise applications • Large community and ecosystem • Constant updates and long-term support	• Enterprise apps • Android apps • Banking systems
2.	Python	• OOP with simple syntax • Supports multiple paradigms • Extensive standard libraries	• Easy to learn and use • High demand in AI, ML, Data Science • Versatile and beginner friendly	• Web development • AI / ML, Data Science • Automation, Scripting
3.	C++	• Supports OOP and generic programming • High performance • Memory management control	• Used in system/software development • Game development, embedded systems • Foundation for modern languages	• Games • Operating systems • Embedded systems
4.	C#	• Pure OOP language • Simple, modern and type-safe • Strong .NET support	• Used for Windows & web applications • Unity game engine support • Regular updates by Microsoft	• Desktop apps • Web apps (ASP.NET) • Game development (Unity)
5.	JavaScript (ES6+)	• Prototype-based OOP • Supports classes (ES6+) • Widely used in web	• Backbone of modern web development • Works on both frontend and backend (Node.js) • Huge community and frameworks	• Web development • Server-side apps (Node.js) • Web and mobile apps
6.	TypeScript	• OOP with classes and interfaces • Strongly typed superset of JS • Better tooling and maintainability	• Increasing adoption in large projects • Improves code quality and scalability • Supported by Microsoft	• Large scale web apps • Frontend frameworks (Angular) • Enterprise applications
7.	PHP	• OOP support • Easy to learn • Great for web development	• Powers most of the websites • Continuous performance improvements • Strong ecosystem (Laravel, etc.)	• Web development • Content management systems • E-commerce websites
8.	Ruby	• Pure OOP language • Simple and elegant syntax • Convention over configuration	• Developer productivity • Popular framework Ruby on Rails • Clean and readable code	• Web development • Startups and rapid prototyping • SaaS applications

Note :
The choice of language depends on the project requirements, performance needs and developer expertise.

Key Takeaway : All these languages support OOP principles like encapsulation, inheritance, polymorphism and abstraction, which help in building modular, reusable and maintainable software.

OOP CONCEPTS – DATA ABSTRACTION AND ENCAPSULATION

Abstraction and Encapsulation are two important OOP concepts that help in building secure, flexible, and easy to maintain applications.

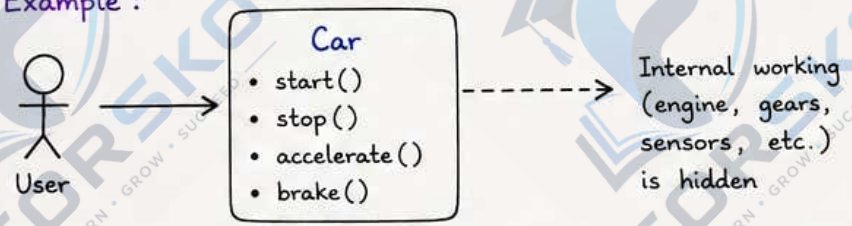
1. DATA ABSTRACTION

Abstraction is the process of hiding implementation details and showing only the essential features (functionality) to the user.

Key Points :

- Hides unnecessary information.
- User focuses on what the object does, not how it does.
- Achieved using abstract classes and interfaces.

Example :



The user knows what the car does, but not how it works.

Python Example (using abstract class) :

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

Shape is an abstract class. It shows only the essential feature (area), not how to calculate it.

2. DATA ENCAPSULATION

Encapsulation is the process of binding data (variables) and the methods (functions) that operate on the data into a single unit (class) and restricting direct access to some of the object's components.

Key Points :

- Protects data from direct access and modification.
- Data can be accessed only through public methods.
- Achieved using access specifiers (private, protected, public).

Example :

```
class BankAccount:
    def __init__(self, owner, balance):
        self.__owner = owner      # private variable
        self.__balance = balance  # private vartable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

Data is hidden (private).

Access to data is provided through methods.

Output / Usage :

```
acc = BankAccount("Ali", 1000)
acc.deposit(500)
print(acc.get_balance())  # Output: 1500
```

We cannot access __balance directly from outside the class. This protects the data.

Summary :

- ✓ Abstraction hides the complexity and shows only essential features.
- ✓ Encapsulation hides the data and provides controlled access through methods.

Together, they help in building programs that are :

- ✓ Secure
- ✓ Easy to maintain
- ✓ Reusable
- ✓ Scalable

CLASSES AND OBJECTS, DEFINING A CLASS

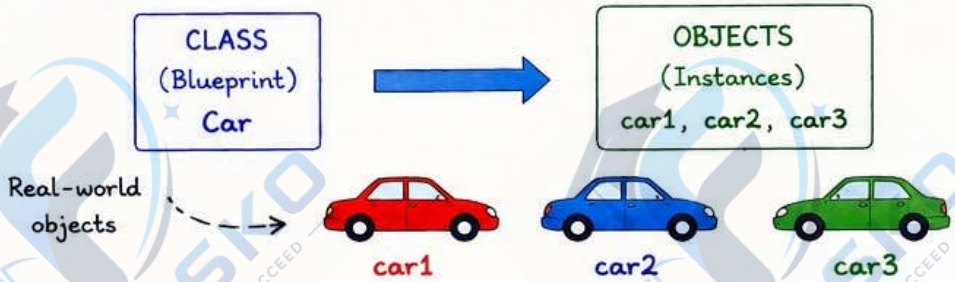
In OOP, a **class** is a **blueprint** or **template** for creating objects. An **object** is an **instance** of a class.

(1) CLASSES AND OBJECTS

→ A class defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.

→ An object is a real-world entity that has state (data) and behavior (functions).

Example :



(2) DEFINING A CLASS

A class is defined using the keyword **class**.

```
class ClassName:
    def __init__(self, parameters):
        # Constructor (optional)
        # attributes (data members)
    def method1(self):
        # method body
    def method2(self):
        # method body
        ...
```

- `class` keyword followed by class name
- → Constructor method. Called automatically when an object is created.
- → Attributes (variables) that belong to the class
- → Methods (functions) that define the behavior of objects

(3) EXAMPLE – CLASS AND OBJECTS

```
class Car:
    def __init__(self, brand, color, speed):
        self.brand = brand
        self.color = color
        self.speed = speed

    def display(self):
        print(f"Car[{self.brand}], Color[{self.color}],
              Speed[{self.speed}] km/h")
```

Creating objects

```
car1 = Car("Toyota", "Red", 60)
car2 = Car("Honda", "Blue", 80)
```

Accessing methods

```
car1.display()
car2.display()
```

Output :

```
Car[Toyota], Color[Red], Speed[60] km/h
Car[Honda], Color[Blue], Speed[80] km/h
```

(4) KEY POINTS

- A class is defined once, but can be used to create many objects.
- Each object has its own copy of attributes.
- Methods define the behavior of objects.
- **self** refers to the current object.

→ "self" is a reference to the current instance of the class.

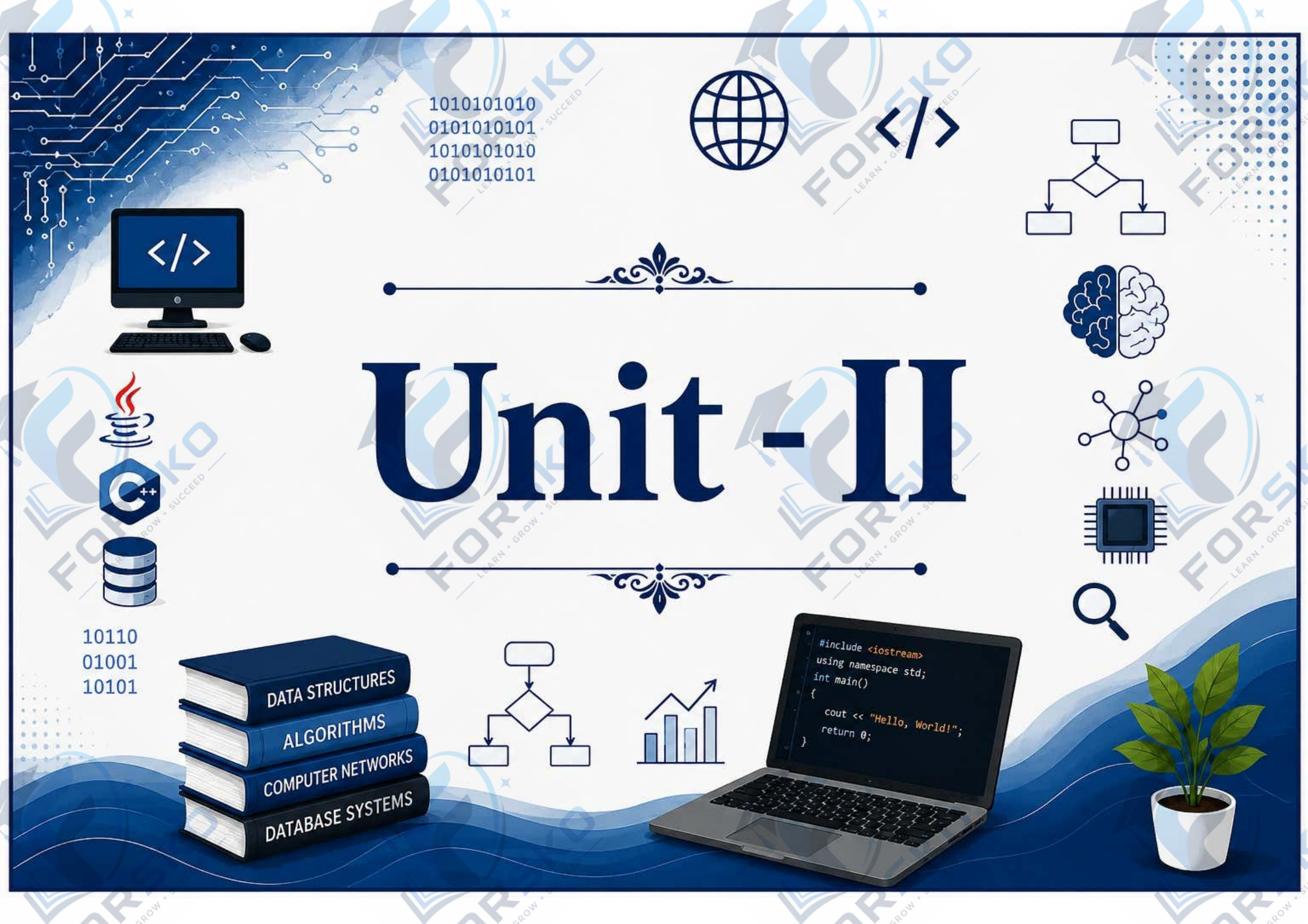
(5) REAL-WORLD ANALOGY

OOP Concept	Real-World Analogy
Class	Blueprint of a house
Object	Actual house built from the blueprint
Attributes	Features like color, rooms, size
Methods	Actions like open door, close window

Summary :

- ✓ A class is a blueprint that defines attributes and methods.
- ✓ An object is an instance of a class.
- ✓ We define a class using the keyword **class** and create objects from it.
- ✓ Objects have state (data) and behavior (methods).
- ✓ This is the foundation of **Object-Oriented Programming**.

Unit -II


1010101010
0101010101
1010101010
0101010101
</>
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Functions : Function Prototype

C++

1. What is a Function Prototype ?

A function prototype (or function declaration) tells the compiler about a function's name, return type and parameters before the actual function is used.

→ It does not contain the body of the function.

It only ends with a semicolon (;).

2. Why do we need a Function Prototype ?

- It allows the compiler to check the function call for correct number, type and order of arguments.
- It enables us to call a function before its definition.
- It makes the program modular and easier to read.
- It helps in separating function interface from implementation.

3. Syntax of Function Prototype

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

- return_type → Type of value returned
- function_name → Name of function
- parameter_list → Types of parameters

Examples :

```
1. int add(int, int);          // function with two
                               // integer parameters
2. float area(float);          // function with one
                               // float parameter
3. void display();             // function with no
                               // parameters
4. double power(double, int);  // function with two
                               // parameters
```

4. How it Works ?

When the compiler sees a function call, it checks the prototype to know :

① What is the return type ?
② How many arguments are expected ?
③ What are their data types ?

5. Example Program

```
// Function Prototype (Declaration)
int add(int, int);    // Prototype

int main() {
    int a = 10, b = 20, sum;
    sum = add(a, b);   // Function call
    cout << "Sum = " << sum << endl;
    return 0;
}

// Function Definition
int add(int x, int y) {
    int s = x + y;
    return s;
}
```

Tells the compiler about add()

Actual body of the function (Definition)

6. Important Points

- ☆ Prototype must end with a semicolon.
- ☆ Prototype and definition must have the same return type, name and parameters.
- ☆ Parameter names in prototype are optional, but types and order are required.
- ☆ A function can have multiple prototypes in different files, but only one definition.

Note :

The prototype is like a promise of what the function will do, while the definition is the actual implementation.

In Short → Function Prototype = Declaration of a function. : It helps the compiler understand the function before it is used.

Inline Function in C++

1. What is Inline Function ?

An inline function is a function in which the compiler is requested to expand the function code at the point of function call instead of performing a normal function call.

→ It is achieved using the keyword 'inline'.

2. Why use Inline Function ?

- Reduces function call overhead (no call/return).
- Increases execution speed for small functions.
- Saves time in pushing parameters and returning control.
- Best suited for small and frequently called functions.

3. How it Works ?

- ☆ Instead of jumping to the function, the compiler copies the function body at the call site.
- ☆ No jump back, so it saves time.

4. Syntax

```
inline return_type function_name(parameter_list)
{
    // function body
}
```

5. Important Points

- ☆ Inline is only a request to the compiler, not a command.
- ☆ The compiler may ignore the request if the function is large or complex.
- ☆ Best for small, frequently used functions (like getters, setters, simple calculations).
- ☆ Excessive use of inline may increase the size of the executable file.

6. When Not to use Inline ?

- × For large functions.
- × For recursive functions.
- × For functions with static variables.
- × When the function is called rarely.

7. Example Program

```
// Inline function example
#include <iostream>
using namespace std;

// Inline function
inline int square(int x) {    // function definition
    return x * x;
}

int main() {
    int a = 5, result;

    // Function call
    result = square(a);    // Inline expansion

    cout << "Square = " << result << endl;
    return 0;
}
```

Output :

```
Square = 25
```

Note :

The compiler replaces the call to square(a) with "return a * a;"

8. Advantages

- ✓ Faster execution.
- ✓ Reduced function call overhead.
- ✓ Improved performance for small functions.

9. Disadvantages

- × Increases code size.
- × Not suitable for large functions.
- × Compiler may ignore inline request.

In Short

Inline function is a request to the compiler to replace the function call with the function body at the call site. Use it wisely for small and frequently used functions.

DEFAULT ARGUMENT in C++

1. What is Default Argument ?

A default argument is a value provided in the function declaration. If an argument is not passed during the function call, the default value is used.

→ It makes the function call simpler and more flexible.

2. Why use Default Argument ?

- Reduces the number of arguments in function call.
- Makes the program easier to read and write.
- Increases code reusability and flexibility.
- Useful when some values are common or assumed.

3. Rules for Default Arguments

① Default arguments are given in function declaration, not in function definition.

② Default arguments are assigned from right to left.

③ Once a parameter has a default value, all parameters to its right must also have default values.

④ Default arguments are used only when the corresponding arguments are omitted in the function call.

4. Syntax

```
return_type function_name( data_type  p1 = value1,
                           data_type  p2 = value2,
                           ...        ,
                           data_type  pn = valuen );
```

5. Example Program

```
// Function Declaration with Default Arguments
int sum(int a, int b = 10, int c = 20)
{
    return a + b + c;
}
int main()
{
    cout << "sum(5)       = " << sum(5)       << endl;  // 5 + 10 + 20
    cout << "sum(5, 15)   = " << sum(5, 15)   << endl;  // 5 + 15 + 20
    cout << "sum(5, 15, 25) = " << sum(5, 15, 25) << endl;  // 5 + 15 + 25
    return 0;
}
```

6. How it Works ?

If we do not pass some arguments in the function call, the compiler uses the default values from right to left.

```
sum(5)       → a = 5,  b = 10 (default), c = 20 (default)
sum(5, 15)   → a = 5,  b = 15,           c = 20 (default)
sum(5, 15, 25) → a = 5,  b = 15,         c = 25
```

7. Important Points

- ★ Default arguments improve code readability.
- ★ They should be used when the default values are meaningful.
- ★ Overuse of default arguments can make the program confusing.
- ★ Default arguments are substituted by the compiler at compile time.

8. More Examples

```
// Function with two default arguments
void display(string name = "Guest", int age = 18)
{
    cout << "Name: " << name << ", Age: " << age << endl;
}

int main()
{
    display();              // Name: Guest, Age: 18
    display("Riya");        // Name: Riya, Age: 18
    display("Riya", 21);    // Name: Riya, Age: 21
    return 0;
}
```

9. Advantages

- Simplifies function calls.
- Enhances flexibility.
- Reduces function overloading in many cases.
- Improves code maintainability.

10. Disadvantages

- Excessive default arguments may hide important values.
- If default values change, it may affect the program logic.
- Not suitable for all types of functions.

In Short

Default Argument = A value provided in the function declaration that is used when the corresponding argument is not provided in the function call.

Function Overloading in C++

1. What is Function Overloading ?

Function overloading means having multiple functions with the same name but different parameters in the same scope.

→ The function to be called is decided by the compiler at compile time based on the number, type or order of arguments.

2. Key Points

- It is a type of compile time polymorphism.
- Functions must have the same name.
- They must differ in number, type or order of parameters.
- Return type alone cannot be used to overload a function.
- Increases readability and makes the code more flexible.

3. Syntax

```
return_type  functionName(parameter list);
return_type  functionName(different parameter list);
return_type  functionName(...);
```

4. How it Works ?

Function Call

↓

Compiler checks the arguments

↓

Matches with the best function signature

↓

Calls the appropriate overloaded function

5. Example Program

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

class Calculator {
public:
    // Function with two int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Function with three int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Function with two float parameters
    float add(float a, float b) {
        return a + b;
    }
};

int main() {
    Calculator obj;

    cout << "Sum (2, 3) = "
         << obj.add(2, 3) << endl;   // calls add(int, int)

    cout << "Sum (2, 3, 4) = "
         << obj.add(2, 3, 4) << endl; // calls add(int, int, int)

    cout << "Sum (2.5, 3.5) = "
         << obj.add(2.5f, 3.5f) << endl; // calls add(float, float)

    return 0;
}
```

Output

```
Sum (2, 3) = 5
Sum (2, 3, 4) = 9
Sum (2.5, 3.5) = 6
```

6. Rules for Overloading

1. Functions must have the same name.
2. Parameter list must be different (number, type or order).
3. Return type alone cannot distinguish overloaded functions.
4. Default arguments can be used with function overloading.

7. Advantages

- ✓ Improves code readability.
- ✓ Provides flexibility.
- ✓ Allows the same operation to work on different data types.
- ✓ Makes the program easy to use and maintain.

8. Disadvantages

- → Can make the program confusing if overused.
- → Increases the number of function definitions to manage.

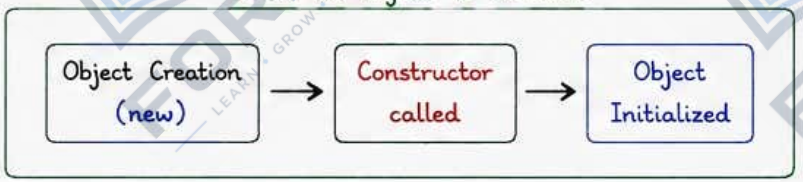
In Short : Function Overloading allows multiple functions with the same name but different parameter lists in the same scope, and the compiler selects the best match at compile time.

CONSTRUCTORS : TYPES OF CONSTRUCTORS

A constructor is a special member function of a class that is automatically called when an object is created.

- It has the same name as the class.
- It has no return type (not even void).
- It is used to initialize objects.

When an object is created



TYPES OF CONSTRUCTORS

① DEFAULT CONSTRUCTOR

- A constructor that takes no argument.
- If we do not define any constructor, the compiler provides a default constructor automatically.

Example :

```
class Student {
    int roll;
    String name;

    // Default Constructor
    Student() {
        roll = 0;
        name = "Unknown";
    }
}
```

Example Output :

```
roll = 0
name = Unknown
```

② PARAMETERIZED CONSTRUCTOR

- A constructor that takes one or more arguments.
- Used to initialize the object with specific values.

Example :

```
class Student {
    int roll;
    String name;

    // Parameterized Constructor
    Student(int r, String n) {
        roll = r;
        name = n;
    }
}
```

Example Output :

```
roll = 101          // If we create
name = Yunes        // Student(101, "Yunes")
```

③ COPY CONSTRUCTOR

- A constructor that creates an object as a copy of another object of the same class.

Example :

```
class Student {
    int roll;
    String name;

    // Copy Constructor
    Student(Student s) {
        roll = s.roll;
        name = s.name;
    }
}
```

Example :

```
Student s1 = Student(101, "Yunes");
Student s2 = Student(s1);   // copy
Output: s2 will have roll = 101,
        name = Yunes
```

④ DYNAMIC CONSTRUCTOR

- Allocates memory dynamically inside the constructor using new keyword.
- Useful when we need memory at run time whose size may vary.

Example :

```
class ArrayDemo {
    int *arr;
    int size;

    // Dynamic Constructor
    ArrayDemo(int n) {
        size = n;
        arr = new int[n];   // Dynamic memory
    }
}
```

Example :

```
ArrayDemo obj = ArrayDemo(5);
Memory for 5 integers is allocated
dynamically inside constructor.
```

Important Notes

- ✓ Constructor is called automatically.
- ✓ It cannot return any value.
- ✓ Constructors can be overloaded.

Summary Table

Type	Arguments	Purpose	Provided by Compiler	Example Use
Default Constructor	No arguments	Initializes with default values	Yes (if not defined)	Object with default state
Parameterized Constructor	One or more	Initializes with given values	No	Object with specific values
Copy Constructor	Object of same class	Creates copy of existing object	No	Copying objects
Dynamic Constructor	One or more (size, etc.)	Allocates memory dynamically	No	Dynamic memory allocation

DEFAULT, PARAMETERIZED AND COPY CONSTRUCTOR

A constructor is a special member function of a class that is automatically called when an object is created. It has the same name as the class and has no return type (not even void). Constructors are used to initialize objects.

① DEFAULT CONSTRUCTOR

- A constructor that takes no argument.
- If we do not define any constructor, compiler provides a default constructor automatically.

Syntax : `ClassName() { ... }`

Example :

```
class Student {
    int roll;
    string name;
public:
    Student() {          // Default Constructor
        roll = 0;
        name = "Unknown";
    }
    void display() {
        cout << "roll = " << roll << ", name = " << name;
    }
};
```

Example Output :

```
roll = 0 , name = Unknown
```

② PARAMETERIZED CONSTRUCTOR

- A constructor that takes one or more arguments.
- Used to initialize the object with specific values.

Syntax : `ClassName(type p1, type p2, ...) { ... }`

Example :

```
class Student {
    int roll;
    string name;
public:
    Student(int r, string n) {  // Parameterized Constructor
        roll = r;
        name = n;
    }
    void display() {
        cout << "roll = " << roll << ", name = " << name;
    }
};
```

Example Output :

```
roll = 101 , name = Yunus
(for object Student s(101, "Yunus"))
```

③ COPY CONSTRUCTOR

- A constructor that creates an object as a copy of another object of the same class.
- Used to copy the values from one object to another.

Syntax : `ClassName(const ClassName &obj) { ... }`

Example :

```
class Student {
    int roll;
    string name;
public:
    Student(int r, string n) {  // Parameterized Constructor
        roll = r;
        name = n;
    }
    Student(const Student &s) {  // Copy Constructor
        roll = s.roll;
        name = s.name;
    }
    void display() {
        cout << "roll = " << roll << ", name = " << name;
    }
};
```

Example :

```
Student s1(101, "Yunus");   // Parameterized Constructor
Student s2(s1);             // Copy Constructor
s2.display();
```

Output :

```
roll = 101 , name = Yunus
(s2 is a copy of s1)
```

Key Points :

→ Default Constructor : No argument, initializes object with default values.
→ Parameterized Constructor : Takes arguments, initializes with given values.
→ Copy Constructor : Takes another object of same class, creates its copy.

INHERITANCE in C++

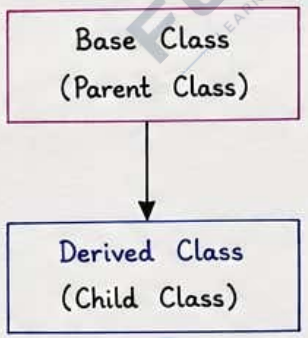
★ 1. DEFINITION

Inheritance is one of the fundamental features of Object-Oriented Programming (OOP). It allows a new class (derived class) to acquire the properties and behaviors (data members and member functions) of an existing class (base class).

Key Points :

- It promotes code reusability.
- It establishes an "IS-A" relationship.
- It enables the derived class to add its own features in addition to the base class.

Example :



Existing class whose properties are inherited.

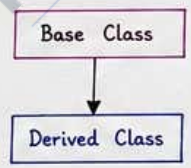
New class that inherits properties of base class.

2. TYPES OF INHERITANCE

Inheritance can be classified into the following types in C++ :

1. Single Inheritance

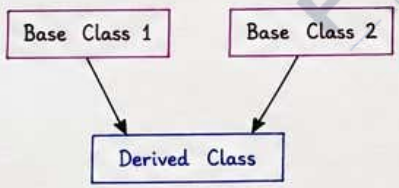
A derived class is created from a single base class.



Ex : Class B is derived from Class A.

2. Multiple Inheritance

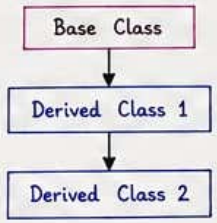
A derived class is created from more than one base class.



Ex : Class C is derived from Class A and Class B.

3. Multilevel Inheritance

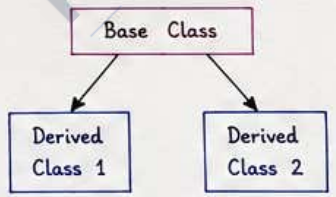
A derived class is created from another derived class.



Ex : Class C is derived from Class B, which is derived from Class A.

4. Hierarchical Inheritance

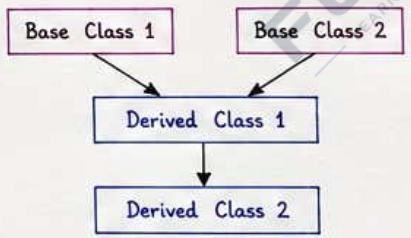
More than one derived class is created from a single base class.



Ex : Class B and Class C are derived from Class A.

5. Hybrid Inheritance

Combination of two or more types of inheritance.



Ex : Combination of multiple and multilevel inheritance.

Summary :

- → Inheritance helps in reusing existing code.
- → It reduces redundancy and improves maintainability of the program.
- → It is a transitive property, i.e., if Class C is derived from Class B, and Class B is derived from Class A, then Class C automatically inherits features of Class A.

Inheritance makes code reusable and extensible. ☺

INHERITANCE in C++

Inheritance is the feature of OOP in C++ that allows a new class (derived class) to acquire the properties and behaviors (data members and member functions) of an existing class (base class).

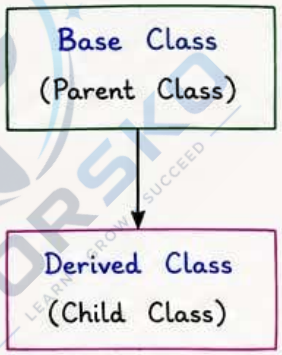
Key Points :

- Reusability of code
- Establishes IS-A relationship
- Reduces redundancy
- Improves maintainability

TYPES OF INHERITANCE

1. SINGLE INHERITANCE

When a derived class is created from a single base class, it is called single inheritance.



Example :

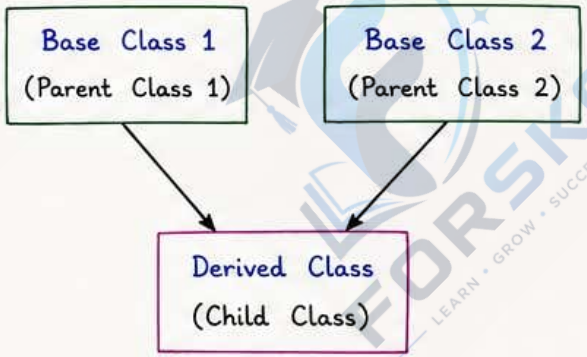
```
class Animal {
public:
    void eat() {
        cout << "Eating...";
    }
};

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

Explanation :

Dog (derived class) inherits the properties and functions of Animal (base class).

2. MULTIPLE INHERITANCE

When a derived class is created from more than one base class, it is called multiple inheritance.



Explanation :

SportsPerson (derived class) inherits the properties and functions of both Student and Athlete (base classes).

Example :

```
class Student {
public:
    void study() {
        cout << "Studying...";
    }
};
class Athlete {
public:
    void play() {
        cout << "Playing...";
    }
};
class SportsPerson : public Student,
                     public Athlete {
public:
    void info() {
        cout << "I am a Sports Person";
    }
};
```

NOTE :

→ Single inheritance represents one parent to one child.

→ Multiple inheritance represents multiple parents to one child.

IS-A Relationship

The derived class IS-A type of base class.

Example : Dog IS-A Animal

SportsPerson IS-A Student, Athlete

INHERITANCE in C++

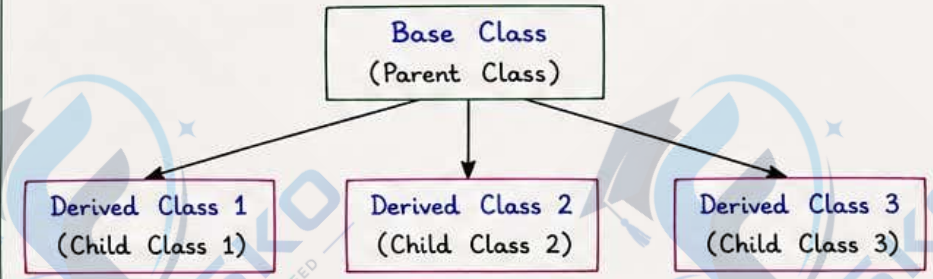
Inheritance is the feature of OOP in C++ that allows a new class (derived class) to acquire the properties and behaviors (data members and member functions) of an existing class (base class).

Key Points :

- ★ Reusability of code
- ★ Establishes IS-A relationship
- ★ Reduces redundancy
- ★ Improves maintainability

3. HIERARCHICAL INHERITANCE

When more than one derived class is created from a single base class, it is called hierarchical inheritance.



Example :

```
class Shape {                          // Base Class
  public:
    void display() {
        cout << "This is a shape";
    }
};
class Circle : public Shape {
};
class Rectangle : public Shape {
};
class Triangle : public Shape {
};
```

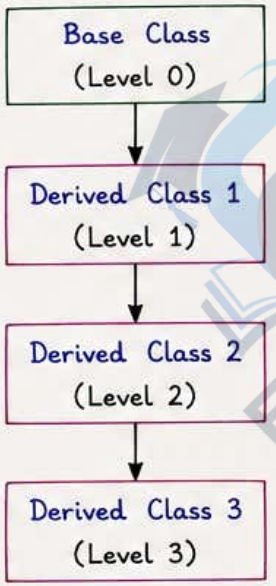
// Derived Classes

Explanation :

Circle, Rectangle and Triangle are derived from Shape. All derived classes inherit the properties of the single base class (Shape).

4. MULTILEVEL INHERITANCE

When a derived class is created from another derived class, it is called multilevel inheritance.



Example :

```
class Animal {                         // Level 0
  public:
      void eat() {
          cout << "Eating...";
      }
};
class Mammal : public Animal {         // Level 1
public:
      void walk() {
          cout << "Walking...";
      }
};
class Dog : public Mammal {            // Level 2
public:
      void bark() {
          cout << "Barking...";
      }
};
class Puppy : public Dog {             // Level 3
public:
      void weep() {
          cout << "Weeping...";
};    }
```

Explanation :

Puppy is derived from Dog, Dog is derived from Mammal, and Mammal is derived from Animal. Inheritance is carried in a chain.

Note :

Inheritance represents the IS-A relationship. It is transitive in nature. ☺

MULTILEVEL AND HYBRID INHERITANCE

Inheritance is an OOP concept in which a new class (derived/child class) acquires the properties and behaviors of an existing class (base/parent class).

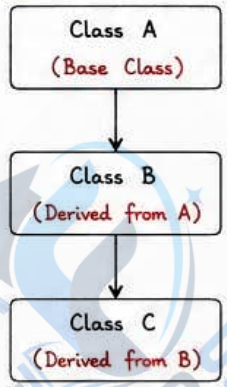
1 MULTILEVEL INHERITANCE

In multilevel inheritance, a derived class is created from another derived class.

Syntax :

```
class A {
    // base class
}
class B : public A {
    // derived class of A
}
class C : public B {
    // derived class of B
}
```

Diagram :



Example :

```
class GrandParent {
  public:
    void showGP() {
        cout << "I am GrandParent";
    }
};
class Parent : public GrandParent {
  public:
    void showParent() {
        cout << "I am Parent";
    }
};
```

```
class Child : public Parent {
  public:
    void showChild() {
        cout << "I am Child";
    }
};
int main() {
    Child obj;
    obj.showGP();     // inherited from GrandParent
    obj.showParent(); // inherited from Parent
    obj.showChild();
    return 0;
}
```

Output :

```
I am GrandParent
I am Parent
I am Child
```

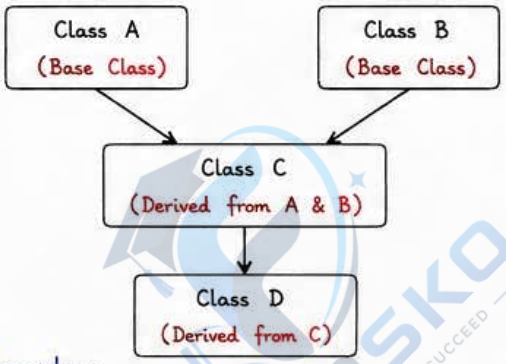
Key Points :

- The child class (C) inherits from another derived class (B).
- It forms a chain (A → B → C).
- All public members of all base classes are accessible in the child class.

2 HYBRID INHERITANCE

Hybrid inheritance is a combination of two or more types of inheritance. (e.g., Multiple + Multilevel)

Diagram :



Syntax :

```
class A { ... };
class B { ... };
class C : public A, public B {
    // derived from A and B
};
class D : public C {
    // derived from C
};
```

Example :

```
class A {
  public:
    void showA() {
        cout << "Class A";
    }
};
class B {
  public:
    void showB() {
        cout << "Class B";
    }
};
```

```
class C : public A, public B {
  public:
    void showC() {
        cout << "Class C";
    }
};
class D : public C {
  public:
    void showD() {
        cout << "Class D";
    }
};
```

```
int main() {
    D obj;
    obj.showA();   // from A
    obj.showB();   // from B
    obj.showC();   // from C
    obj.showD();   // from D
    return 0;
}
```

Output :

```
Class A
Class B
Class C
Class D
```

Key Points :

- Class C inherits from two base classes (A and B) → Multiple Inheritance.
- Class D inherits from C → Multilevel Inheritance.
- Hence, Hybrid Inheritance (combination of more than one type).

VISIBILITY MODES

Visibility modes (access specifiers) are used to control the accessibility of class members (data members and member functions) from outside the class.
The main visibility modes in OOP are : public, private and protected.

Symbols Used :
+ : public
− : private
: protected

Visibility Mode	Symbol	Access Level	Accessible from	Description
Public	+	High	• Within the class • Outside the class • Derived classes	Members are accessible from anywhere in the program.
Private	−	Low	• Within the class only	Members are accessible only within the class. Not accessible outside the class and not in derived classes.
Protected	#	Medium	• Within the class • Derived classes • Not outside the class	Members are accessible within the class and in derived classes. Not accessible outside the class.

Example Class :

```
class Demo {
 public:
    int a;          // public member
    void showA();   // public function
 private:
    int b;          // private member
    void showB();   // private function
 protected:
    int c;          // protected member
    void showC();   // protected function
};
```

a can be accessed from anywhere.

b can be accessed only within the class Demo.

c can be accessed within Demo and its derived classes. Not outside.

Accessibility of Members :

From	Public (+)	Protected (#)	Private (-)
Within same class	✓	✓	✓
Outside class	✓	✗	✗
In derived class	✓	✓	✗

Example with Derived Class :

```
class Base {
 public:
    int a;
 protected:
    int c;
 private:
    int b;
};
```

class Derived : public Base →

```
class Derived : public Base {
 public:
    void show() {
        a = 10;    // OK
        c = 20;    // OK
       // b = 30;  // Error
    }
};
```

Derived class can access a and c but not b.

Summary :
→ public members are accessible everywhere.
→ private members are accessible only within the class.
→ protected members are accessible within the class and in derived classes.
→ Use visibility modes to hide data and achieve data security (encapsulation).

Key Points :
✓ By default, class members are private (in C++).
✓ Use visibility modes to control how members are accessed in a program.
✓ These modes are essential for data hiding and encapsulation.


1010101010
0101010101
1010101010
0101010101
Unit - III
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Introduction:

History of Python Programming Language

- Python programming language was created by Guido van Rossum. He started working on Python in the late 1980s.
- Python was first released in February 1991. It was designed to be easy to read, easy to use, and powerful.
- The name "Python" was inspired by the British comedy group "Monty Python's Flying Circus", which Guido van Rossum was a big fan of.
- Python 1.0 was released in January 1994, with many new features and improvements.
- Over the years, Python has evolved through many versions. Some major versions are Python 2.0 (2000), Python 3.0 (2008).
- Today, Python is one of the most popular programming languages in the world. It is widely used in web development, data science, artificial intelligence, automation, and many other fields.

Thrust area of Python

Python is a versatile programming Language and it has a strong presence in many thrust areas.

1. Web Development : Python is widely used for web development using frameworks like Django, Flask, Pyramid, etc. It helps in building dynamic and powerful websites quickly.
2. Data Science and Analytics : Python is a top choice for data analysis, data visualization and machine learning. Libraries like NumPy, Pandas, Matplotlib, Scikit-learn are widely used in this area.
3. Artificial Intelligence and Machine Learning : Python supports AI and ML with libraries such as TensorFlow, Keras, PyTorch. It is used to build smart systems that can learn and make decisions.
4. Automation and Scripting : Python is used for automating repetitive tasks like file handling, web scraping, testing, and system administration.
5. Game Development : Python is used in game development using libraries like Pygame. It allows easy creation of 2D games and simple interactive applications.
6. Scientific and Numeric Computing : Python is used in scientific research, simulations and engineering applications. Libraries like SciPy and SymPy are commonly used in this field.

Parts of Python Programming Language:

1. Identifiers

- Identifiers are the names given to variables, functions, classes, modules or any other user-defined objects in a Python program.
- They help to identify different elements in the program.

Rules for Identifiers :

1. An identifier can contain letters (a-z, A-Z), digits (0-9) and underscore (_).
2. An identifier must start with a letter (a-z, A-Z) or underscore (_).
3. An identifier cannot start with a digit.
4. Identifiers are case-sensitive. (myVar and myvar are different)
5. No special symbols are allowed except underscore (_).
6. Python keywords cannot be used as identifiers.

Examples :

Valid Identifiers			Invalid Identifiers		
name	_name	name1	1name	name!	my var
student_name		sum1	for	while	2value
age_2024		_value	class	@data	sum-1
myVariable		TOTAL			

2. Keywords

- Keywords are reserved words in Python that have a special meaning.
- They have predefined purposes and cannot be used as identifiers (variable names, function names, etc.).
- All keywords in Python are written in lowercase.

List of Python Keywords :

False	class	finally	is	return	super	while
None	continue	for	lambda	try	def	with
True	break	from	nonlocal	except	del	as
and	else	global	not	raise	elif	assert
or	pass	if	or	in	import	yield

Statements

In Python, a statement is an instruction that the Python interpreter can execute. Each statement performs some action. Python statements are written on a single line or can span multiple lines using line continuation.

1. Simple Statements :

These are the most basic statements and perform a simple action.

Examples :

- Assignment statement → `x = 10` # Assignment
- Expression statement → `x + y` # Expression
- Print statement → `print("Hello, Python!")` # Print
- Delete statement → `del x` # Delete
- Pass statement → `pass` # Pass (does nothing)

2. Compound (Structured) Statements :

These statements contain one or more blocks of statements.

Examples :

- If statement
- If...else statement
- For loop
- While loop
- Function definition
- Class definition
- Try...except statement

```
if x > 0:
    print("Positive")                    # If statement
if x > 0:
    print("Positive")
else:
    print("Non-positive")                # If...else statement
for i in range(5):
    print(i)                             # For loop
while x > 0:
    x = x - 1                            # While loop
def add(a, b):
    return a + b                         # Function definition
class Student:
    pass                                 # Class definition
try:
    x = 10 / y
except ZeroDivisionError:
    print("Cannot divide by zero")       # Try...except statement
```

3. Multi-line Statements :

A statement can extend over multiple lines using line continuation ( ).

Example :

```
total = 1 + 2 + 3 + \
        4 + 5 + 6                        # Line continuation using backslash
print(total)                             # Output: 21
```

Expressions

An expression in Python is a combination of values, variables, operators and function calls that is evaluated to produce a single result (a value). Expressions are used in computations, assignments, conditions, function calls, etc.

Components of an Expression :

- Operands → Values or variables on which operations are performed.
- Operators → Symbols that perform operations on operands.
- Parentheses → Used to group sub-expressions and control precedence.

Examples of Expressions :

```
①  5 + 3                  →  8                          # Arithmetic expression
②  x * y                  →  product of x and y         # Using variables
③  (a + b) / 2            →  average of a and b         # Using parentheses
④  x**2 + 3*x + 2         →  polynomial expression      # Using exponent and operators
⑤  len(name)              →  length of string "name"    # Function call expression
⑥  a > b                  →  True or False              # Relational expression
⑦  (x > 0) and (y < 5)    →  True or False              # Logical expression
```

Types of Expressions :

Type	Description	Example	Evaluates to
Arithmetic Expression	Uses arithmetic operators (+, -, *, /, %, //, **)	10 + 4 * 2	18
Relational Expression	Uses relational operators (==, !=, <, >, <=, >=)	x >= 10	True / False
Logical Expression	Uses logical operators (and, or, not)	(x > 0) and (y < 5)	True / False
Assignment Expression	Assigns a value to a variable	x = 5 + 3	8 (assigned to x)
Function Call Expression	Calls a function and returns a value	max(a, b)	Maximum of a and b

Rules :

- An expression must have at least one operand.
- The order of evaluation is determined by operator precedence.
- Parentheses can be used to change the default order.

Example in Python Shell :

```
>>> x = 10              # Statement (assignment)
>>> y = 3
>>> x + y               # Expression
    13
>>> (x * y) + 2         # Expression
    32
>>> x > y               # Expression
    True
```

More Examples :

```
>>> 2 ** 3                  # Exponent expression
    8
>>> (5 + 2) * (8 - 3)       # Arithmetic expression
    35
>>> not (x == y)            # Logical expression
    True
```

Variables

A variable is a name given to a memory location in a program. It is used to store data (values) that can be changed during the execution of a program.

Key Points :

- Variables act as containers for storing data values.
- The value stored in a variable can change.
- Python is a dynamically typed language, so you do not need to declare the type of variable.
- A variable is created automatically when you assign a value to it.

Syntax :

```
variable_name = value
```

variable_name → name of the variable
value → data stored in the variable

Example :

```
x = 10              # integer variable
name = "Python"     # string variable
price = 25.5        # float variable
is_valid = True     # boolean variable
```

Rules for Naming Variables :

1. Must start with a letter (a-z, A-Z) or underscore (_).
2. Can contain letters, digits (0-9) and underscore (_).
3. Cannot start with a digit.
4. No spaces or special characters allowed other than underscore (_).
5. Variable names are case-sensitive (age, Age and AGE are different).

Examples of Valid and Invalid Variables :

Valid Variables	Invalid Variables	
age	1age	(starts with digit)
_name	total marks	(contains space)
total_marks	@name	(contains special character)
student1	for	(keyword cannot be used)
Price2	stu$dent	(contains special character)

Changing the Value of a Variable :

```
x = 5               # x stores 5
x = x + 2           # x now stores 7
x = 20              # x now stores 20
```

Multiple Variables in One Line :

```
a, b, c = 1, 2, 3       # a = 1, b = 2, c = 3
x = y = z = 100         # x = 100, y = 100, z = 100
```

Operators

Operators are special symbols that perform operations on one or more operands (values or variables). Python supports a wide range of operators.

1. Types of Operators in Python :

- Arithmetic Operators
- Assignment Operators
- Comparison (Relational) Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

2. Arithmetic Operators : Used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 3	1.666...
//	Floor Division (Quotient)	5 // 3	1
%	Modulus (Remainder)	5 % 3	2
**	Exponentiation (Power)	5 ** 3	125

3. Assignment Operators : Used to assign values to variables.

Operator	Example	Same as	Operator	Example	Same as
=	x = 5	x = 5	/=	x /= 3	x = x / 3
+=	x += 3	x = x + 3	%=	x %= 3	x = x % 3
-=	x -= 3	x = x - 3	//=	x //= 3	x = x // 3
*=	x *= 3	x = x * 3	**=	x **= 3	x = x ** 3

4. Comparison (Relational) Operators : Used to compare two values.

Operator	Description	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	5 <= 3	False

5. Logical Operators : Used to combine conditions.

Operator	Description	Example	Result
and	True if both are True	(5 > 3) and (3 > 1)	True
or	True if any one is True	(5 < 3) or (3 > 1)	True
not	Reverse the result	not (5 < 3)	True

6. Membership Operators :

Used to test membership in a sequence.

Operator	Description	Example	Result
in	Present in	3 in [1,2,3,4]	True
not in	Not present in	5 not in [1,2,3]	True

7. Identity Operators :

Used to compare memory locations of objects.

Operator	Description	Example	Result
is	True if same object	a is b	True/False
is not	True if not same object	a is not b	True/False

8. Bitwise Operators :

Work with binary representation of integers.

Operator	Description	Example
&	Bitwise AND	5 & 3
|	Bitwise OR	5 | 3
^	Bitwise XOR	5 ^ 3
~	Bitwise NOT	~5
<<	Left Shift	5 << 1
>>	Right Shift	5 >> 1

Precedence and Associativity

When an expression contains multiple operators, Python evaluates it according to a fixed order of precedence.

If two operators have the same precedence, the associativity rule decides the order of evaluation.

1. Operator Precedence in Python

Precedence (High to Low)	Operator	Description	Example	Associativity
1	( )	Parentheses	(2 + 3) * 4	–
2	**	Exponentiation (Power)	2 ** 3	Right to Left
3	+x -x ~x	Unary plus, Unary minus, Bitwise NOT	-5 , +5 , ~5	Right to Left
4	* / // %	Multiplication, Division, Floor Division, Modulus	10 * 2 , 10 % 3	Left to Right
5	+ -	Addition, Subtraction	10 + 5 , 10 - 5	Left to Right
6	<< >>	Left shift, Right shift	5 << 1	Left to Right
7	&	Bitwise AND	5 & 3	Left to Right
8	^	Bitwise XOR	5 ^ 3	Left to Right
9	|	Bitwise OR	5 | 3	Left to Right
10	< <= > >=	Relational operators	5 < 3	Left to Right
11	== !=	Equality operators	5 == 3	Left to Right
12	not	Logical NOT	not True	Right to Left
13	and	Logical AND	True and False	Left to Right
	or	Logical OR	True or False	Left to Right

2. Associativity

If operators with the same precedence appear in an expression, the associativity rule decides in which direction they are evaluated.

- **Left to Right**
  Evaluation starts from the leftmost operator and moves to the right.
- **Right to Left**
  Evaluation starts from the rightmost operator and moves to the left.

3. Examples

- a) 12 + 5 * 2
  → Multiplication has higher precedence than addition.
  → So, 5 * 2 is evaluated first.
  = 12 + 10 = 22
- b) (12 + 5) * 2
  → Parentheses have highest precedence.
  → So, 12 + 5 is evaluated first.
  = 17 * 2 = 34
- c) 2 ** 3 ** 2
  → ** is right to left associative.
  → So, 3 ** 2 is evaluated first.
  = 2 ** 9 = 512
- d) 20 - 6 - 4
  → - has same precedence and is left to right.
  → (20 - 6) - 4 = 14 - 4 = 10
- e) True or False and False
  → and has higher precedence than or.
  → So, False and False is evaluated first.
  = True or False = True
- f) not True and False
  → not has higher precedence than and.
  → not True is evaluated first.
  = False and False = False

Summary :

1. Operators are evaluated according to precedence (High to Low).
2. If operators have the same precedence, associativity (Left to Right / Right to Left) determines the order.

Data Types in Python

A data type specifies the type of value a variable can store and the type of operations that can be performed on that value.

1. Built-in Data Types :

Data Type	Description	Example	Value	Type (in Python)
int (Integer)	Represents whole numbers (positive, negative or zero)	x = 10 y = -25	10 -25	<class 'int'>
float (Floating point)	Represents decimal numbers (numbers with fractional part)	pi = 3.14 rate = -0.75	3.14 -0.75	<class 'float'>
complex (Complex number)	Represents complex numbers in the form a + bj	z = 3 + 4j	(3+4j)	<class 'complex'>
bool (Boolean)	Represents truth values True or False	a = True b = False	True False	<class 'bool'>
str (String)	Represents sequence of characters (text)	name = "Yunus" msg = 'Hello'	"Yunus" 'Hello'	<class 'str'>
list (List)	Represents ordered collection of items (can be of different types)	L = [10, 20, 'abc', 3.5]	[10, 20, 'abc', 3.5]	<class 'list'>
tuple (Tuple)	Represents ordered collection of items (immutable)	T = (10, 20, 'abc')	(10, 20, 'abc')	<class 'tuple'>
set (Set)	Represents unordered collection of unique items	S = {10, 20, 30}	{10, 20, 30}	<class 'set'>
dict (Dictionary)	Represents collection of key-value pairs (unordered)	D = {'id': 1, 'name': 'Ali'}	{'id': 1, 'name': 'Ali'}	<class 'dict'>

2. Type Checking :

- We can check the data type of any value using the type() function.

Example : type(10) → <class 'int'> type(3.14) → <class 'float'> type("Yunus") → <class 'str'>

type(True) → <class 'bool'> type([1,2,3]) → <class 'list'> type({'a': 1}) → <class 'dict'>

Note : Python is a dynamically typed language, so we do not need to declare the type of a variable explicitly.

Indentation

Indentation refers to the spaces at the beginning of a line. In Python, indentation is very important. It is used to define a block of code.

- Python uses indentation to indicate a block of code.
- All statements within the same block must have the same indentation level.
- Unlike other languages that use { }, Python uses indentation.

Example :

Correct Indentation

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")
```

→ Here, print() statements are indented inside the if and else blocks.

Incorrect Indentation

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")
```

→ This will give an IndentationError because the indentation is incorrect.

Note :

- Use consistent indentation (usually 4 spaces) throughout the program.
- Do not mix tabs and spaces.

Comments

Comments are used to explain the code. They are ignored by the Python interpreter and are used to make the code more readable.

1. Single-line Comments

In Python, single-line comments start with the # symbol.

Example :

```
# This is a single-line comment
x = 10    # x stores the value 10
print(x)  # This will print the value of x
```

2. Multi-line Comments

Python does not have a special syntax for multi-line comments like /* ... */ in other languages. We can use triple quotes (''' or """) to write multi-line comments.

Example :

```
'''
This is a
multi-line comment.
It can span
across multiple lines.
'''
x = 25
print(x)
```

→ The text inside triple quotes is ignored by the interpreter.

Note : Comments are useful for documentation and understanding your code, especially in large programs.

Reading Input, Printing Output

1. Reading Input

In Python, we use the input() function to read data from the user.

```
Syntax :   variable = input(prompt)
```

- input() always reads the data as a string.
- We can convert it to other data types using type conversion functions like int(), float() etc.

Example 1 : Reading a string input

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

Sample Run :

```
Enter your name: Yunus
Hello, Yunus!
```

Example 2 : Reading integer input

```
num = int(input("Enter a number: "))
print("You entered:", num)
```

Sample Run :

```
Enter a number: 25
You entered: 25
```

Example 3 : Reading float input

```
price = float(input("Enter price: "))
print("Price is:", price)
```

Sample Run :

```
Enter price: 99.50
Price is: 99.5
```

Note : The input() function always returns a string. Use int(), float(), etc. to convert it to the required data type.

2. Printing Output

In Python, we use the print() function to display output on the screen.

```
Syntax :   print(value1, value2, ..., sep=' ', end='\n')
```

- sep is used to separate values (default is space).
- end is used to end the line (default is newline).

Example 1 : Printing a single value

```
print("Hello, Python!")
```

Output :

```
Hello, Python!
```

Example 2 : Printing multiple values

```
name = "Yunus"
age = 20
print("Name:", name, "Age:", age)
```

Output :

```
Name: Yunus Age: 20
```

Example 3 : Using sep parameter

```
print(10, 20, 30, sep=' - ')
```

Output :

```
10 - 20 - 30
```

Example 4 : Using end parameter

```
print("Hello", end=' ')
print("Python!")
```

Output :

```
Hello Python!
```

Note : The print() function is used to show results, messages or any output to the user.

Type Conversions

Type conversion (or type casting) is the process of converting a value from one data type to another.
In Python, we can convert data types using built-in functions.

1. Implicit Type Conversion (Automatic)

Python automatically converts a smaller data type to a larger data type to avoid data loss.

Example :

```
a = 10        # int
b = 3.5       # float
c = a + b     # int is converted to float
print(c)                    Output : 13.5
```

2. Explicit Type Conversion (Manual)

We can convert one data type to another using built-in functions.

Common type conversion functions :

Function	Converts to	Example	Result
int( )	Integer	int(3.9)	3
float( )	Float	float(10)	10.0
str( )	String	str(25)	'25'
bool( )	Boolean	bool(0)	False

3. Examples of Type Conversion

a) int( ) – to convert to integer

```
x = 7.89
y = int(x)
print(y)
```

Output : 7 # decimal part is removed

b) float( ) – to convert to float

```
x = '25.6'
y = float(x)
print(y)
```

Output : 25.6 # string is converted to float

c) str( ) – to convert to string

```
x = 100
y = str(x)
print(y)
```

Output : '100' # number is converted to string

d) bool( ) – to convert to boolean

```
x = 0
y = bool(x)
print(y)
```

Output : False # 0 is False in boolean

4. Type Conversion in Expressions

```
a = '15'        # string
b = 5           # integer
c = int(a) + b  # '15' is converted to 15
print(c)        # 15 + 5 = 20      Output : 20
```

Note :

- While converting, make sure the value is in a valid format.
- Invalid conversion will raise an error.

  Example :

  int('abc') will give a ValueError.

The type() Function and Is Operator

1. The type() Function

The type() function is used to find the data type of a value or variable. It returns the class type of the object.

Syntax : `type(object)`

Example :

Code	Output	Explanation
`x = 10` `print(type(x))`	`<class 'int'>`	x is an integer
`y = 3.14` `print(type(y))`	`<class 'float'>`	y is a float
`name = "Yunus"` `print(type(name))`	`<class 'str'>`	name is a string
`is_valid = True` `print(type(is_valid))`	`<class 'bool'>`	is_valid is boolean
`lst = [1, 2, 3]` `print(type(lst))`	`<class 'list'>`	lst is a list
`tpl = (1, 2, 3)` `print(type(tpl))`	`<class 'tuple'>`	tpl is a tuple
`d = {'a': 1, 'b': 2}` `print(type(d))`	`<class 'dict'>`	d is a dictionary

Note : type() tells us what type (class) the object belongs to. It does not check the value, only the data type.

2. Is Operator

The is operator is used to check whether two variables refer to the same object in memory.

Syntax : `variable1 is variable2`

Example 1 :

```
a = 100
b = 100
print(a is b)      # Output : True
```

→ Both a and b refer to the same object with value 100 (small integers are cached).

Example 2 :

```
x = [1, 2, 3]
y = x
print(x is y)      # Output : True
```

→ y refers to the same list object as x.

Example 3 :

```
p = [1, 2, 3]
q = [1, 2, 3]
print(p is q)      # Output : False
```

→ p and q have the same values but are different objects in memory.

Key Points :

- 'is' checks identity (same object in memory).
- It is not used to compare values.
- Use == to compare values.

Note : Use 'is' carefully. For most comparisons of values, use ==.

Dynamic and Strongly Typed Language

1. Dynamic Language

A dynamic language is one in which the data type of a variable is checked and decided at runtime.

- We do not need to declare the type of a variable.
- The type can change during program execution.
- More flexible and easier to write code.

Example (Python) :

```
x = 10          # x is an integer
print(type(x))  # Output: <class 'int'>
x = "Yunus"     # now x is a string
print(type(x))  # Output: <class 'str'>
x = 25.5        # now x is a float
print(type(x))  # Output: <class 'float'>
```

Key Point :

- Type of variable is determined at runtime.
- Same variable can hold different types of values.
- Example languages: Python, JavaScript, Ruby, PHP.

2. Strongly Typed Language

A strongly typed language is one in which the data type of a variable is checked at compile time.

- We must declare the type of a variable.
- The type cannot be changed once declared.
- Less flexible but safer and reduces type-related errors.

Example (Java) :

```
int x = 10;
System.out.println(x);  // Output: 10

x = "Yunus";  // Error: Type mismatch
// (cannot assign String to int)
```

Key Point :

- Type of variable is determined at compile time.
- Once declared, a variable cannot hold a different type.
- Example languages: Java, C, C++, C#, Rust.

Note : Python is both dynamic and strongly typed.

- ➤ Dynamic – type is decided at runtime.
- ➤ Strongly Typed – operations are performed only on compatible types (e.g., int + int is allowed, int + "text" is not allowed).


1010101010
0101010101
1010101010
0101010101
</>
</>
Unit - IV
10110
01001
10101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Control Flow Statements: The if, if-else

Control flow statements are used to control the execution of a program based on certain conditions.

1. The if Statement

The if statement is used to execute a block of code only if the given condition is True.

Syntax :

```
if condition :
    # block of code to be executed
      if condition is True
```

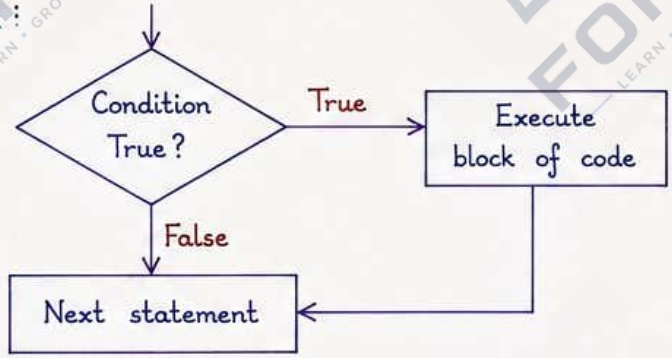
Example :

```
age = 18
if age >= 18 :
    print ("You are eligible to vote.")
```

Output :

```
You are eligible to vote.
```

Flowchart :



2. The if - else Statement

The if-else statement is used to execute one block of code if the condition is True and another block if the condition is False.

Syntax :

```
if condition :
    # block of code if condition is True
else :
    # block of code if condition is False
```

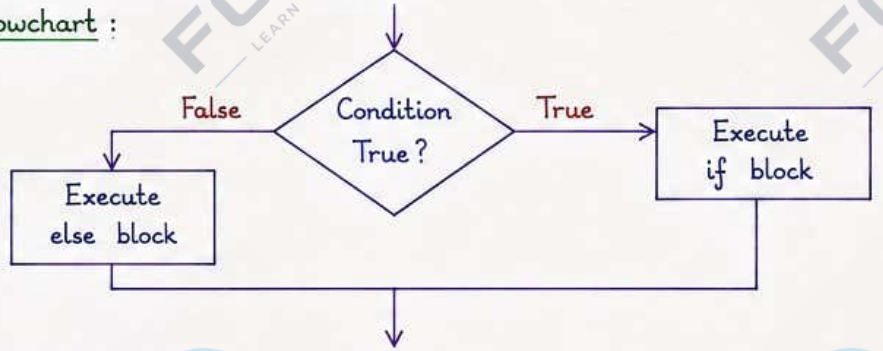
Example :

```
marks = 60
if marks >= 40 :
    print ("You passed!")
else :
    print ("You failed!")
```

Output :

```
You passed!
```

Flowchart :



Note :

- In the if statement, the else part is optional.
- Indentation is very important in Python.
- Code inside the if block runs only when the condition is True.
- In if-else, either the if block or the else block will be executed, not both.

if - elif - else Decision Control Statement

The if - elif - else statement is used to execute one block of code from multiple alternatives based on different conditions.

1. Syntax :

```
if   condition1 :
     # block of code if condition1 is True
elif  condition2 :
     # block of code if condition2 is True
elif  condition3 :
     # block of code if condition3 is True
     ...
else :
     # block of code if all conditions are False
```

2. Example (Python) :

```
marks = int(input("Enter your marks (0-100): "))
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
elif marks >= 35:
    print("Grade: D")
else:
    print("Grade: Fail")
```

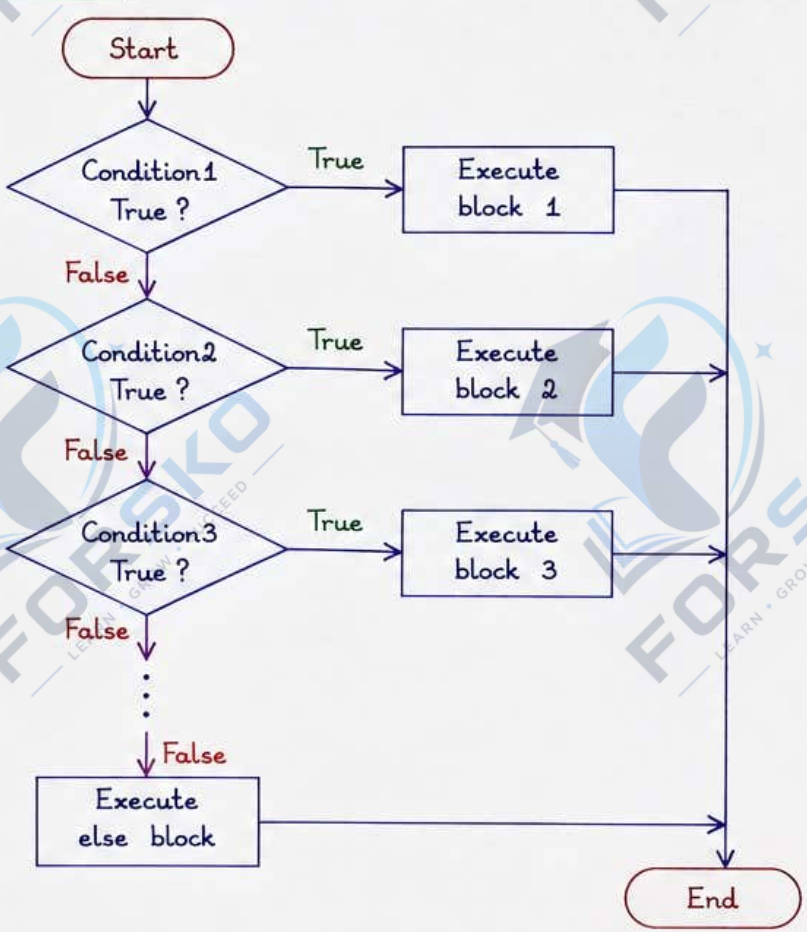
Sample Run :

```
Enter your marks (0-100): 82
Grade: B
```

```
Enter your marks (0-100): 56
Grade: C
```

```
Enter your marks (0-100): 28
Grade: Fail
```

3. Flowchart :



Note :

- The program checks conditions from top to bottom.
- As soon as one condition is True, its block is executed and the rest are skipped.
- If none of the conditions are True, the else block is executed.

Nested if Statement

A nested if statement is an if statement inside another if (or else) statement.
It is used to check multiple conditions (one condition inside another).

1. Syntax :

```
if condition1 :
    if condition2 :
        # block of code if condition1 and condition2 are True
    else :
        # block of code if condition2 is False
else :
    # block of code if condition1 is False
```

2. Example (Python) :

```
marks = int(input("Enter your marks (0-100): "))
if marks >= 50:
    if marks >= 75:
        print("Distinction")
    else :
        print("First Class")
else :
    if marks >= 35:
        print("Pass Class")
    else :
        print("Fail")
```

3. Sample Runs :

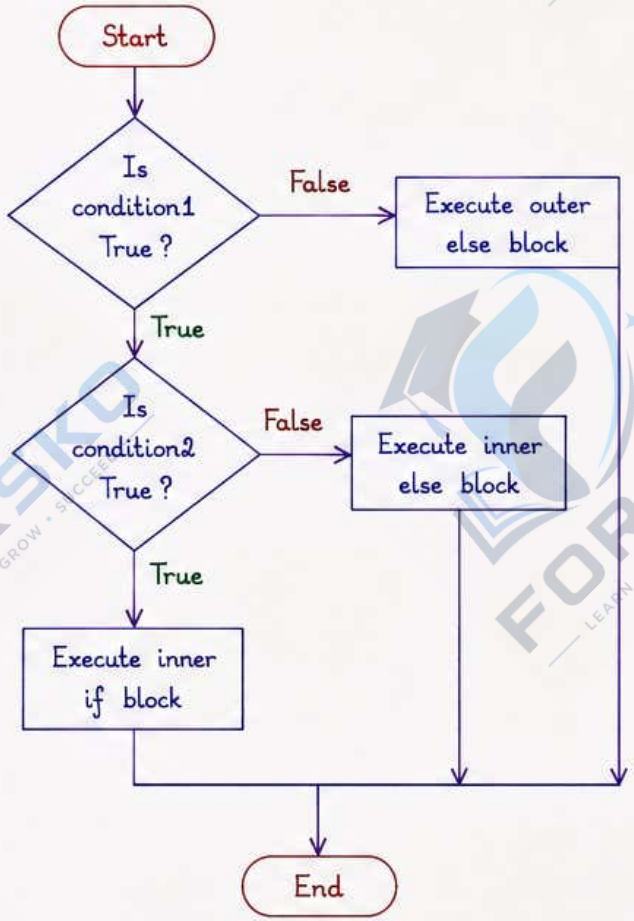
```
Enter your marks (0-100): 85
Distinction
```

```
Enter your marks (0-100): 60
First Class
```

```
Enter your marks (0-100): 40
Pass Class
```

```
Enter your marks (0-100): 20
Fail
```

4. Flowchart :



Note :

- The inner if-else is executed only when the outer if condition is True.
- If the outer if condition is False, the inner if is skipped and the outer else block is executed.
- Indentation is very important in nested if statements.

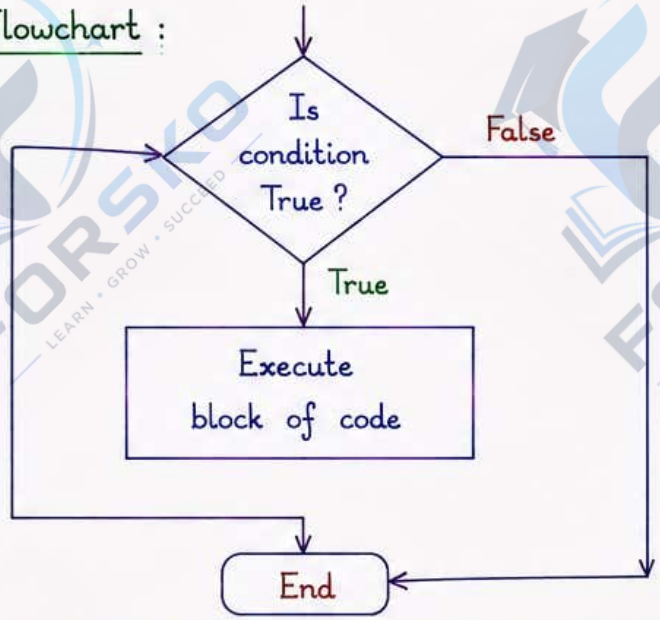
The while Loop

The while loop is used to execute a block of code repeatedly as long as the given condition is True.

1. Syntax :

```
while condition :
    # block of code
    # (repeated while condition is True)
```

2. Flowchart :



3. Example (Python) :

```
i = 1                       # initialization
while i <= 5:               # condition
    print(i, end = " ") # block of code
    i = i + 1               # update
```

Output : 1 2 3 4 5

4. Example Explanation :

- Initially, i = 1.
- Check condition: i <= 5 → True → print 1 and i becomes 2.
- Again check: i <= 5 → True → print 2 and i becomes 3.
- This continues until i = 5.
- After printing 5, i becomes 6.
- Now, i <= 5 → False, so loop terminates.

Key Points :

- The block of code inside the while loop is executed zero or more times.
- If the condition is False at the beginning, the block will not execute even once.
- Make sure to update the condition inside the loop, otherwise it may cause an infinite loop.

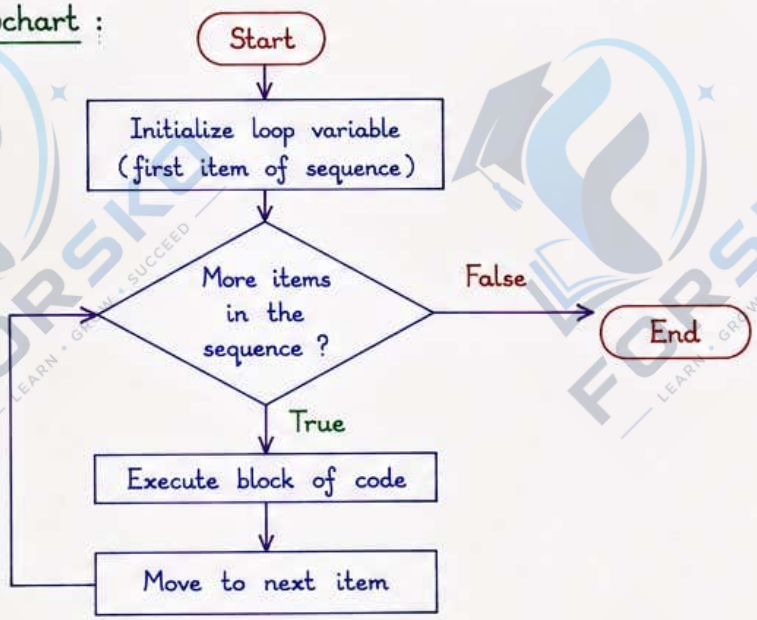
The for Loop

The for loop is used to iterate over a sequence (such as list, tuple, string, range etc.) and execute a block of code for each item.

1. Syntax :

```
for variable in sequence :
    # block of code
    # (repeated for each item in the sequence)
```

2. Flowchart :



3. Example (Python) :

```
fruits = ["Apple", "Banana", "Mango", "Orange"]   # sequence (list)
for fruit in fruits :                             # loop through each item
    print(fruit)                                  # block of code
```

Output :

```
Apple
Banana
Mango
Orange
```

4. Example using range() :

```
for i in range(1, 6):      # range from 1 to 5
    print(i, end = " ")    # print numbers on same line
```

Output :

```
1 2 3 4 5
```

5. Explanation :

- The loop variable takes the first item from the sequence.
- The block of code is executed.
- Then the loop variable takes the next item and the block is executed again.
- This continues until all items in the sequence are processed.
- When there are no more items, the loop ends.

Key Points :

- The for loop is used when the number of iterations is known (or we want to iterate over a sequence).
- It can be used with lists, tuples, strings, sets, dictionaries, and the range() function.
- It automatically takes each item one by one from the sequence.
- No need to manually update the loop variable.

The continue and break Statements.

The continue and break statements are used inside loops to change the normal flow of execution.

1. The continue Statement :

The continue statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

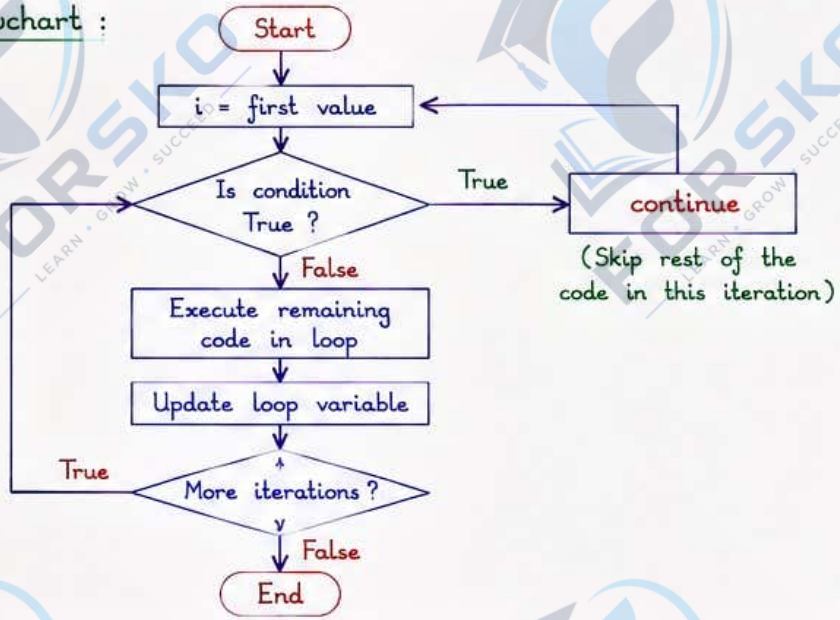
Example (Python) :

```
for i in range(1, 6):
    if i == 3:
        continue      # skip when i is 3
    print(i, end = " ")
```

Output :

```
1 2 4 5
```

Flowchart :



2. The break Statement :

The break statement terminates the loop completely and transfers the control to the statement after the loop.

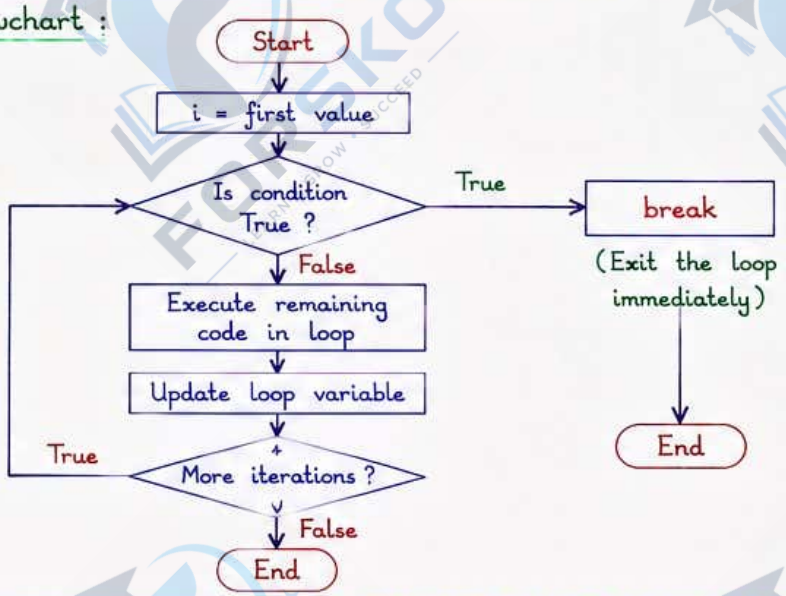
Example (Python) :

```
for i in range(1, 6):
    if i == 3:
        break      # exit the loop when i is 3
    print(i, end = " ")
print("Loop ended.")
```

Output :

```
1 2
Loop ended.
```

Flowchart :



Key Points :

- continue skips the rest of the code in the loop for the current iteration and moves to the next iteration.
- break terminates the loop completely.
- Both statements are used inside loops (for or while).
- They are useful for controlling the flow of the program.

Functions: Built-In Functions

Python provides a number of built-in functions that are always available for use.
They perform common tasks and make programs easier.

1. Common Built-In Functions :

Function	Description	Example	Result
print()	Displays the output on the screen	print("Hello, Python")	Hello, Python
type()	Returns the type of the object	type(25.5)	<class 'float'>
int()	Converts the value to integer	int("123")	123
float()	Converts the value to float	float("12.34")	12.34
str()	Converts the value to string	str(100)	'100'
len()	Returns the length of the object	len("Python")	6
input()	Takes input from the user	input("Enter name: ")	(User input)
max()	Returns the largest value	max(10, 25, 5)	25
min()	Returns the smallest value	min(10, 25, 5)	5
abs()	Returns the absolute value	abs(-7)	7
round()	Rounds the number to nearest integer	round(4.67)	5

2. Example Program Using Built-In Functions :

```
name = input("Enter your mame: ")                    # input()
age = int(input("Enter your age: "))                 # int()
print("Hello", name)                                 # print()
print("Next year, you will be", age + 1)             # + operator
print("Type of age:", type(age))                     # type()
print("Length of name:", len(name))                  # len()
```

Sample Run :

```
Enter your name: Yunus
Enter your age: 20
Hello Yunus
Next year, you will be 21
Type of age: <class 'int'>
Length of name: 5
```

3. Explanation :

- input() gets data from the user.
- int() converts the input to integer.
- print() displays output.
- type() returns the data type.
- len() gives the number of characters in the name.

Key Points :

- Built-in functions are predefined in Python.
- They save time and reduce the need to write complex code.
- Some of the most commonly used built-in functions are: print(), type(), len(), input(), int(), float(), max(), min(), abs(), round(), etc.

Commonly Used Modules

Python provides a large number of standard modules that extend its functionality.
Some of the most commonly used modules are listed below.

1. List of Commonly Used Modules :

Module	Purpose	Commonly Used Functions	Example
math	Mathematical functions and constants	sqrt(), pow(), sin(), cos(), pi, etc.	math.sqrt(16) → 4.0
random	Generate random numbers and choices	randint(), choice(), random(), shuffle(), etc.	random.randint(1,10) → e.g. 7
datetime	Work with dates and times	datetime(), date(), now(), strftime(), etc.	datetime.now() → 2024-05-20 10:30:00
time	Time related functions	time(), sleep(), ctime(), localtime(), etc.	time.sleep(2) → waits for 2 seconds
os	Interact with operating system	listdir(), mkdir(), remove(), path, etc.	os.listdir('.') → list of files
sys	System-specific parameters and functions	argv, exit(), path, version, etc.	sys.version → Python version
re	Regular expressions (pattern matching)	match(), search(), findall(), sub(), etc.	re.findall(r"\d+", "A1B2") → ["1", "2"]
json	Work with JSON data	dumps(), loads(), load(), dump(), etc.	json.dumps({"a":1}) → "{"a": 1}"
collections	Special container datatypes	Counter(), defalutdict(), namedtuple(), deque(), etc.	Counter('banana') → Counter({'a':3,'n':2,'b':1})

2. How to Use a Module :

- Import the entire module

```
import module_name                    # access with module_name.function()
```

- Import specific function(s)

```
from module_name import function1, function2    # use function directly
```

- Import module with an alias

```
import module_name as alias           # access with alias.function()
```

3. Examples :

(i) math module

```
import math
print(math.sqrt(25))                  # Output: 5.0
```

(ii) random module

```
from random import randint
print(randint(1, 100))                # Output: any number between 1 and 100
```

(iii) datetime module

```
from datetime import datetime
print(datetime.now())                 # Output: current date and time
```

(iv) os module

```
import os
print(os.getcwd())                    # Output: current working directory
```

Key Points :

- Modules provide pre-written code that can be used in our programs.
- Using modules saves time and effort.
- We can import entire modules or specific functions as per our requirement.
- Some modules are built-in (standard library) and some need to be installed using pip.

Function Definition and Calling the Function

In Python, a function is a block of code that performs a specific task. We first define (create) the function and then call (use) it whenever needed.

1. Function Definition :

A function is defined using the keyword **def**, followed by the function name, parentheses (), and a colon :

```
def greet(name):                    # function definition
    print("Hello, " + name + "!")   # function body
                                        (indented block)
```

Explanation :

- **def** is the keyword used to define a function.
- **greet** is the function name.
- **name** is a parameter (input).
- The indented block is the body of the function.

2. Calling the Function :

A function is executed (run) only when it is **called**.

```
greet("Yunus")        # function call
```

Output :

```
Hello, Yunus!
```

3. Example Program :

```
# Function definition
def add(a, b):
    sum = a + b             # statement inside function
    print("Sum is:", sum)   # print the result

# Calling the function
add(10, 20)       # function call
```

Output :

```
Sum is: 30
```

4. Explanation :

- add(a, b) is a function that takes two parameters a and b.
- When add(10, 20) is called, the values 10 and 20 are passed to a and b respectively.
- The function performs the addition and prints the result.

Key Points :

- A function must be defined before it is called.
- Parameters are the values received by the function.
- Arguments are the actual values passed while calling the function.
- A function can be called multiple times.

The return Statement and void Function

In Python, a function can return a value back to the caller using the return statement.
A function that does not return any value is called a void function.

1. The return Statement :

- The return statement is used to send a value from the function back to the caller.
- As soon as return is executed, the function terminates and control is transferred back to the caller.

```
def add(a, b):                  # function definition
    sum = a + b                 # calculate sum
    return sum                  # return the result
```

```
# Calling the function
result = add(15, 25)        # function call and store returned value
print("Sum:", result)       # Output: Sum: 40
```

2. void Function :

- A function that does not return any value is called a void function.
- It is used to perform some task without returning any result.

```
def greet(name):                        # void function definition
    print("Hello, " + name + "!")       # perform task
    # No return statement               # nothing is returned
```

```
# Calling the void function
greet("Yunus")                          # function call
print("Function call completed.")       # Execution continues here
```

Output :

```
Hello, Yunus!
Function call completed.
```

3. Example : Using both return and void functions

```
# Function with return
def square(n):
    return n * n

# Function without return (void)
def show(msg):
    print("Message:", msg)
```

```
# Main program
x = square(6)
print("Square is:", x)          # uses returned value

show("Welcome to Python!")      # void function call

print("Program ended.")
```

Output :

```
Square is: 36
Message: Welcome to Python!
Program ended.
```

Key Points :

- return sends a value from the function back to the caller.
- A function with return must return a value.
- void function does not return any value.
- A void function is used to perform actions (like printing, updating, etc.).
- Even after calling a void function, the program continues execution.

Scope and Lifetime of Variables

The scope of a variable defines the part of a program where the variable can be accessed.

The lifetime of a variable defines the period for which the variable exists in memory.

1. Scope of Variables :

Scope determines the visibility or accessibility of a variable in different parts of a program.

Type of Variable	Scope	Example

Local Variable	Declared inside a function or block.
	Can be accessed only within that function or block.
		`def fun():`
		`  x = 10  # local variable`
		`  print(x)`
		`fun()`
		`# x cannot be accessed here`

Global Variable	Declared outside all functions.
	Can be accessed from any function in the program.
		`y = 20  # global variable`
		`def fun():`
		`  print(y)  # accessible`
		`fun()`
		`print(y)`

Parameter Variable	Declared in the function definition.
	Acts as a local variable within that function.
		`def add(a, b):  # a and b are`
		`  c = a + b  # parameters`
		`  print(c)`
		`add(5, 6)`

Key Points :

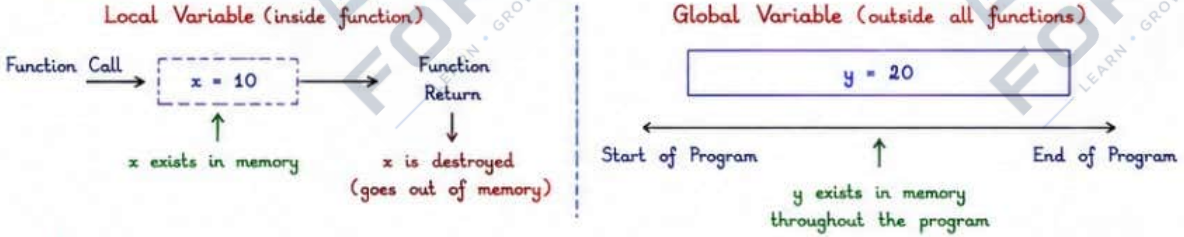
- Local variables have local scope.
- Global variables have global scope.
- Parameter variables have local scope (within the function).
- A variable can have only one scope at a time.

2. Lifetime of Variables :

Lifetime determines the time period from creation to destruction of a variable.

Type of Variable	Lifetime	Explanation
Local Variable	From the time the block or function is entered till it is exited.	Created when the function/block starts execution and destroyed when it exits.
Global Variable	From the start of the program till the end of the program.	Exists throughout the execution of the program.
Parameter Variable	From the time the function is called till it returns.	Created when the function is called and destroyed when the function returns.

3. Visualization (Memory Representation)



4. Example :

```
x = 100     # global variable
def show():
    y = 20    # local variable
    print("Inside function, y =", y)  # accessible here
show()
print("Outside function, x =", x)  # accessible
# print(y)  # Error: y is not accessible here
```

```
# x is global, exists throughout the program.
# y is local, exists only during function call.
# After function ends, y is destroyed.
# x can be used anywhere in the program.
```

Default Parameters

Definition :

Default parameters are the values assigned to parameters in a function definition. If no value is passed during function call, the default value is used.

Syntax :

```
def function_name(parameter = default_value):
    statement(s)
```

Example 1 :

```
def greet(name, msg = "Hello"):
    print(msg + ", " + name + "!")

# Function calls
greet("Yunus")                  # uses default msg
greet("Yunus", "Good Morning")  # uses provided msg

# Output
Hello, Yunus!
Good Morning, Yunus!
```

Example 2 :

```
def add(a, b = 10, c = 20):
    sum = a + b + c
    print("Sum =", sum)

# Function calls
add(5)          # b and c are default
add(5, 15)      # c is default
add(5, 15, 25)  # no default

# Output
Sum = 35
Sum = 45
Sum = 45
```

Important Points :

- Default parameters make functions more flexible and easy to use.
- Default values are used only when the argument is not provided.
- Default parameters must be at the end of tha parameter list.

Rules :

1. A parameter with a default value cannot precede a parameter without a default value.
2. Default values are evaluated only once, when the function is defined.
3. They are helpful in reducing the number of arguments in a function call.

Example (Invalid) :

```
def invalid(a = 10, b):   ✗ Wrong
                          # SyntaxError
```

Example (Valid) :

```
def valid(a, b = 10):     ✓ Correct
```

Real Life Analogy :

Think of a function like ordering food. Some items (like water) are default (automatically provided), but you can also choose extra items if you want.