C++

Akash Patil

What is C++?

C++ is a powerful, high-performance, general-purpose programming language that extends **C** with Object-Oriented Programming (OOP) features. It was developed by **Bjarne Stroustrup** in **1979** as an enhancement of the C language to support features like **classes, objects, and polymorphism**.

Key Features of C++

1. **Multi-Paradigm** – Supports **procedural**, **object-oriented**, and **generic programming**.
2. **High Performance** – Faster execution compared to high-level languages like Python or Java.
3. **Memory Control** – Uses manual memory management with pointers and dynamic allocation (`new` and `delete`).
4. **Object-Oriented Programming (OOP)** – Includes **encapsulation, inheritance, polymorphism, and abstraction**.
5. **Standard Library** – Provides built-in support for **data structures (STL), file handling, and algorithms**.

C++ Keywords

C++ **keywords** are reserved words that have special meanings in the language. These keywords **cannot** be used as variable names, function names, or identifiers.

Category	**Keywords**
Data Types	`int`, `float`, `double`, `char`, `bool`, `void`, `wchar_t`
Control Flow	`if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`
Storage Classes	`auto`, `register`, `static`, `extern`, `mutable`, `thread_local`
OOP	`class`, `struct`, `public`, `private`, `protected`, `virtual`, `this`, `new`, `delete`, `friend`
Exception Handling	`try`, `catch`, `throw`
Memory Management	`new`, `delete`
Namespace & Type	`namespace`, `using`, `typename`, `sizeof`, `typedef`
Miscellaneous	`const`, `volatile`, `explicit`, `inline`, `asm`, `nullptr`, `static_assert`, `operator`

MCQ

Which of the following is NOT a feature of C++?

a) Object-Oriented Programming

b) Automatic Memory Management

c) Multi-Paradigm Support

d) High Performance

Which feature of C++ allows the use of classes and objects?

a) Encapsulation

b) Inheritance

c) Object-Oriented Programming

d) Polymorphism

Which keyword is used in C++ for dynamic memory allocation?

a) allocate

b) malloc

c) new

d) memory

Why is C++ considered faster than Python or Java?

a) It uses an interpreter

b) It is compiled and provides better memory control

c) It has built-in garbage collection

d) It runs on a virtual machine

Basic Syntax of C++

```
#include <iostream>  // Standard input-output library
using namespace std;
int main() {
    cout << "Hello, World!" << endl;  // Print output
    return 0;
}
```

Explanation:

- `#include <iostream>` – Includes input-output stream for `cout` and `cin`.
- `using namespace std;` – Avoids writing `std::cout` every time.
- `int main()` – Entry point of the program.
- `cout << "Hello, World!" << endl;` – Prints output.
- `return 0;` – Indicates successful execution.

What Does "Avoids Writing `std::cout` Every Time" Mean?

In **C++**, the `std` namespace (short for **standard**) contains many useful functions, objects, and classes, including `cout`, `cin`, and `endl`, which are part of the **iostream** library.

Without `using namespace std;`
If you **don't** use `using namespace std;`, you have to explicitly specify `std::` before standard library elements.

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

- **`std::cout`** – Used for printing output.
- **`std::endl`** – Used for a new line.

Here, you must write `std::cout` and `std::endl` every time you use them.

With `using namespace std;`

If you **do** use `using namespace std;`, you can omit `std::` and directly use the functions

```
#include <iostream>
using namespace std;  // This allows us to use cout, cin, and endl without std::
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

- **Now `cout` and `endl` work without `std::`!**

Should You Always Use `using namespace std;`?

No, it's not always recommended!

- If multiple libraries have the **same function names**, it may lead to **ambiguity**.
- In large projects, it's better to use **`std::`** **explicitly** to avoid conflicts.

Best Practice: Use std:: explicitly in professional code
Instead of `using namespace std;`, you can use:

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

This allows you to use `cout` and `endl` without importing the entire `std` namespace.

C++ Input & Output: cin and cout

Content:

- C++ provides standard input (`cin`) and output (`cout`) for handling user interaction.
- Both belong to the **iostream** library.
- `cin` (Console Input) → Takes user input.
- `cout` (Console Output) → Displays output on the screen.

Header File Required

```
#include <iostream>  // Required for cin & cout
using namespace std;  // Avoids using std:: prefix
```

`#include <iostream>` → Required for `cin` and `cout`
`using namespace std;` → Allows direct use of `cin` & `cout` without `std::`

Using `cout` (Console Output) / Using `cin` (Console Input)

1. `cout` is used to print output on the screen.
   Uses the `<<` (insertion operator).
   Example:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

2. `cin` is used to take input from the user.
   Uses the `>>` (extraction operator).
   Example:

```
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You are " << age << " years old.";
    return 0;
}
```

Multiple Inputs & Outputs

Handling Multiple Inputs & Outputs

`cin` and `cout` can handle multiple values.

```
#include <iostream>
using namespace std;
int main() {
    string name;
    int age;

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

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

Basic Syntax and Variables

Data Types: `int`, `float`, `double`, `char`, `bool`, `string`

Variables & Constants

Operators: `+ - * / %`

```
int age = 25;

float pi = 3.14;

char grade = 'A';

string name = "Alice";

bool isPassed = true;
```

```
#include <iostream>
#include <string>//Required for string data
type
using namespace std;

int main() {
    // Variable Declarations
    int age = 25;
    float pi = 3.14;
    char grade = 'A';
    string name = "Alice";
    bool isPassed = true;

    // Displaying the Values
    cout << "Age: " << age << endl;
    cout << "Value of Pi: " << pi << endl;
    cout << "Grade: " << grade << endl;
    cout << "Name: " << name << endl;
    cout << "Passed Exam: " << (isPassed ?
"Yes" : "No") << endl;

    return 0;
}
```

Control Statements (if-else, switch)

- **Used for decision-making**
- **if-else and switch**

```
#include <iostream>

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;

    if (num % 2 == 0)
        std::cout << "Even" << std::endl;
    else
        std::cout << "Odd" << std::endl;

    return 0;
}
```

```
#include <iostream>

int main() {
    int day;
    std::cout << "Enter a number (1-7) for the
day of the week: ";
    std::cin >> day;

    switch (day) {
        case 1: std::cout << "Sunday"; break;
        case 2: std::cout << "Monday"; break;
        case 3: std::cout << "Tuesday"; break;
        case 4: std::cout << "Wednesday"; break;
        case 5: std::cout << "Thursday"; break;
        case 6: std::cout << "Friday"; break;
        case 7: std::cout << "Saturday"; break;
        default: std::cout << "Invalid input!
Enter a number between 1 and 7.";
    }

    return 0;
}
```

Loops (for, while, do-while)

while loop → Runs until condition becomes false

```
while (i <= 5) {
 std::cout << i << " ";
 i++;
     }
```

do-while loop → Executes at least once

```
do {
        std::cout << i << " ";
        i++;
    } while (i <= 5);
```

for loop → Used when iteration count is known

```
#include <iostream>

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

What is the output?

```
#include <iostream>

int main() {

    int i;

    for (i = 1; i <= 10; i++);

    {

        std::cout << i << " ";

    }

    return 0;

}
```

Arrays & Strings

Arrays in C++

- An **array** is a **fixed-size** collection of elements of the **same data type** stored **sequentially in memory**.
- Used to store multiple values in a **single variable** instead of declaring multiple variables separately.
- **Size is fixed** and must be known at the time of declaration.

Strings in C++

- A **string** is a **sequence of characters** stored in **continuous memory**.
- In C++, strings can be represented in **two ways**:
  - **C-style strings** (character arrays)
  - **C++ string class (std::string)**

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

int main()
{
int numbers[] = {1, 2, 3, 4, 5};
string name = "Alice";

cout << "First number: " << numbers[0] <<endl;
cout << "Name: " << name;

    return 0;
}
```

Functions in C++

Functions are **reusable blocks of code** that help organize and simplify programs by breaking them into smaller tasks.

void functions vs returning functions

void greet()
{ std::cout << "Hello"; }

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

```
#include <iostream>
// Void function (No return value)
void greet() {
    std::cout << "Hello, World!" << std::endl;
}

// Returning function (Returns sum)
int add(int a, int b) {
    return a + b;
}

int main() {
    greet();  // Calls void function
    int result = add(5, 3);  // Calls returning
function
    std::cout << "Sum: " << result << std::endl;
    return 0;
}
```

Pointers & Memory Management

Pointers: Store Memory Addresses

- A **pointer** is a variable that **stores the memory address** of another variable.
- Declared using *, and memory address is accessed using & operator

```
int x = 10;
int* ptr = &x;  // Pointer stores the address of x
std::cout << "Address: " << ptr << ", Value: " << *ptr;  // Dereferencing
```

Dynamic Memory Allocation (new & delete)

- new is used to allocate memory dynamically at runtime.
- delete is used to free allocated memory, preventing memory leaks.

```
int* p = new int(20);  // Allocates memory for an integer
std::cout << "Value: " << *p;
delete p;  // Frees allocated memory
```

Pointers & Memory Management

```cpp
#include <iostream>

int main() {
    // Pointer to a variable (stores memory address)
    int x = 10;
    int* ptr = &x;  // Pointer stores the address of x

    std::cout << "Address of x: " << ptr << std::endl;  // Prints memory address
    std::cout << "Value of x using pointer: " << *ptr << std::endl<< std::endl;  // Dereferencing

    // Dynamic memory allocation
    int* p = new int(20);  // Allocates memory for an integer in heap
    std::cout << "Address of p: " << p << std::endl;
    std::cout << "Dynamically allocated value: " << *p << std::endl;

    // Free allocated memory
    delete p;

    return 0;
}
```

File Handling

Writing and Reading Files using ofstream and ifstream

File handling allows **reading from and writing to files** using file streams. The <fstream> library provides:

- ofstream (**output file stream**) – Used to write data to a file.
- ifstream (**input file stream**) – Used to read data from a file.

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("example.txt");  // Open file in write mode
    if (file) {
        file << "Hello, File Handling in C++!";  // Write data
        file.close();  // Close file
    }
    return 0;
}
```

Exception Handling in C++

Using try, catch, throw

```
try {

    int age = -1;

    if (age < 0)

     throw "Invalid Age!";

} catch (const char* msg) {

    cout << msg;

}
```

Object Oriented Programming in C++

OOP is a **programming paradigm** based on the concept of **objects**, which bundle data and behavior together. It makes code **modular, reusable, and scalable**.

Object Oriented Programming – As the name suggests uses **objects** in programming. **Object-oriented programming** aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Key Characteristics of Object-Oriented Programming (OOP)
Object-Oriented Programming (OOP) is based on fundamental concepts that serve as its building blocks:

Table of Contents

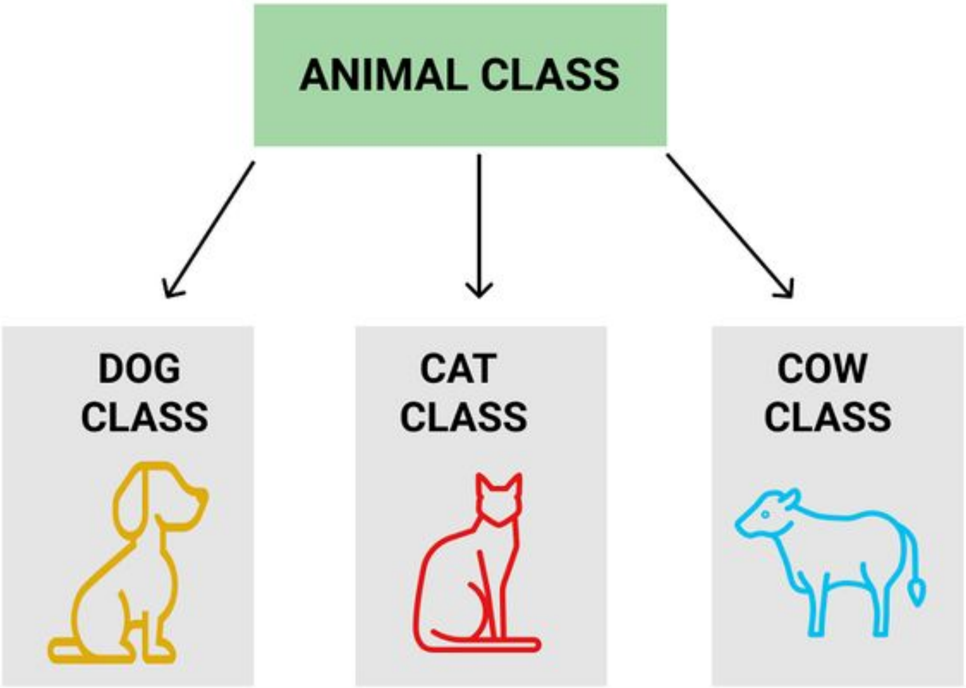
- Class
- Object
- Encapsulation
- Abstraction
- Polymorphism
- Inheritance

Classes and Objects

A **class** is a **blueprint** for creating objects. An **object** is an instance of a class.

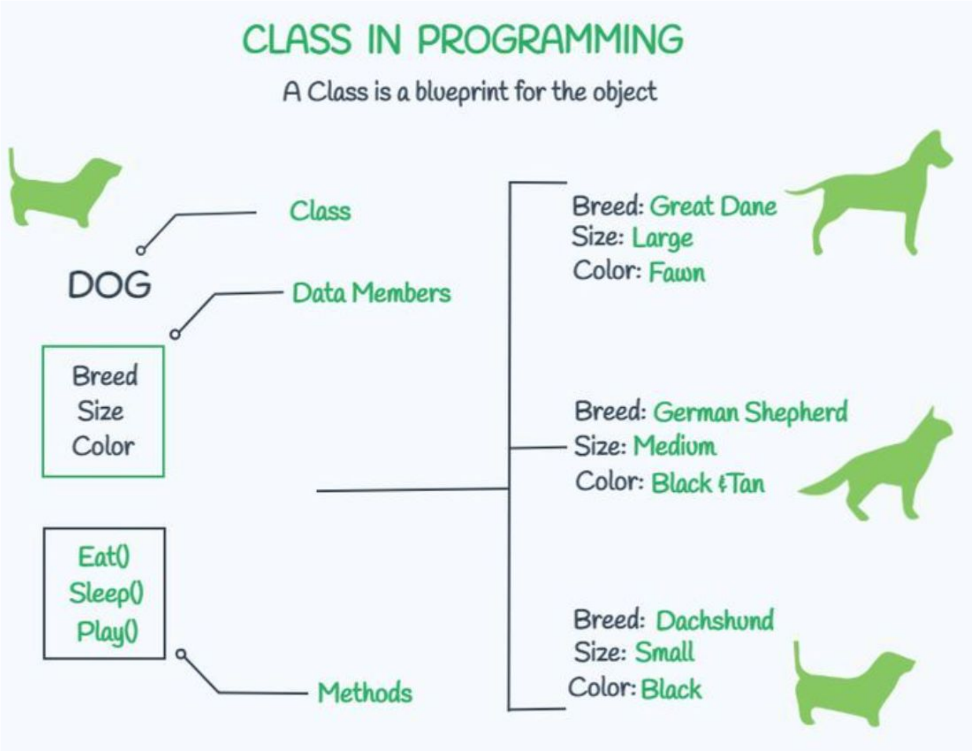
Classes

The fundamental building block of Object-Oriented Programming (OOP) in C++ is the **Class**. A class is a user-defined data type that serves as a blueprint for creating objects, which share common properties and behaviors. These properties are represented as **data members**, while behaviors are defined through **member functions**.



Classes and Objects

A **class** is a **blueprint** for creating objects. An **object** is an instance of a class.



Objects

An **Object** is a distinct, identifiable entity with specific characteristics and behaviors. In C++, an object is an **instance of a class**.

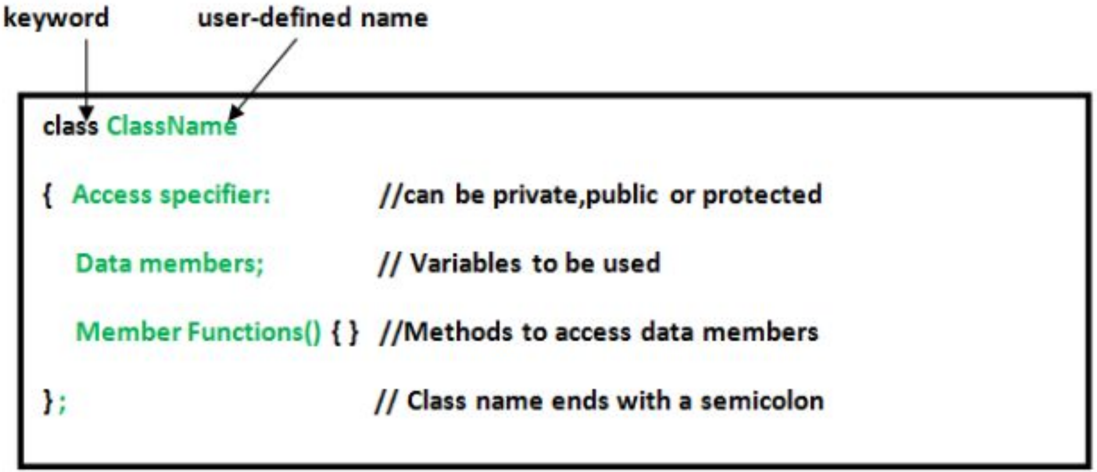
For example, the **Animal** class represents a general concept or category, but it does not exist as a tangible entity. However, a **black Dog named VoidShadowDarkFangReaper** is a real, specific animal that belongs to the **Animal** class. Similarly, **classes define concepts**, while **objects represent actual instances of those concepts**.

Classes and Objects

A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

Defining Class in C++

```
class ClassName {
    access_specifier:
    // Body of the class
};
```



Example

```
class ThisClass {
    public:
    int var;     // data member
    void print() {               // member method
        cout << "Hello";
     }
};
```

Classes and Objects

Class Definition (`Car`)

- The `Car` class has two **data members**:
  - `brand` (string) → Stores the car's brand.
  - `speed` (int) → Stores the car's speed in km/h.
- It also has a **member function** `showDetails()`, which prints the brand and speed.

Object Creation (`myCar`)

- In the `main()` function, an **object** `myCar` of class `Car` is created.
- The **brand** is set to `"Toyota"`, and **speed** is set to `180`.

Function Call (`showDetails()`)

- `myCar.showDetails();` prints the car's details.

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

class Car {
public:
    string brand;
    int speed;

    void showDetails() {
        cout << "Brand: " << brand << ", Speed: " << speed << " km/h"
<< endl;
    }
};

int main() {
    Car myCar;
    // class_name object_name  ->   Object of class Car
    myCar.brand = "Toyota";
    myCar.speed = 180;
    myCar.showDetails();  // Calling member function
    return 0;

}
```

Scope Resolution Operator

The **Scope Resolution Operator (`::`)** in C++ is used to define or access **global** and **class-specific** variables and functions when there is a naming conflict.

Accessing Global Variables When There is a Name Conflict

If a **local variable** has the same name as a **global variable**, `::` helps access the **global version**.

🔹 **Example:**

int x = 100; // Global variable

```
int main() {
    int x = 50;  // Local variable
    cout << "Local x: " << x << endl;      // Prints 50
    cout << "Global x: " << ::x << endl;   // Accesses global x (100)
    return 0;
}
```

Define Class Functions Outside the Class

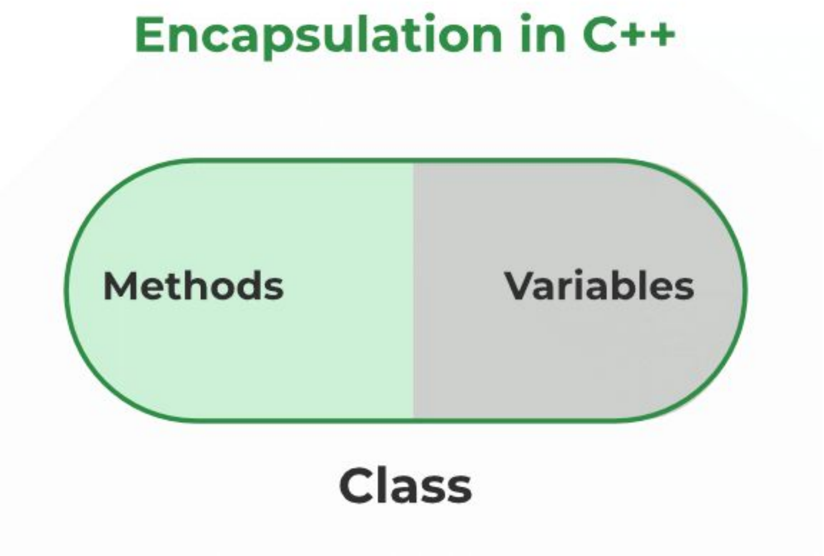
```
class Car {
public:
    void show();  // Function declaration
};

// Function definition outside the class
void Car::show() {
    cout << "This is a Car" << endl;
}

int main() {
    Car obj;
    obj.show();
    return 0;
}
```

Encapsulation

Encapsulation means **bundling data and methods** in a class and **restricting direct access** to data using **private members**.

Encapsulation is the process of bundling data and related functions into a single unit. In Object-Oriented Programming, it means binding data and the functions that operate on it within a **class**.

For example, in an **Animal** class, data members like `species`, `age`, and `name` are encapsulated along with member functions such as `eat()` and `sleep()`. Using **<u>access specifiers</u>** like `protected`, encapsulation helps restrict direct access to the class's data from outside, enhancing security and data integrity.



Encapsulation

Access Specifiers in C++

Access specifiers in C++ **control the visibility and accessibility** of class members (variables and functions). C++ provides three types of access specifiers:

1. Public (`public`)

- **Accessible from anywhere** (inside and outside the class).
- Used when class members should be available to other parts of the program.

2. Private (`private`)

- **Accessible only within the class** (not from outside).
- Used for **data hiding** to protect sensitive information.

3. Protected (`protected`)

- **Accessible within the class and its derived (child) classes**.
- Used when members should be hidden from outside but still accessible in derived classes.

Encapsulation

Class Definition (`BankAccount`)

- The class has a **private** member variable `balance`, which **cannot** be accessed directly from outside.
- It provides **public** functions:
  - `setBalance(double amount)`: Sets the balance.
  - `getBalance()`: Returns the balance.

Object Creation and Function Calls (`main()`)

- A `BankAccount` object `account` is created.
- `setBalance(5000);` sets the balance to `$5000`.
- `getBalance();` retrieves and prints the balance.

Why use `private`?

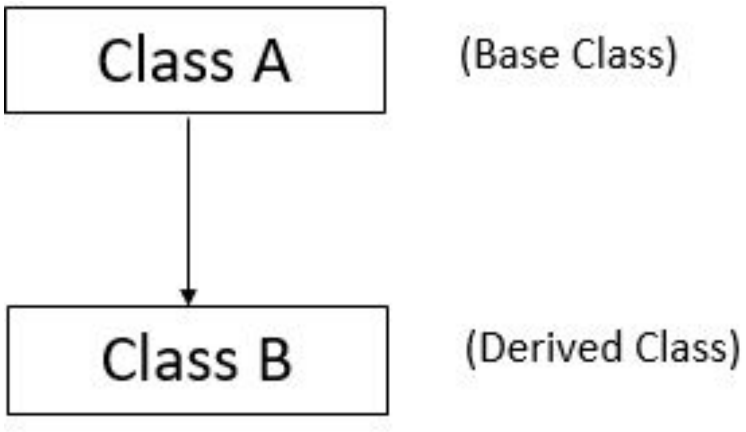
- Prevents direct modification of `balance`.
- Ensures data is accessed securely through controlled methods (`setBalance()` and `getBalance()`).

```
class BankAccount {

private:

    double balance;   // Private member

public:

    void setBalance(double amount) { balance = amount; }

    double getBalance() { return balance; }

};

int main() {

    BankAccount account;

    account.setBalance(5000);

    cout << "Balance: $" << account.getBalance() << endl;

    return 0;

}
```

Inheritance

Inheritance allows a **child (derived) class** to acquire properties and methods of a **parent (base) class**.

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.



Inheritance

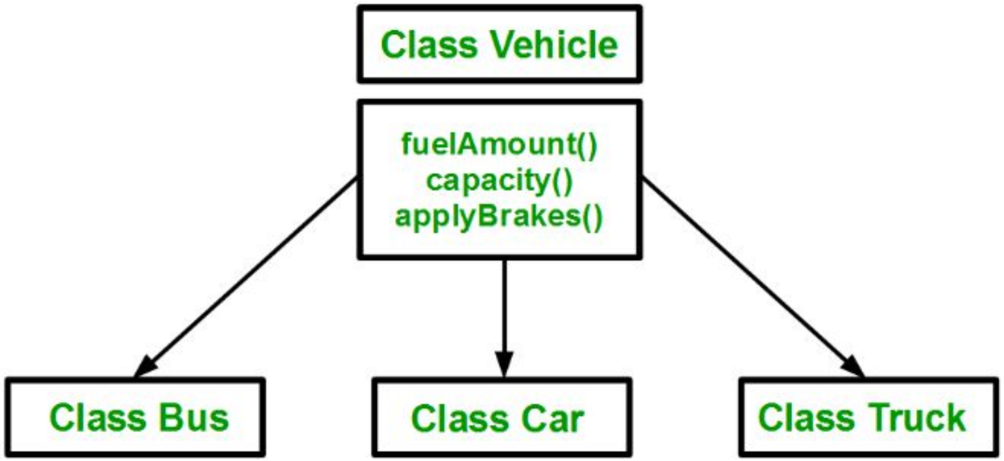
Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class: The class whose properties are inherited by a sub-class is called Base Class or Superclass.

Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Syntax for Inheritance in C++

```
class DerivedClass : access_specifier BaseCl

    // Members of the derived class

};
```



What Is the Use of `"access specifier"` in Inheritance?

In C++, `"class Child : public Parent"` is a way to **inherit properties and behaviors** from a base class (`Parent`) into a derived class (`Child`).

Purpose of `"public Parent"` in Inheritance:

- Allows **code reuse**: The `Child` class can use functions and variables of `Parent`.
- Establishes a **"is-a" relationship**: If `Child` inherits from `Parent`, then `Child` **is a** `Parent`.
- Supports **polymorphism**: Enables method overriding and dynamic behavior.

```
class Parent {
public:
    void show() { cout << "This is Parent class" << endl; }
};
//DerivedClass  : access_specifier BaseClass
class Child : public Parent { };  // Child class inherits Parent class

int main() {
    Child obj;
    obj.show();  // Child object can access Parent's function
    return 0;
}
```

Can We Use `private` and `protected` Instead of `public`?

Yes! The access specifier (`public`, `protected`, `private`) **affects how members of the base class are inherited** in the derived class.

Public Inheritance (`class Child : public Parent`)

- **"Is-a" relationship** is maintained (Child **is a** Parent).
- `public` members of `Parent` stay **public** in `Child`.
- `protected` members of `Parent` stay **protected** in `Child`.
- `private` members of `Parent` **are not accessible** in `Child`.

```
class Parent {
public:
    int a;   // Public member
protected:
    int b;   // Protected member
private:
    int c;   // Private member
};

class Child : public Parent {
public:
    void show() {
        cout << a;   // Allowed (public stays
public)
        cout << b;   // Allowed (protected stays
protected)
        // cout << c;   // Not accessible (private
members not  inherited)
    }
};
```

Can We Use `private` and `protected` Instead of `public`?

Protected Inheritance (`class Child : protected Parent`)

- **"Is-a" relationship is weakened** (Child **is not necessarily a** Parent).
- `public` members of `Parent` become **protected** in `Child`.
- `protected` members of `Parent` remain **protected** in `Child`.
- `private` members of `Parent` **are not accessible** in `Child`.

```
class Child : protected Parent {
public:
    void show() {
        cout << a;  // Allowed (public becomes protected)
        cout << b;  // Allowed (protected remains protected)
        // cout << c;  // Not accessible (private members not inherited)
    }
};

int main() {
    Child obj;
    // obj.a = 10;  // Error! 'a' is protected in Child
}
```

Why Can't `obj.a = 10;` Be Accessed in `main()`?

Even though `show()` is **public** in `Child`, `a` (which was `public` in `Parent`) becomes **protected** due to **protected inheritance**.

- The `public` members of `Parent` become protected in `Child`.
- The `protected` members of `Parent` remain protected in `Child`.
- The `private` members of `Parent` are not inherited at all.

Thus, in `Child`, `a` is now **protected**, meaning it can **only be accessed within `Child` or its subclasses**, but **not from `main()` or any other external function**.

How `show()` Can Access `a`?

Even though `a` is protected, **it is still accessible inside the `Child` class**. Since `show()` is a member function of `Child`, it has permission to access `protected` members of `Parent`.

```
class Parent {
public:
    int a;  // Public in Parent
protected:
    int b;  // Protected in Parent
private:
    int c;  // Private in Parent (Not inherited)
};

class Child : protected Parent {
public:
    void show() {
        cout << a;  // ✅ Allowed: a is protected in Child, accessible inside class
        cout << b;  // ✅ Allowed: b is already protected, so accessible inside Child
        // cout << c;  // ❌ Error: c is private, not inherited
    }
};

int main() {
    Child obj;
    obj.show();  // ✅ Allowed: show() is public in Child, so can be called
    // obj.a = 10;  // ❌ Error: 'a' is protected in Child, not accessible outside
}
```

Types of Inheritance

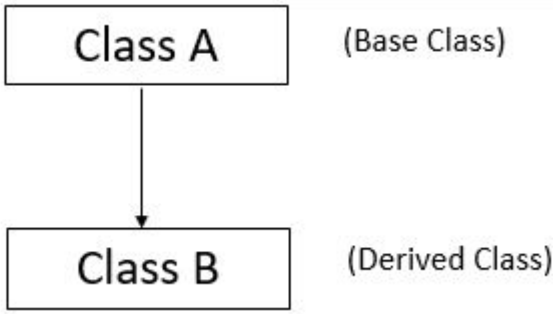


Fig: Single Inheritance

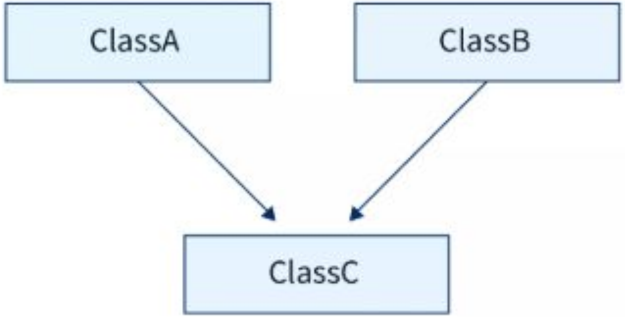


Fig: Multiple Inheritance

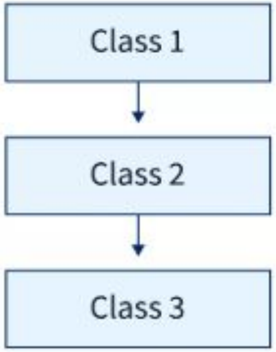


Fig: Multilevel Inheritance

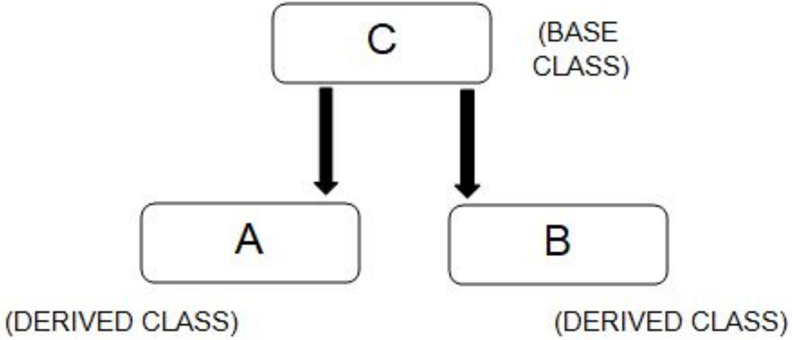


Fig: Hierarchical Inheritance

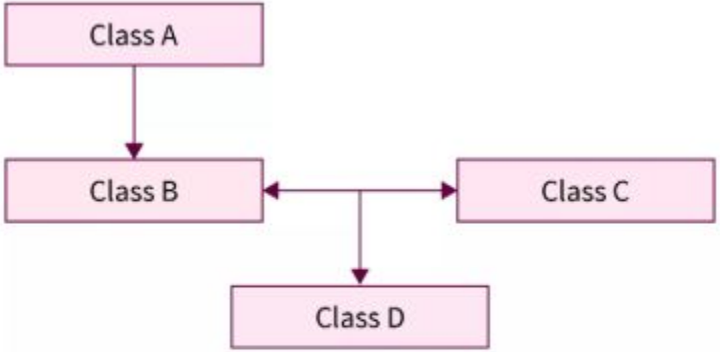


Fig: Hybrid Inheritance

Single Inheritance

A **single derived class** inherits from a **single base class**.

Allows **code reuse** and extension of functionality.

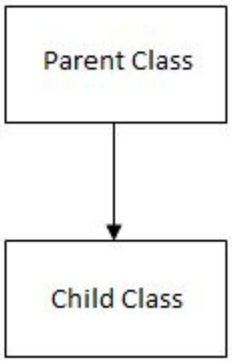


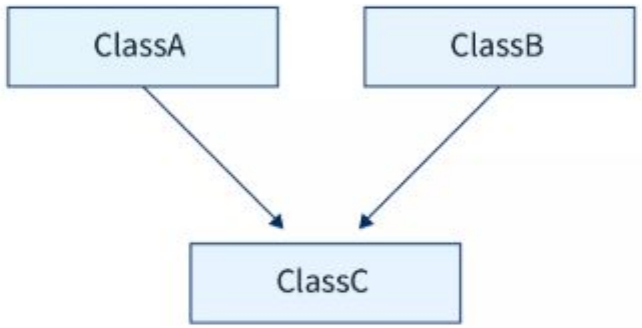
Fig: Single inheritance

```
#include <iostream>

using namespace std;

class Parent {

public:

    void show() { cout << "Parent class" << endl; }

};


class Child : public Parent { };  // Child inherits from Parent


int main() {

    Child obj;

    obj.show();  // Inherited function

    return 0;

}
```

Multiple Inheritance

A **single derived class** inherits from **multiple base classes**.

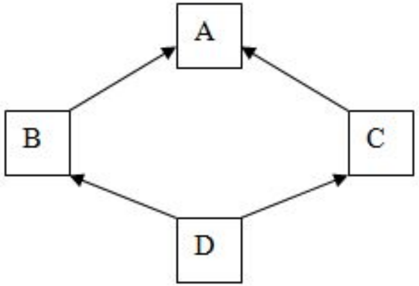
Allows a class to have features from multiple sources.




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

class B {
public:
    void showB() {
    cout << "Class B" << endl;
    }
};

class C : public A, public B { };  // Multiple
Inheritance

int main() {
    C obj;
    obj.showA();
    obj.showB();
    return 0;
}
```

Multiple Inheritance

One common problem with multiple inheritance is called the "diamond problem." It happens when a class inherits from two other classes that share a common parent class. If both of these parent classes have changed the same method or attribute, it creates confusion, because the system doesn't know which version to use




```
class A {
public:
    void display() {
        cout << "Class A" << endl;    }     };

class B : public A {
public:
    void display() {
        cout << "Class B" << endl;    }     };

class C : public A {
public:
    void display() {
        cout << "Class C" << endl;    }     };

class D : public B, public C {
    // D will inherit from both B and C, both of which
inherit from A
};
int main() {
    D d;
    d.display();  // Ambiguous call to display()
    return 0;
}
```

Multilevel Inheritance

A **derived class** acts as a **base class** for another class.

Forms a **chain of inheritance**.

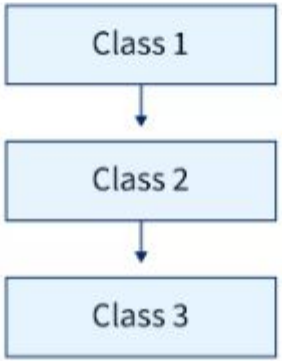


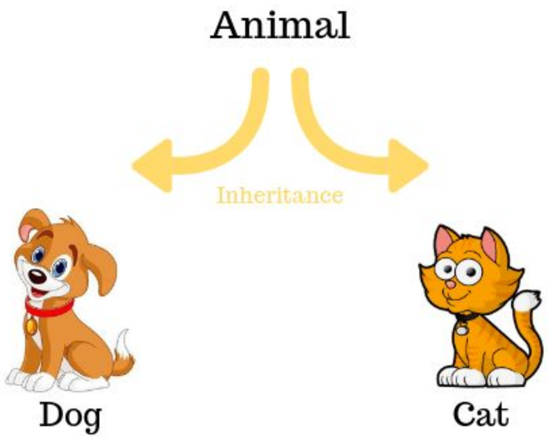
Fig: Multilevel Inheritance

```
class Grandparent {
public:
    void show() { cout << "Grandparent class" <<
endl; }
};

class Parent : public Grandparent { };

class Child : public Parent { };

int main() {
    Child obj;
    obj.show();  // Inherited from Grandparent
    return 0;
}
```

Hierarchical Inheritance

Multiple derived classes inherit from a **single base class**.

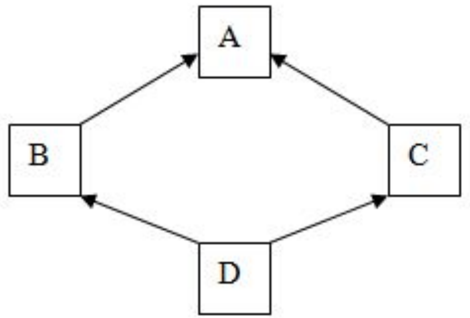
Useful when multiple classes share common functionality.




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

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

class Cat : public Animal {
public:
    void meow() { cout << "Cat meows" << endl; }
};

int main() {
    Dog d;
    d.sound();  // Inherited from Animal
    d.bark();

    Cat c;
    c.sound();  // Inherited from Animal
    c.meow();

    return 0;
}
```

Hybrid Inheritance

Combination of multiple inheritance types, using **virtual** to prevent duplication.




```
class A {
public:
    void show() { cout << "Class A" << endl; }
};

class B : virtual public A { };  // Virtual Inheritance
class C : virtual public A { };

class D : public B, public C { };

int main() {
    D obj;
    obj.show();  // Resolves ambiguity
}
```

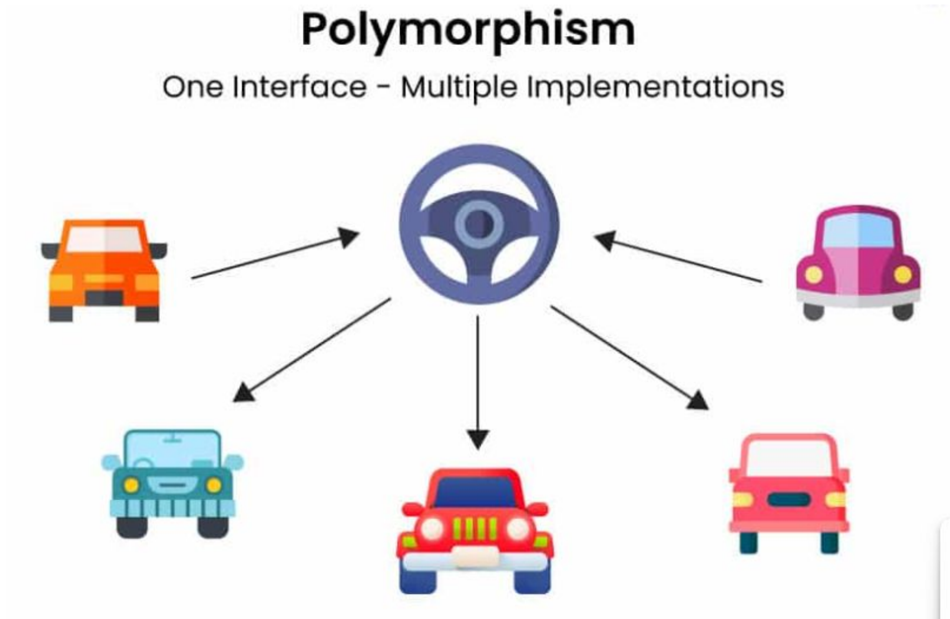
Polymorphism

Polymorphism means **"many forms"** and allows a single function or object to behave in different ways. It helps achieve **code reusability** and **flexibility** in Object-Oriented Programming (OOP).

There are **two types of polymorphism in C++**:

1. **Compile-time (Static) Polymorphism** – Achieved using **Function Overloading & Operator Overloading**.

2. **Run-time (Dynamic) Polymorphism** – Achieved using **Method Overriding (with Virtual Functions)**.



Polymorphism

Compile-Time (Static) Polymorphism

1. Function calls are resolved at **compile time**.
2. Achieved through **Function Overloading** and **Operator Overloading**.

Function Overloading

Multiple functions with the **same name** but **different parameters**.

Function Overloading is when you have multiple functions with the same name but different parameters (either in number, type, or both). This allows you to perform similar operations in different ways, depending on the arguments passed to the function.

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

class Math {
public:
    int add(int a, int b)
     { return a + b; }
    double add(double a, double b)
     { return a + b; }  // Different parameter
types
};

int main() {
    Math obj;
    cout << obj.add(5, 3) << endl;
// Calls int version → Output: 8
    cout << obj.add(4.2, 2.3) << endl;
// Calls double version → Output: 6.5
}
```

Polymorphism

Compile-Time (Static) Polymorphism

1. Function calls are resolved at **compile time**.
2. Achieved through **Function Overloading** and **Operator Overloading**.

Operator Overloading

Operator Overloading allows you to redefine the behavior of operators (like `+`, `-`, `*`, `==`, etc.) for user-defined types (such as classes or structs). This lets you use operators on objects of your own classes in a way that makes sense for that class, similar to how they work for built-in types.

```
class Complex {
private:
    float real;
    float imag;

public:
    // Constructor to initialize the complex number
    Complex(float r, float i) : real(r), imag(i) {}

    // Overloading the + operator to add two Complex numbers
    Complex operator + (const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    // Function to display the complex number
    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex num1(3.0, 4.0);  // 3 + 4i
    Complex num2(1.5, 2.5);  // 1.5 + 2.5i

    Complex sum = num1 + num2;  // Calls the overloaded +
operator

    sum.display();  // Displays the result of the addition

    return 0;
}
```

Polymorphism

Run-Time (Dynamic) Polymorphism

1. Function calls are resolved at **runtime** using **virtual functions**.
2. Allows method overriding (redefining a function in the derived class).

Method Overriding (Using Virtual Functions)

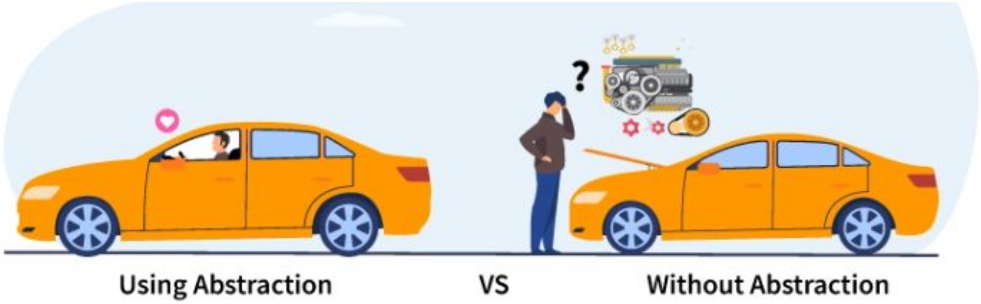
Method Overriding is a feature in object-oriented programming (OOP) where a derived class provides its own implementation of a method that is already defined in its base class. The key here is that the base class function is declared as **virtual**. This allows the derived class's version of the method to be called even if the object is being referenced through a pointer or reference to the base class. This behavior is known as **runtime polymorphism**.

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

class Dog : public Animal {
public:
    void sound() override { cout << "Dog barks" << endl; }  // Overriding base function
};

int main() {
    Animal* a;
    Dog d;
    a = &d;   // Base class pointer pointing to derived class object
    a->sound();  // Calls Dog's sound() due to virtual function → Output: Dog barks
}
```

Abstraction

Abstraction is one of the key principles of **Object-Oriented Programming (OOP)**. It allows you to **hide implementation details** and only show the necessary features of an object.

- **Why is Abstraction Important?**
  - Reduces complexity by exposing only essential details.
  - Prevents direct access to sensitive data.
  - Improves maintainability and security.
- How to Achieve Abstraction in C++?
  In C++, **abstraction** is achieved using:
  - **Abstract Classes (with Pure Virtual Functions)**
  - **Encapsulation (Using Access Specifiers: `private`, `protected`)**



Abstraction

Abstraction Using Encapsulation

- **Data hiding** is a form of abstraction.
- **Private members** restrict direct access to data.
- **Only public methods** expose controlled access.


```
class BankAccount {
private:
    double balance;  // Hidden data

public:
    BankAccount(double initialBalance) { balance =
initialBalance; }

    void deposit(double amount) { balance += amount; }

    void withdraw(double amount) {
        if (amount <= balance) balance -= amount;
        else cout << "Insufficient balance!" << endl;
    }

    double getBalance() { return balance; }  //
Controlled access
};

int main() {
    BankAccount account(5000);
    account.deposit(1000);
    account.withdraw(2000);

    cout << "Balance: $" << account.getBalance() << endl;
// Output: Balance: $4000
}
```

Constructors

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

Key Features of a Constructor

- Same name as the class
- No return type (not even `void`)
- Called automatically when an object is created
- Can be overloaded (multiple constructors with different parameters)

Types of Constructors in C++

- Default Constructor (No Parameters)
- Parameterized Constructor (With Parameters)
- Copy Constructor (Copies an Object)
- Constructor Overloading (Multiple Constructors in the Same Class)
- Dynamic Constructor (Using `new`)

Default Constructor (No Parameters)

If no constructor is defined, C++ provides a default constructor that initializes variables with garbage values.

```
class Car {
public:
    Car() {  // Default Constructor
        cout << "Car is created!" << endl;
    }
};

int main() {
    Car myCar;  // Constructor is automatically called
    return 0;
}
```

Parameterized Constructor

A **constructor with parameters** allows us to **initialize values when an object is created**.

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

    Car(string b, int s) {  // Parameterized Constructor
        brand = b;
        speed = s;
    }

    void show() {
        cout << "Brand: " << brand << ", Speed: " << 
speed << " km/h" << endl;
    }
};

int main() {
    Car car1("Toyota", 180);  // Passing values to 
constructor
    car1.show();
    return 0;
}
```

Copy Constructor (Copies One Object to Another)

A **copy constructor** is used to create a **new object** as a **copy of an existing object**

```
class Car {
public:
    string brand;

    Car(string b) {  // Parameterized Constructor
        brand = b;
    }

    Car(const Car &c) {  // Copy Constructor
        brand = c.brand;
    }

    void show() { cout << "Brand: " << brand << endl; }
};

int main() {
    Car car1("BMW");   // Original Object
    Car car2 = car1;   // Copy Constructor Called

    car1.show();
    car2.show();
    return 0;
}
```

Constructor Overloading

You can have **multiple constructors** in a class with **different parameters**.

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

    // Default Constructor
    Car() { brand = "Unknown"; speed = 0; }

    // Parameterized Constructor
    Car(string b, int s) { brand = b; speed = s; }

    void show() { cout << "Brand: " << brand << ", Speed: 
" << speed << " km/h" << endl; }
};

int main() {
    Car car1;                   // Calls Default Constructor
    Car car2("Audi", 200);  // Calls Parameterized 
Constructor

    car1.show();
    car2.show();
    return 0;
}
```

Dynamic Constructor (Using new)

A **dynamic constructor** allocates memory **at runtime** using `new`.

```
class Car {
private:
    int* speed;
public:
    Car(int s) {  // Dynamic Constructor
        speed = new int;
        *speed = s;
    }

    void show() { cout << "Speed: " << *speed << " km/h"
<< endl; }

    ~Car() {  // Destructor to free memory
        delete speed;
        cout << "Memory freed!" << endl;
    }
};

int main() {
    Car car1(150);
    car1.show();
    return 0;
}
```

Destructor

A destructor is a special member function that is automatically called when an object is destroyed. It is mainly used to release resources (e.g., memory allocated using `new`, file handles, database connections, etc.).

Key Features of a Destructor

- Same name as the class but with a ~ (tilde) prefix
- No return type (not even `void`)
- No parameters (cannot be overloaded)
- Automatically called when an object goes out of scope or `delete` is used

Syntax of a Destructor

```
class ClassName {
public:
    ~ClassName() {
        // Destructor Code
    }
};
```

Example: Destructor Without Dynamic Memory

Destructor is called automatically when `myCar` goes out of scope.

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

class Car {
public:
    Car() { cout << "Car is created!" << endl; }
    ~Car() { cout << "Car is destroyed!" << endl; }
};

int main() {
    Car myCar;  // Constructor is automatically called
    return 0;   // Destructor is called when program
exits
}
```

Destructor with Dynamic Memory (new and delete)

If we use `new` inside a constructor, we must use `delete` in the destructor to **free memory**.

If a base class has a virtual function, its destructor should be `virtual` to avoid memory leaks.

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

class Car {
private:
    int* speed;
public:
    Car(int s) {  // Constructor
        speed = new int;  // Dynamic memory allocation
        *speed = s;
        cout << "Car created with speed: " << *speed << "
km/h" << endl;
    }

    ~Car() {  // Destructor
        delete speed;  // Free allocated memory
        cout << "Memory freed! Car is destroyed!" << endl;
    }
};

int main() {
    Car* car1 = new Car(200);  // Create object dynamically
    delete car1;  // Destructor is called
    return 0;
}
```

Manual Call Can Cause Errors

Even though destructors are **automatically** called, manually calling them can cause **unexpected behavior**.

Why is this a problem?

- If you call a destructor explicitly, it does **not free memory allocated using `new`**, leading to **double deletion** issues.
- Objects allocated dynamically **must be deleted using `delete`**, not by calling the destructor directly.

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

class Car {
public:
    ~Car() { cout << "Car destroyed!" << endl; }
};

int main() {
    Car myCar;
    myCar.~Car();  // Manually calling destructor (bad
practice)
    return 0;
}
```

Virtual Destructors Required for Proper Inheritance

If a base class destructor is **not virtual**, deleting a derived class object through a base class pointer can cause **memory leaks**.

Why is this a problem?

- The base class destructor gets called, but the derived class destructor does **not** get called.
- This leads to **incomplete cleanup**, causing **resource leaks**.

```
class Vehicle {
public:
    ~Vehicle() { cout << "Vehicle destroyed!" << endl; }
// ❌ Not virtual
};

class Car : public Vehicle {
public:
    ~Car() { cout << "Car destroyed!" << endl; }
};

int main() {
    Vehicle* v = new Car();
    delete v;  // Car's destructor is NOT called, causing
memory leaks!
    return 0;
}
```

Destructor Order in Multiple Inheritance Can Be Tricky

When using **multiple inheritance**, destructors **may not be called in the expected order**, leading to **dangling pointers** or **resource leaks**.

Why is this a problem?

- The destructor order depends on the **order of inheritance**, which can cause **unexpected behavior** if not managed properly.

```
class A {
public:
    ~A() { cout << "Destructor of A" << endl; }
};

class B {
public:
    ~B() { cout << "Destructor of B" << endl; }
};

class C : public A, public B {
public:
    ~C() { cout << "Destructor of C" << endl; }
};

int main() {
    C obj;  // What is the order of destructor calls?
    return 0;
}
```

Exception Handling

Exception handling is a mechanism in C++ that allows a program to handle **runtime errors** gracefully instead of crashing. It enables the program to detect and respond to unexpected situations (e.g., division by zero, invalid memory access, file errors).

Why Use Exception Handling?

- Prevents program crashes
- Separates error-handling code from normal code
- Makes code cleaner and more readable
- Allows centralized error management

Types of C++ Exception

- **Synchronous:** Exceptions that occur when something goes wrong due to a mistake in the input data or when the program is not equipped to handle the current type of data it's processing, such as dividing a number by zero.
- **Asynchronous**: Exceptions that are beyond the program’s control, such as disk failures, keyboard interrupts, and other external factors.

Basic Syntax of Exception Handling

Syntax of try-catch

```
try {
    // Code that might throw an exception
   throw SomeExceptionType("Error message");
}
catch( ExceptionName e1 )  {
// catch block catches the exception that is thrown from try block
}
```

try:
The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception

throw:
An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

catch:
The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

Basic Syntax of Exception Handling

Handling an Integer Exception

Explanation:

- The `try` block executes normally until it encounters `throw 404;`.
- The `throw` statement raises an **integer exception**.
- The `catch (int errorCode)` block catches the exception and prints the error code.

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

int main() {
    try {
        cout << "Inside try block" <<
endl;
        throw 404;  // Throwing an
integer exception
    }
    catch (int errorCode) {
        cout << "Exception caught!
Error code: " << errorCode << endl;
    }

    return 0;
}
```

Basic Syntax of Exception Handling

Handling a String Exception

- Here, the `throw` statement throws a string (`const char*`), which is caught and displayed.

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

int main() {
    try {
        throw "An error occurred!";
    }
    catch (const char* msg) {
        cout << "Exception: " << msg
<< endl;
    }

    return 0;
}
```

Basic Syntax of Exception Handling

Multiple Catch Blocks

Different `catch` blocks handle different exception types.

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

int main() {
    try {
        throw 3.14;  // Throwing a double
exception
    }
    catch (int e) {
        cout << "Caught an integer: " << e <<
endl;
    }
    catch (double e) {
        cout << "Caught a double: " << e <<
endl;
    }

    return 0;
}
```

Basic Syntax of Exception Handling

Catching All Exceptions (catch(...))

`catch(...)` catches any type of exception, useful when you don’t know what exception might be thrown.

`try` → Code that may throw an exception.
`throw` → Raises an exception.
`catch` → Handles the exception.
Multiple `catch` blocks → Handle different exception types.
`catch(...)` → Catches all exceptions

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

int main() {
    try {
        throw 42;  // Throwing an integer
    }
    catch (...) {
        cout << "Caught an unknown
exception!" << endl;
    }

    return 0;
}
```

Limitations of Exception Handling in C++

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It’s hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.