C++

Akash Patil

Topics

- Operators & Expressions
- Conditional and Looping Statements
- Functions in C++
- Memory Management and Pointers
- OOP Concepts using C++
- Constructor and Destructor
- Inheritance

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**.

What is OOP?

- Programming paradigm based on “objects”
- Combines data + functions into single unit (class)
- Solves limitations of POP

Salient Features of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Data Hiding
- Modularity
- Reusability

Feature	OOP (C++)	POP (C)
Approach	Object-centered (focus on data + behavior)	Function-centered (focus on procedures)
Data Security	Data is hidden using encapsulation (private/protected)	Data is exposed globally and can be accessed freely
Reusability	High, via classes, inheritance, polymorphism	Low, functions need to be rewritten
Modularity	Program organized into objects (classes)	Program organized into functions
Data Handling	Data is tied to objects	Data flows freely between functions
Extensibility	Easy to extend through classes and inheritance	Harder to extend; changes may affect entire program
Polymorphism	Supported (overloading, overriding)	Not supported
Examples	C++, Java, Python (OOP)	C, Fortran, Pascal (POP)

C, Fully POP

Reason:

- C is **procedure-oriented programming (POP)**.
- Focus is on **functions (procedures)** and **flow of execution**, not on data.
- Data is mostly **global** and can be accessed by any function.
- No concept of **class, object, inheritance, or polymorphism**.
- Hence, **C is purely POP**.

C++, OOP (but not 100% pure OOP)

Reason:

- C++ introduced **classes, objects, inheritance, polymorphism, encapsulation, abstraction**, etc.
- It **supports both POP and OOP**. You can still write a **C-style program** in C++ (without classes).
- Example: You can write a simple `printf("Hello")` program in C++ without using objects.
- Therefore:
  - It is called **multi-paradigm** (supports both POP and OOP).
  - Not a **fully pure OOP language**, because it still allows global variables and standalone functions.

Java – Pure OOP? or Partial?

Java – Pure OOP? or Partial?

Reason:

- Java was designed as **Object-Oriented first**:
  1. Everything in Java is part of a **class** (even `main` is inside a class).
  2. Supports all **OOP features** → Encapsulation, Inheritance, Polymorphism, Abstraction.
- However, Java is **not 100% pure OOP** because:
  1. **Primitive Data Types (int, char, float, etc.)** exist → not objects.
     - Though Java introduced **wrapper classes (Integer, Character, Float, etc.)** in `java.lang` to treat them as objects, primitives still exist.
  2. **Static methods** (like `main`) don't need objects.
  3. Uses **operators** (`+`, `-`, etc.) which are not objects.

Hence:

- Java is **"almost pure OOP"** (closer to full OOP than C++).
- But since primitives exist, it is considered **not 100% pure OOP**.

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`

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.

Compilation

```
g++ FileName.cpp -o FileName
```

Steps in C++ Program Completion

1. Editing / Writing the Code

- You write the source code in a `.cpp` file.

Example:

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

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

Steps in C++ Program Completion

2. Preprocessing

- The **preprocessor** handles all lines starting with #.
- Tasks performed:
  - Include header files (#include <iostream>)
  - Replace macros (#define)
  - Remove comments
- Output: **expanded source code** (still human-readable).

Steps in C++ Program Completion

3. Compilation

- The compiler translates preprocessed code into **assembly language** for your CPU.
- Syntax errors (missing ;, wrong types, etc.) are caught here.
- Output: **object file (.o / .obj)** → machine code, but not yet executable.

Steps in C++ Program Completion

4. Assembly

- The assembler converts assembly code into **binary machine instructions**.
- Output: Object file (contains compiled functions but still incomplete).

Steps in C++ Program Completion

5. Linking

- The linker combines all object files and libraries.
- Resolves external references (e.g., cout comes from the standard C++ library).
- Output: **executable file (.exe in Windows, a.out or custom name in Linux).**

Steps in C++ Program Completion

6. Loading

- The OS **loader** loads the executable into memory.
- Allocates space for code, data, stack, heap.
- Prepares the program to run.

Steps in C++ Program Completion

7. Execution

- The program starts execution from the main() function.
- Runs until return 0; or termination.

Steps in C++ Program Completion

Summary Flow

Source Code (.cpp)

↓ **Preprocessor**

Expanded Source Code

↓ **Compiler**

Assembly Code

↓ **Assembler**

Object Code (.o / .obj)

↓ **Linker**

Executable (.exe / a.out)

↓ **Loader**

Program Runs

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;
}
```

Namespaces in C++

- A namespace groups related code (variables, functions, classes) under a named scope.
- Helps avoid naming conflicts in large projects or when combining code from multiple libraries.
- Common in standard libraries, e.g., std in C++.
- Access members with the scope resolution operator ::

  (e.g., std::cout)

Understanding the std Namespace in C++

cout, cin, endl, etc., are part of the C++ Standard Library.

These components belong to the **std namespace** (short for **standard**).

Normally, you must prefix them with std::
(e.g., std::cout, std::cin)

To simplify code, use:

```
using namespace std;
```

Namespaces Example

```
#include <iostream>

namespace MyNamespace {
  int x = 42;
}

using namespace MyNamespace;
using namespace std;

int main() {
  cout << x;  // No need to write
MyNamespace::x
  return 0;
}
```

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;
}
```

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.

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;

}
```

```
*java

class Main {

    public static void main(String[] args) {

        int i;

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

        {

            System.out.println(i);

        }

    }

}
```

What is the output?

```
int a = 5, b = 10;

if (a > b)

    if (a > 0)

        cout << "Case 1";

else

    cout << "Case 2";
```

What is the output?

```
int x = 0;

if (x = 5)

    cout << "True";

else

    cout << "False";
```

What is the output?

```
int main() {

    if (false)

        cout << "Hello";

    else if (false)

        cout << "World";

    else

        if (false);

        else

            cout << "Tricky!";

    return 0;

}
```

What is the output?

```
#include <iostream>

using namespace std;



int main() {

    for (int i = 0; i < 5; cout << i << " ")

        i++;

    return 0;

}
```

1. Write a program in C++ to reverse a number (e.g., 123 → 321)

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

int main() {
    int num, reversed = 0;

    cout << "Enter a number: ";
    cin >> num;

    while (num != 0) {
        int digit = num % 10;         // Extract last digit
        reversed = reversed * 10 + digit; // Append digit to reversed
        num = num / 10;               // Remove last digit
    }
    cout << "Reversed number = " << reversed << endl;
    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;
}
```

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 **contiguous 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;
}
```

Strings

Null Terminator

- **`char`** → no null terminator needed, since it's only **one character**.
- **`std::string`** → internally uses `'\0'` (null terminator) for compatibility with **C-style strings** (`char[]`)

```
    char ch = 'X';      // just one character
    char arr[] = "Hi";
// C-style string (has hidden '\0' at end)
    string str = "Hello";
// std::string (internally adds '\0')
```

`arr[] = "Hi";` → actually stored as `{ 'H', 'i', '\0' }`.

`std::string` also stores characters with an internal `'\0'` but you don't manage it yourself.

Strings

Storage

- `char` → can hold only **one character** (takes 1 byte).
- `std::string` → can hold **many characters** (length is dynamic, grows/shrinks as needed).

```
int main() {
    char ch = 'A';    // Only one character
    string str = "Hello"; // Multiple characters

    cout << "Char: " << ch << endl;
          // Output: A
    cout << "String: " << str << endl;
          // Output: Hello

    // String can grow
    str += " World";
    cout << "After append: " << str << endl;
          // Output: Hello World

    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

```
#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;
}
```

Pointer syntax:

```
int a = 10;      // normal variable
int* p = &a;     // p is a pointer to int, stores address of a
```

int* p declares a pointer to an int.

&a gets the **address** of variable a.

p now holds the memory address of a.

Accessing data via pointer:

```
cout << *p << endl;  // Output: 10
```

The * operator is the dereference operator: it accesses the value at the address stored in the pointer.

Common pointer operations:

Operation	Example	Description
Declare pointer	`int* p;`	Declares a pointer to int
Assign address	`p = &a;`	Stores address of `a` in `p`
Dereference pointer	`*p = 20;`	Sets value at the address pointed by `p` to 20
Pointer arithmetic	`p++`	Moves pointer to next int location

Important notes:

Pointer must be initialized before use, or it may point to garbage (undefined behavior).

```
int* p;  // uninitialized pointer (dangerous)
```

Use nullptr to initialize pointers with no target.

```
int* p = nullptr;
```

Example:

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

int main() {
    int x = 42;
    int* p = &x;

    cout << "Value of x: " << x << endl;        // 42
    cout << "Address of x: " << &x << endl;
    cout << "Pointer p points to address: " << p << endl;
    cout << "Value pointed by p: " << *p << endl;  // 42

    *p = 100;  // change value via pointer
    cout << "New value of x: " << x << endl;        // 100

    return 0;
}
```

Why use pointers?

To efficiently manage memory.

To work with dynamic memory allocation (new, delete).

To pass large data structures or objects efficiently (by reference).

To implement data structures like linked lists, trees, etc.

Pointer Arithmetic in C++

What is pointer arithmetic?

- When you add or subtract an integer to/from a pointer, it moves by **that many elements** in memory, **not bytes**.
- The step size depends on the type the pointer points to.

```
int arr[] = {10, 20, 30, 40};
int* p = arr;     // points to arr[0]

cout << *p << endl;    // 10

p = p + 2;
// moves pointer forward by 2 ints

cout << *p << endl;    // 30
```

Pointer arithmetic rules:

Expression	Result
`p + 1`	points to next element (next memory address by sizeof(type))
`p - 1`	points to previous element
`p2 - p1`	number of elements between two pointers

Pointer to Pointer

What is a pointer to a pointer?

- A pointer that **stores the address of another pointer**.

```
int x = 10;

int* p = &x;        // p points to x

int** pp = &p;      // pp points to p


cout << **pp << endl;  // prints 10
```

```
int x = 10;
int* p = &x;        // p points to x
int** pp = &p;      // pp points to p

cout << **pp << endl;  // prints 10
```

Explanation:

- pp holds the address of p.
- *pp dereferences pp to get p.
- **pp dereferences p to get x.

Dynamic Memory Allocation with Pointers

Why dynamic memory?

- When size or lifetime of data is unknown at compile time, you use dynamic memory.

How to allocate and free memory:

```
int* p = new int;
// allocate memory for one int
*p = 42;                    // assign value
cout << *p << endl;  // 42

delete p;                   // free memory
```

Why do we need double pointers (**)?

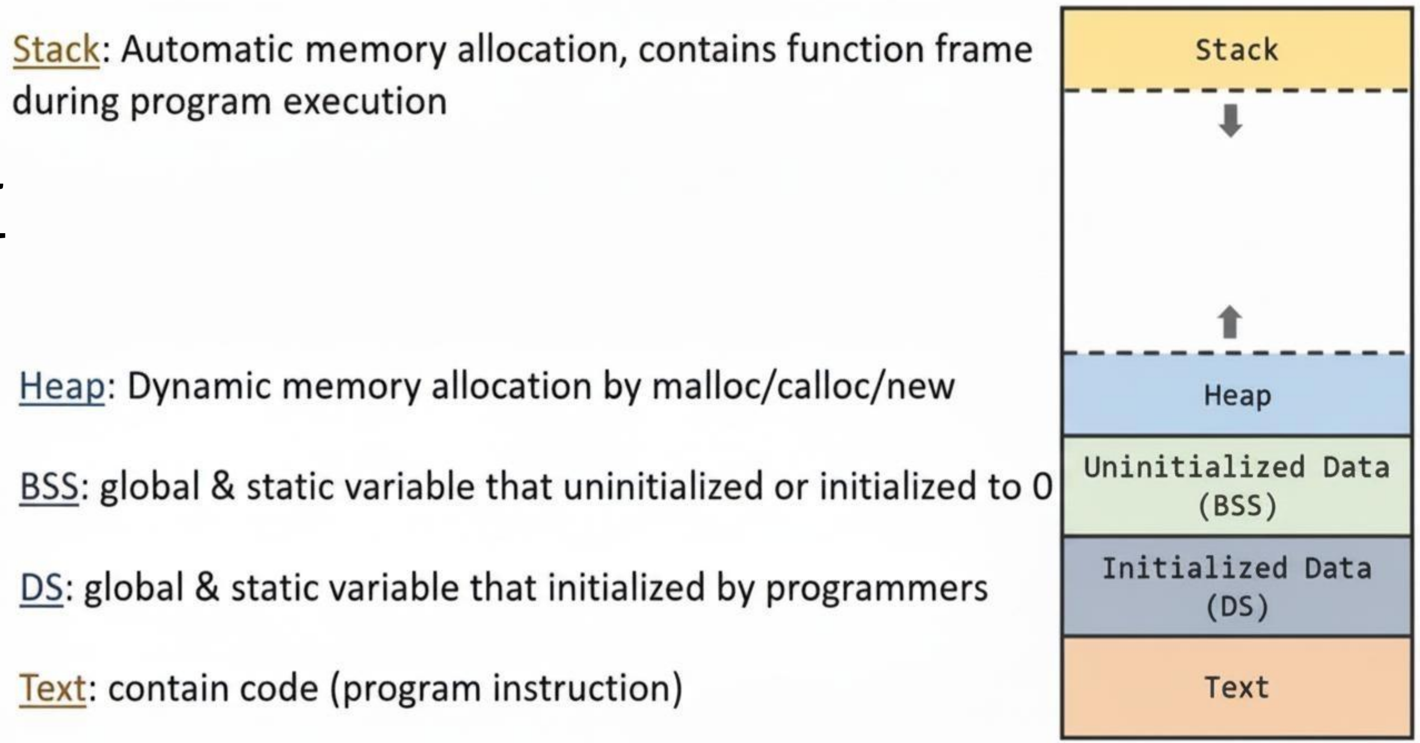
To modify a pointer passed to a function

When you pass a pointer to a function, you get a copy of the pointer.

If you want the function to change the pointer itself (not just the data it points to), you need to pass a pointer to that pointer.

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

void allocate(int** p) {
    *p = new int(42);  // allocate memory and
assign address to *p
}

int main() {
    int* ptr = nullptr;

    allocate(&ptr);   // pass address of ptr
(int**)

    cout << *ptr << endl;  // 42

    delete ptr;  // free memory
    return 0;
}
```

Pointers & Memory Management



Stack: Automatic memory allocation, contains function frame during program execution

Heap: Dynamic memory allocation by malloc/calloc/new

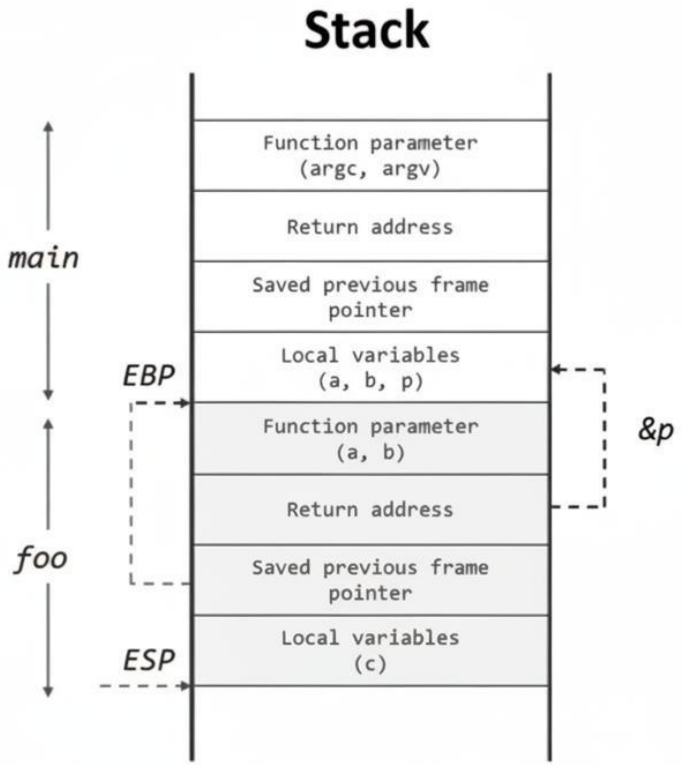
BSS: global & static variable that uninitialized or initialized to 0

DS: global & static variable that initialized by programmers

Text: contain code (program instruction)

Pointers & Memory Management

```
int foo(int a, int b) {
  int c = 10;
  return c + a * b;
}

int main(int argc, char *argv[]) {
  int a = 5, b = 6, p;
  p = foo(a, b );
  return 0;
}
```



Stack
Function parameter (argc, argv)
Return address
Saved previous frame pointer
Local variables (a, b, p)
Function parameter (a, b)
Return address
Saved previous frame pointer
Local variables (c)
main
foo
EBP
ESP
&p

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:

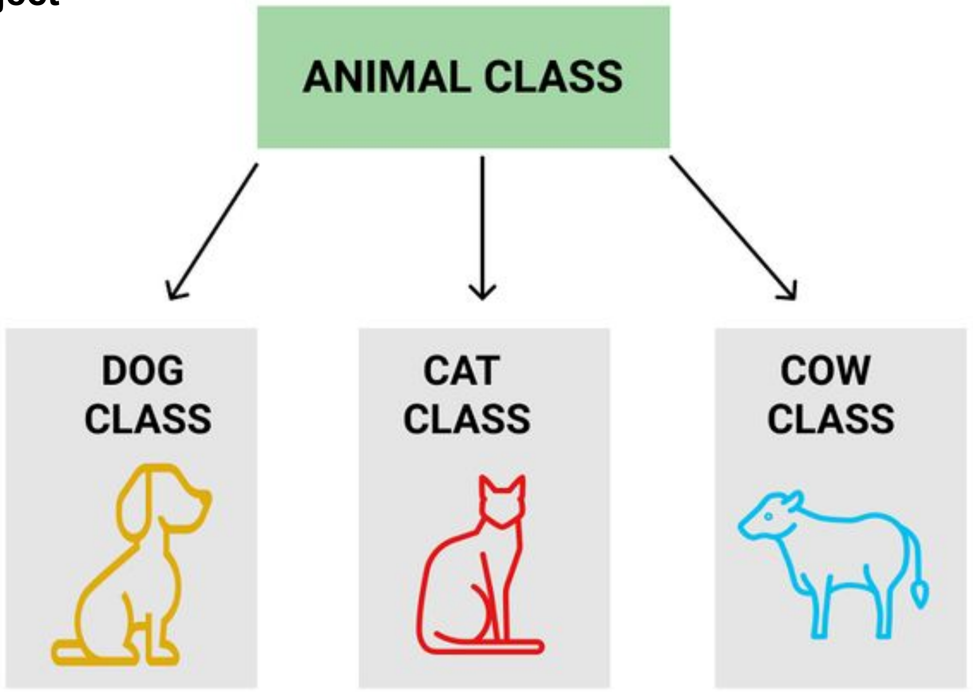
Table of Contents

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**.



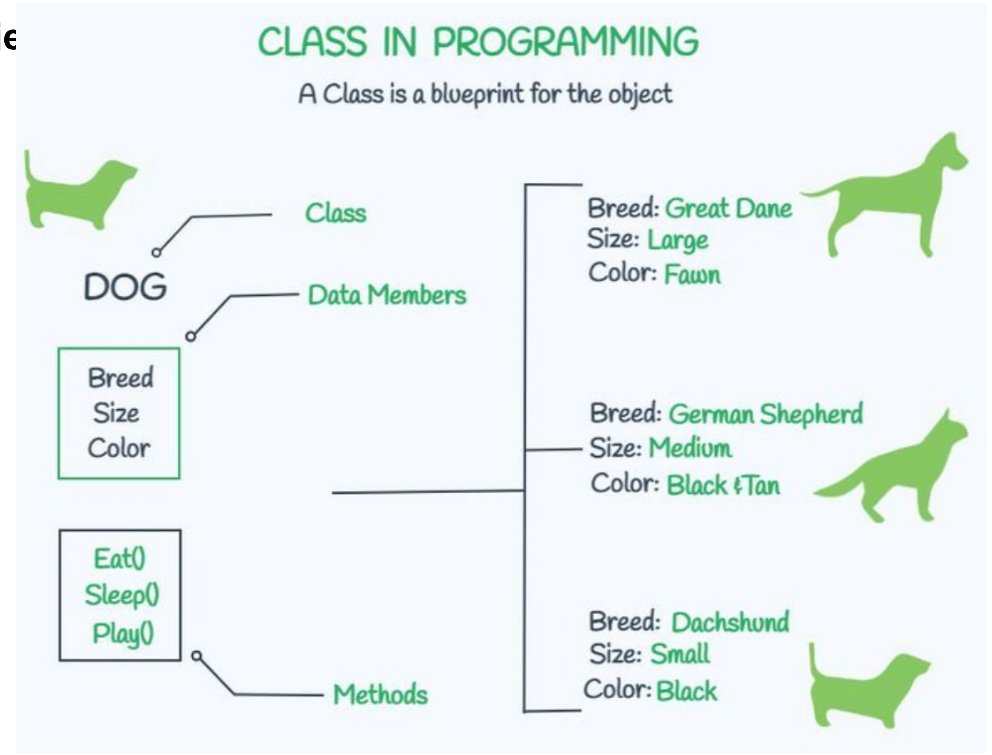
Classes and Objects

A **class** is a **blueprint** for creating objects. An **obje**
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

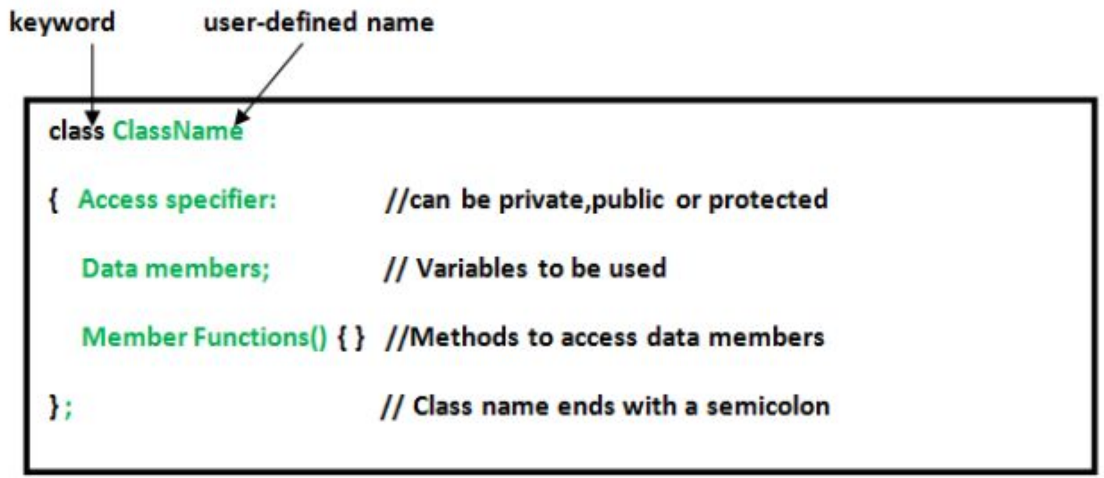
A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

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";
     }
};
```



```
class ClassName
{  Access specifier:            //can be private,public or protected
   Data members;                // Variables to be used
   Member Functions() { }       //Methods to access data members
};                              // Class name ends with a semicolon
```

Classes

A **class** is a user-defined data type.

Blueprint for creating objects.

Combines **data members (variables)** and **member functions (methods)**.

Supports OOP concepts like **Encapsulation,**

Inheritance, Polymorphism.

```
class Car {

    string brand;      // Data Member

    int speed;         // Data Member

public:

    void setData(string b, int s);  // Member
Function

    void display();

};
```

Objects

An **object** is an instance of a class.

Each object has its **own copy** of data members.

Member functions operate on that object's data.

```
Car c1, c2;  // Objects

c1.setData("BMW", 200);

c2.setData("Audi", 180);

c1.display();

c2.display();
```

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;

}
```

Data Members

Variables defined **inside a class**.

Hold data specific to each object.

Can have **different access specifiers**:

- `private` → accessible only within the class.
- `public` → accessible outside the class.
- `protected` → accessible in derived classes.

```
class Student {
    int rollNo;          // Private data member
public:
    string name;         // Public data member
};
```

Member Functions

Functions defined inside a class to operate on its data members.

Can be defined:

- Inside the class (inline by default).
- Outside the class using **scope resolution (::)**.

Control access to data (encapsulation).

```
class Student {
    int rollNo;
    string name;
public:
    void setData(int r, string n) {
        rollNo = r;
        name = n;
    }
    void display() {
        cout << "Roll: " << rollNo << " Name: " << 
name;
    }
};
```

Class & Object Example

```cpp
class Student {
    int rollNo;
    string name;
public:
    void setData(int r, string n) {
        rollNo = r; name = n;
    }
    void display() {
        cout << "Roll No: " << rollNo << ", Name: " << name << endl;
    }
};

int main() {
    Student s1, s2;
    s1.setData(1, "Ethan");
    s2.setData(2, "Rayan");

    s1.display();
    s2.display();
}
```

Note:

Class = Blueprint.

Object = Instance of a class.

Data Members = Variables inside a class.

Member Functions = Methods to operate on data members.

Together they implement **Encapsulation** in OOP

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;
}
```

What is output ?

```
#include <iostream>

using namespace std;

int main() {

    int arr[] = {10, 20, 30, 40};

    cout << *(arr + 2);

    return 0;

}
```

What is output ?

```
#include <iostream>

using namespace std;

int main() {

    int arr[5] = {1, 2, 3};

    cout << arr[3] << " " << arr[4];

}
```

What is output ?

```
#include <iostream>

#include <string>

using namespace std;

int main() {

    string s = "Hi";

    s += '!';

    cout << s;

}
```

What is output ?

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s = "abcd";
    cout << s[4];
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Demo {
public:
    int x;
    Demo() { x = 10; }
};
int main() {
    Demo d1, d2 = d1;
    d1.x = 20;
    cout << d2.x;
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Test {
public:
    int x = 5;
    void change(int val) { x = val; }
};
int main() {
    Test t1, t2;
    t1.change(10);
    cout << t1.x << " " << t2.x;
}
```

2. WAP in C++ to create a class Rectangle with data members length and width. Write member functions to calculate and display the area and perimeter.

Area = `length × width`

Perimeter = `2 × (length + width)`

Encapsulation in C++

Encapsulation = Wrapping data and functions into a single unit (class)

Protects data from direct access

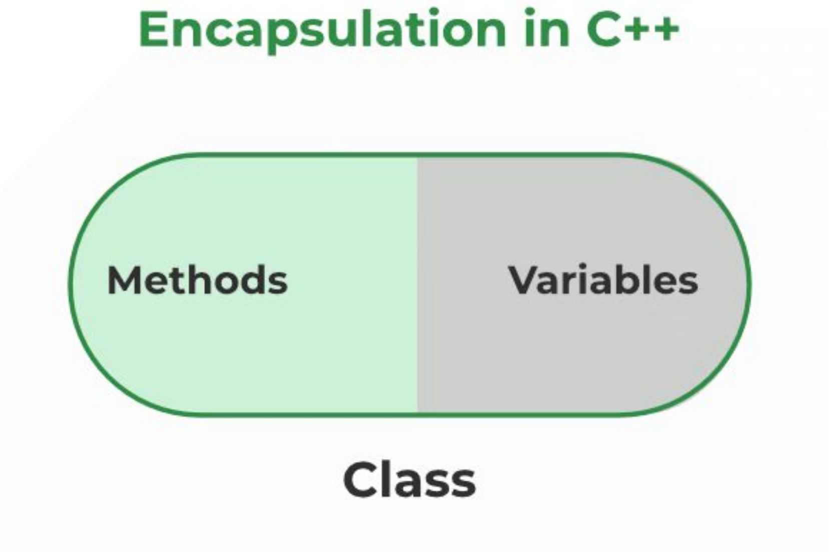
Achieved using access specifiers (private, public, protected)

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 **access specifiers** like `protected`, encapsulation helps restrict direct access to the class's data from outside, enhancing security and data integrity.



Key Features

Data hiding → only accessible via functions

Improved security → prevents unauthorized modification

Increases code maintainability

Helps in modular programming

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;

}
```

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
}
```

Base class member	Public inheritance	Protected inheritance	Private inheritance
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	Not inherited	Not inherited	Not inherited

Types of Inheritance

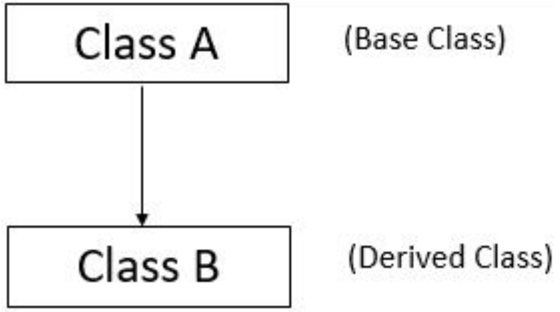


Fig: Single Inheritance

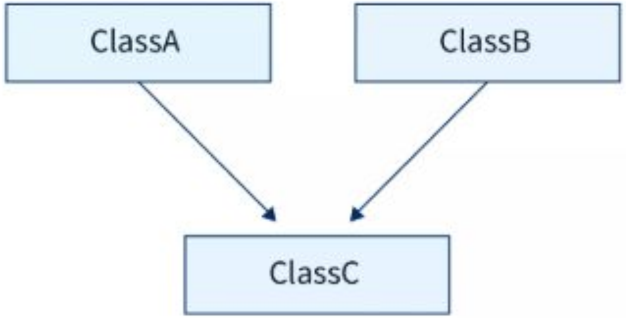


Fig: Multiple Inheritance

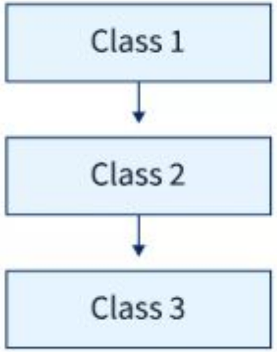


Fig: Multilevel Inheritance

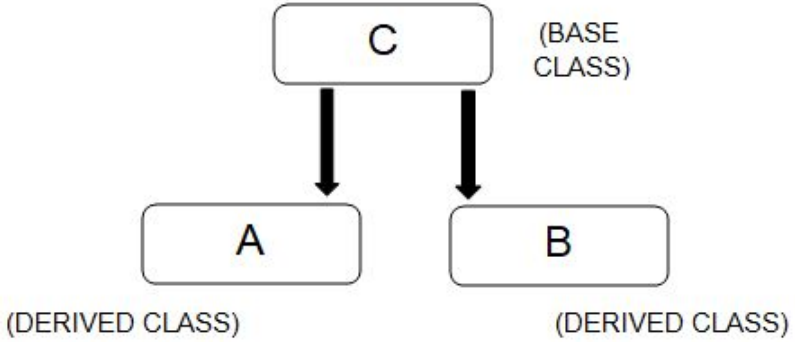


Fig: Hierarchical Inheritance

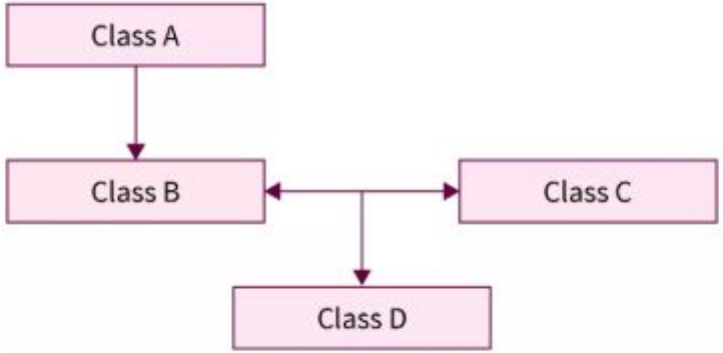


Fig: Hybrid Inheritance

Single Inheritance

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

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

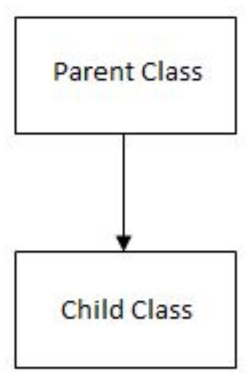


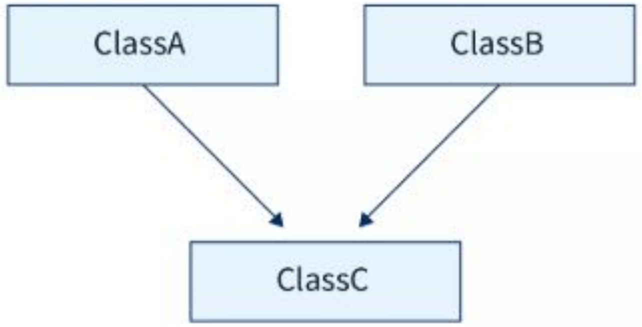
Fig: Single inheritance

```
#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**.

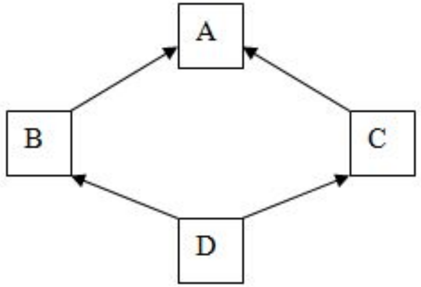
Allows a class to have features from multiple sources.



```
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**.

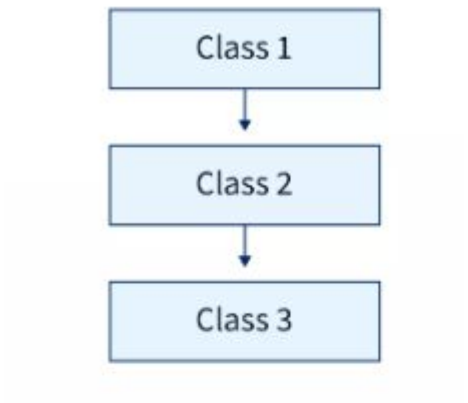


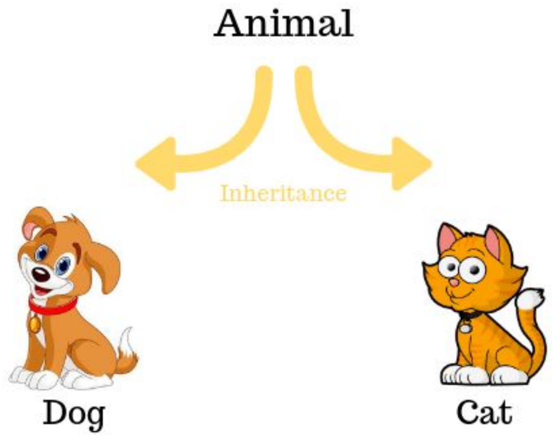
Fig: Multilevel Inheritance

```
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**.

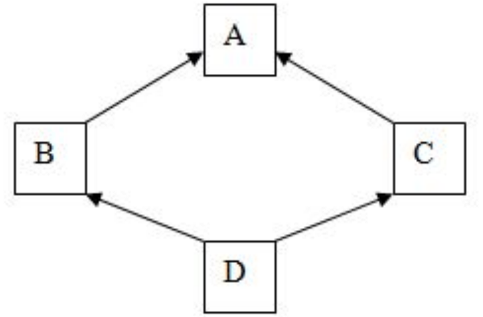
Useful when multiple classes share common functionality.



```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
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Bank {
private:
    int balance = 1000;
public:
    void setBalance(int b) { balance = b; }
    int getBalance() { return balance; }
};
int main() {
    Bank b;
    b.setBalance(5000);
    cout << b.getBalance();
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Data {
private:
    int x = 10;
public:
    void print() { cout << x; }
};
int main() {
    Data d;
    cout << d.x;
    d.print();
}
```

What is output ?

```
class A {
private:
    int a = 10;
protected:
    int b = 20;
public:
    int c = 30;
    void show() { cout << a << " " << b << " " << c; }
};
int main() {
    A obj;
    cout << obj.a;
    cout << obj.b;
    cout << obj.c;
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Base {
protected:
    int x = 5;
};
class Derived : public Base {
public:
    void show() { cout << x; }
};
int main() {
    Derived d;
    d.show();
}
```

What is output ?

```
class Base {
private: int a = 10;
protected: int b = 20;
public: int c = 30;
};
class Derived : private Base {
public:
    void print() {
        cout << a;
        cout << b << " " << c;
    }
};
int main() {
    Derived d;
    d.print();
}
```

What is output ?

```
#include <iostream>
using namespace std;
class Parent {
public:
    int x = 10;
};
class Child : public Parent {
public:
    int x = 20;
};
int main() {
    Child c;
    cout << c.x;
}
```

What is output ?

```
#include <iostream>
using namespace std;
class A {
public:
    void show() { cout << "A"; }
};
class B : public A {
public:
    void show() { cout << "B"; }
};
int main() {
    B obj;
    obj.show();
}
```

What is output ?

```
#include <iostream>
using namespace std;
class A {
public:
    int val = 1;
};
class B : public A {
public:
    int val = 2;
};
int main() {
    B obj;
    cout << obj.val << " " << obj.A::val;
}
```

What is output ?

```
class Data {
private:
    int x = 10;
};
int main() {
    Data d;
    cout << d.x;
}
```

3.WAP in C++ to implement a class BankAccount that has private members accountNumber and balance. Provide public functions to deposit, withdraw, and display balance.

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;
}
```

```cpp
class Number {
    int value;
public:
    // Parameterized constructor
    Number(int v) {
        value = v;
        cout << "Parameterized constructor called for " << value
<< endl;
    }

    // Copy constructor
    Number(const Number &n) {
        value = n.value;
        cout << "Copy constructor called for " << value << endl;
    }
    // Assignment operator (default one works, added for clarity)
    Number& operator=(const Number &n) {
        value = n.value;
        cout << "Assignment operator called for " << value <<
endl;
        return *this;
    }
    void show() {
        cout << "Value: " << value << endl;
    }
};
```

```cpp
int main() {
    Number n1(10);          //
Parameterized constructor called
    Number n2 = n1;         // Copy
constructor called
    Number n3(0);            //
Parameterized constructor called
    n3 = n1;                //
Assignment operator called

    cout << "\nValues:\n";
    n1.show();
    n2.show();
    n3.show();

    return 0;
}
```

`Number(const Number &n)`

- **`Number`** → The class name. This tells the compiler this is a **constructor**.
- **`const Number &n`** → The parameter:
  1. `Number &n` → n is a **reference** to another `Number` object.
     - Using reference prevents **copying the whole object** (which would call copy constructor again → infinite loop!).
  2. `const` → Ensures the object being copied is **not modified** inside the constructor.
     - Without `const`, you cannot pass **const objects** to copy constructor.

So this means: "I am creating a new `Number` object by referring to an **existing `Number` object**, without modifying it."

```
Number(const Number &n) {
    value = n.value;
    cout << "Copy constructor
called for " << value <<
endl;
}
```

the &n **does NOT give the address of n** in the usual sense like a pointer. It is a **reference**, which is slightly different from a pointer.

```
Number(const Number &n) {
    value = n.value;
    cout << "Copy constructor
called for " << value <<
endl;
}
```

What &n actually means

- n is a **reference to an existing object of type Number**.
- The & in const Number &n means: **"n is an alias for another Number object"**.

So when you write:

```
Number n2 = n1;  // copy constructor called
```

- n in the constructor **refers to n1**, the existing object.
- Any access to n.value **actually accesses n1.value**.

Shallow Copy

Explanation:

- obj2 points to the **same memory** as obj1.
- Changing obj1 affects obj2.
- Deleting obj1 leaves obj2 with a dangling pointer → crash.

```cpp
class Shallow {
    int* data;
public:
    Shallow(int d) {
        data = new int(d);
        cout << "Constructor called, data = " << *data << endl;
    }

    // Shallow copy constructor
    Shallow(const Shallow &s) {
        data = s.data;  // Copies pointer, NOT the actual value
        cout << "Shallow copy constructor called, data = " << *data << endl;
    }

    void setValue(int d) { *data = d; }
    void show() { cout << "Data = " << *data << endl; }

    ~Shallow() {
        delete data;
        cout << "Destructor called, data deleted" << endl;
    }
};

int main() {
    Shallow obj1(10);
    Shallow obj2 = obj1; // shallow copy

    obj1.setValue(20);   // change original
    cout << "After changing obj1:" << endl;
    obj1.show();
    obj2.show();         // obj2 also affected

    cout << "Deleting obj1..." << endl;
    // obj1 goes out of scope, destructor called → deletes data
    // obj2 still points to the same memory → dangling pointer

    return 0;  // obj2 destructor will cause crash (double delete)
}
```

Deep Copy

Explanation:

- obj2 has **its own memory**, separate from obj1.
- Changing obj1 does **not affect obj2**.
- Deleting obj1 does **not crash** because obj2 still owns its own memory.

```
class Deep {
    int* data;
public:
    Deep(int d) {
        data = new int(d);
        cout << "Constructor called, data = " << *data << endl;
    }

    // Deep copy constructor
    Deep(const Deep &s) {
        data = new int(*s.data);  // Allocate new memory and copy value
        cout << "Deep copy constructor called, data = " << *data << endl;
    }

    void setValue(int d) { *data = d; }
    void show() { cout << "Data = " << *data << endl; }

    ~Deep() {
        delete data;
        cout << "Destructor called, data deleted" << endl;
    }
};

int main() {
    Deep obj1(10);
    Deep obj2 = obj1; // deep copy

    obj1.setValue(20);   // change original
    cout << "After changing obj1:" << endl;
    obj1.show();
    obj2.show();         // obj2 unaffected

    cout << "Deleting obj1..." << endl;
    // obj1 destructor deletes its own memory
    // obj2 memory is independent → safe

    return 0;  // no crash
}
```

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;
}
```

Output?

```
class Base {
public:
    Base() {
        cout << "Base constructor called" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived constructor called" << endl;
    }
};

int main() {
    Derived d;
    return 0;
}
```

4.Copy Constructor in Inheritance

Write a program:

Base class Book with title and author.

Derived class TextBook with additional member subject.

Implement copy constructor in both base and derived classes.

Create a copy of a TextBook object and display all details.

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

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.

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

}
```

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.

Hierarchy of std::exception

std::exception is the **base class** for all standard exceptions in C++.

- It defines the virtual function what() that returns an error message.
- Derived classes provide specific error types.

```
std::exception
 ├── std::logic_error
 │    ├── std::invalid_argument
 │    ├── std::domain_error
 │    ├── std::length_error
 │    └── std::out_of_range
 │
 └── std::runtime_error
      ├── std::range_error
      ├── std::overflow_error
      ├── std::underflow_error
      └── std::system_error
```

const std::exception& e to catch standard exceptions.

`throw runtime_error("Something went wrong!")`

- This throws a `std::runtime_error`, which is derived from `std::exception`.

`catch (const exception& e)`

- This catches **all exceptions derived from `std::exception`**.
- **`e.what()`** returns the error message.

```
#include <iostream>

//#include <stdexcept>

int main() {

    try {

        throw runtime_error("Something
went wrong!");

// Throwing a standard exception

    }

    catch (const exception& e) {

// Catching standard exceptions

        cout << "Caught an exception: "
<< e.what() << endl; }

    return 0;

}
```

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++

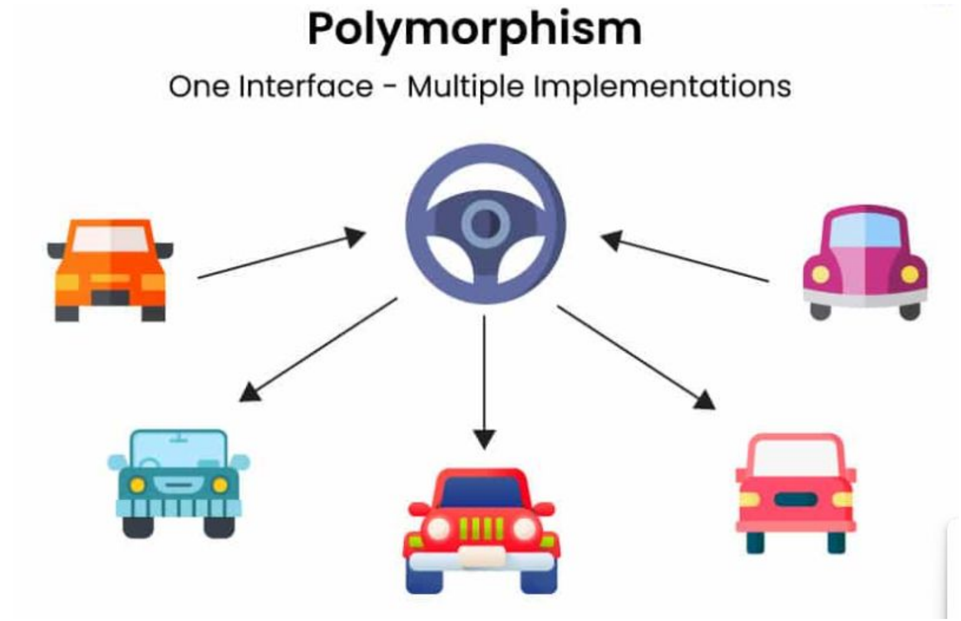
- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It’s hard to learn how to write Exception code that is safe.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

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)

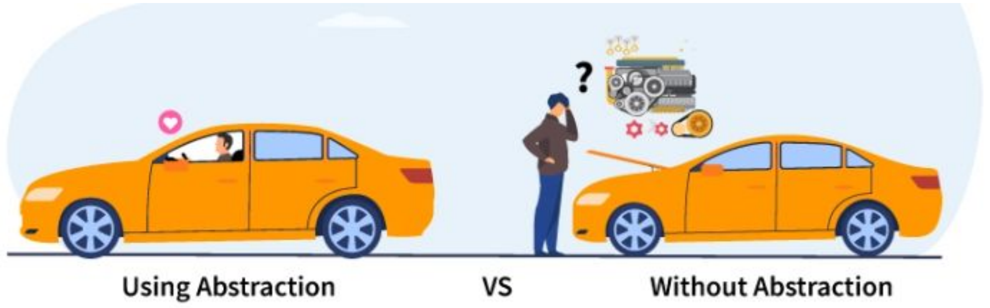
Method Overriding is a feature in object-oriented programming (OOP) where a derived class provides its own implementation of a method that is already defined in its base class. The key here is that the base class function is declared as **virtual**. This allows the derived class's version of the method 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.

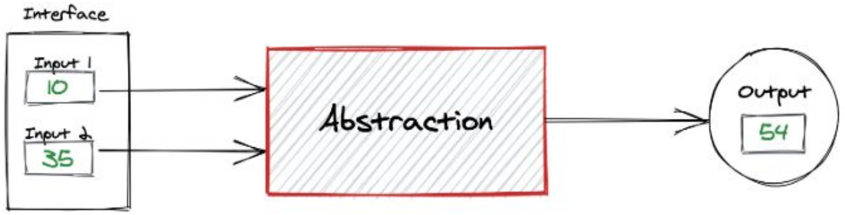
- **Why is Abstraction Important?**
  - Reduces complexity by exposing only essential details.
  - Prevents direct access to sensitive data.
  - Improves maintainability and security.
- How to Achieve Abstraction in C++?
  In C++, **abstraction** is achieved using:
  - **Abstract Classes (with Pure Virtual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