OOPS

Object-Oriented Programming (OOPs) is a way to **organize** and **design software** using **"objects"** which are instances of **"classes."** Classes define attributes and behaviors, while objects are specific instances of those classes. OOP helps in creating reusable, modular, and maintainable code.

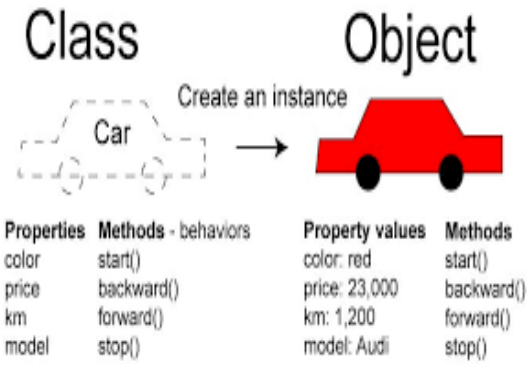
It offers several benefits like:

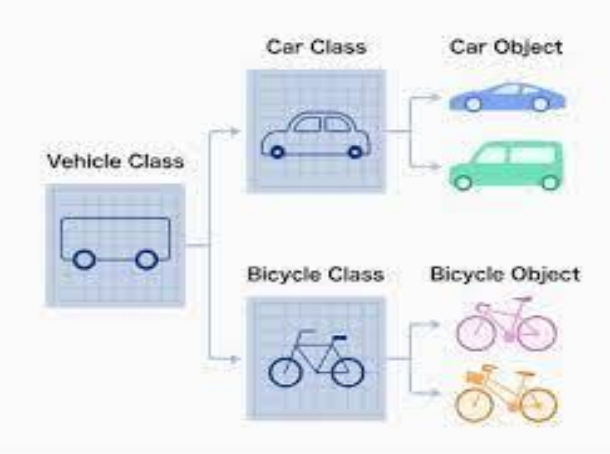
1. **Modularity:** Code is organized into classes, making it easier to manage and understand.
2. **Reusability:** Classes and objects can be reused across different programs.
3. **Scalability:** Easier to manage and scale large projects.
4. **Maintainability:** Code is easier to maintain and update.
5. **Encapsulation:** Data and methods are bundled together, enhancing security and data integrity.
6. **Inheritance:** New classes can inherit properties and methods from existing ones, reducing redundancy.
7. **Polymorphism:** Allows objects to be treated as instances of their parent class, enabling flexible and dynamic code.

CLASSES & OBJECTS

Objects: ***Entities*** in ***real world***. An object is an instance of a class, representing a specific entity that can use the class's attributes and methods.

Classes: ***Blueprint*** of these ***entities.*** It defines attributes (data) and methods (functions) that the objects will have





ACCESS MODIFIERS

Access modifiers are **keywords** used to **define** the **accessibility of members** (attributes and methods) within a **class**. They determine the scope and visibility of class members.

TYPES OF ACCESS MODIFIERS:

PUBLIC:

- Members declared as **public** can be **accessed from anywhere** in the **program**.

```
public:
    int age;
    void display() {
        cout << age;
    }
```

PRIVATE:

- Members declared as **private** can only be **accessed within the same class**.
- They are **not accessible from outside the class**. They are **set by default when we define a class.** But these **can be accessed** using **getter function**(method) & values can be **set using setter function provided** they are **methods of public access modifiers**

```
private:
    int salary;
    void setSalary(int s) {
        salary = s;
    }
```

PROTECTED:

- Members declared as **protected** can be **accessed within the same class** and **by derived classes**.
- They are **not accessible from outside** the **class unless through inheritance**.

```
protected:
    string name;
    void setName(string n) {
        name = n;
    }
```

DEFINING A CLASS:

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

class Teacher
{
public:
    // Properties / Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Methods / Memeber Functions
    void changeDept(string newDept)
    {
        dept = newDept;
    }
    void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"subject: "<<subject<<endl;
        cout<<"dept: "<<dept<<endl;
        cout<<"salary: Rs. "<<salary<<endl;
    }
};
```

CREATING AN OBJECT:

```
int main()
{
    Teacher t1; // object created
    t1.name = "Akash";
    t1.subject = "C++";
    t1.dept = "Computer Science";
    t1.salary = 500000;

    t1.getInfo();

    return 0;
}
```

CONSTRUCTOR

A constructor is a **special member function** of a **class** that **initializes objects of that class**.

- It is **automatically called** when an **object is created**.
- The **constructor** has the **same name** as the **class** and **does not have a return type**.
- Class doesn't have memory as such. Class is like a blueprint/template. The **object created through class takes place in memory**.
- Thus, **Memory allocation** only **happens** when **constructor is called/invoked**.

TYPES OF CONSTRUCTORS:

NON-PARAMETERIZED CONSTRUCTOR:

Takes no arguments / parameters.

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

class Teacher
{
public:
    // Properties / Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Non - Parameterized Constructor
    Teacher(){
        cout<<"Hi! I'm a Teacher"<<endl;
    }
    void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"subject: "<<subject<<endl;
        cout<<"dept: "<<dept<<endl;
        cout<<"salary: Rs. "<<salary<<endl;
    }
};
int main()
{
    Teacher t1;
    t1.name = "Akash";
    t1.subject = "C++";
    t1.dept = "Computer Science";
    t1.salary = 500000;
    t1.getInfo();
    return 0;
}
```

PARAMETERIZED CONSTRUCTOR:

Takes one or more arguments / parameters.

THIS POINTER

this is a special pointer in C++ that points to the current object.

this ->prop is same as ****(this.prop)***

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

class Teacher
{
public:
    // Properties / Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Parameterized Constructor using this pointer
    Teacher(string name, string subj,string dept,double sal){
       this->name = name;
       this->subject = subj;
       this->dept = dept;
       this->salary = sal;
    }

    void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"subject: "<<subject<<endl;
        cout<<"dept: "<<dept<<endl;
        cout<<"salary: Rs. "<<salary<<endl;
    }
};
int main()
{
    Teacher t1("Akash","C++","Computer Science",50000);
    t1.getInfo();

    return 0;
}
```

COPY CONSTRUCTOR:

It is a special Constructor (default) used to copy properties of one object into another.

```
class class_name
{
    int a;

public:
    //copy constructor
    class_name(class_name &obj)
    {
        a = obj.a;
    }
};
```

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

class Teacher
{
public:
    // Properties / Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Parameterized Constructor using this pointer
    Teacher(string name,string subj,string dept, double sal){
       this->name = name;
       this->subject = subj;
       this->dept = dept;
       this->salary = sal;
    }

    void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"subject: "<<subject<<endl;
        cout<<"dept: "<<dept<<endl;
        cout<<"salary: Rs. "<<salary<<endl;
    }
};
int main()
{
    Teacher t1("Akash","C++","Computer Science",50000);

    Teacher t2(t1); // default copy constructor invoked

    return 0;
}
```

DEEP COPY & SHALLOW COPY:

SHALLOW COPY

A ***shallow copy*** of an ***object copies all*** of the ***member values*** from ***one object to*** the ***another***.

- If members include pointers, only the pointer addresses are copied, leading to potential issues with shared resources and double deletion.

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

class Shallow {
public:
    int *data;

    Shallow(int d) {
        data = new int(d);
    }

    // Shallow copy constructor
    Shallow(const Shallow &source) : data(source.data) {
        cout<<"Shallow copy constructor -shallow copy"<<endl;
    }

    ~Shallow() {
        delete data;
        cout << "Destructor freeing data" << endl;
    }

    void display() {
        cout << "Data: " << *data << endl;
    }
};

int main() {
    Shallow obj1(100);
    Shallow obj2 = obj1; // Shallow copy
    obj2.display();

    return 0;
}
```

DEEP COPY

A ***deep copy***, on the other hand, not only ***copies the member values*** but also ***makes copies of any dynamically allocated memory*** that the ***member points to***.

- It ensures that each object has its own copy of the data, preventing shared resource problems.

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

class Student{
public:
string name;
double* cgpaPtr;

// Dynmaic Memory Allocation && Deep Copy
Student(string name, double cgpa){
    this->name = name;
    cgpaPtr = new double;
    *cgpaPtr = cgpa;
}
Student(Student &obj){
 this->name = obj.name;
 cgpaPtr = new double;
 *cgpaPtr = *obj.cgpaPtr;
}

void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"CGPA: "<<*cgpaPtr<<endl;

    }

};
int main()
{
   Student s1("Shruti",8.8);
   Student s2(s1);

   s1.getInfo();
   *(s2.cgpaPtr) = 9.2;
   s2.name = "Akash";
   s2.getInfo();

    return 0;
}
```

DESTRUCTOR

It is a special member function of a class that is ***executed when an object is destroyed***. Its **primary purpose** is to ***free resources acquired by the object***, such as ***deallocating memory***, ***closing file handles***, and ***releasing network resources***.

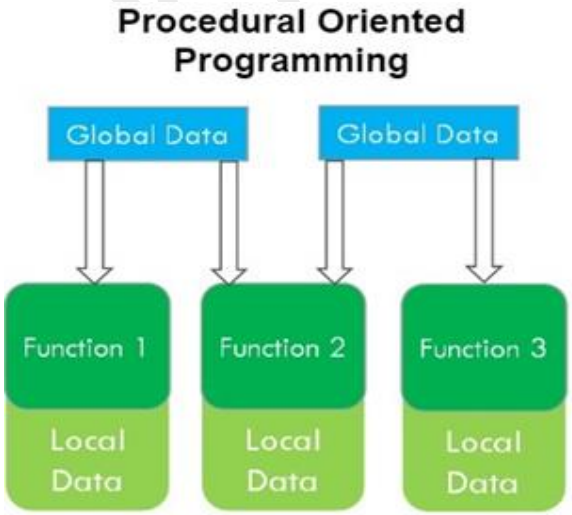
Key Points about Destructors

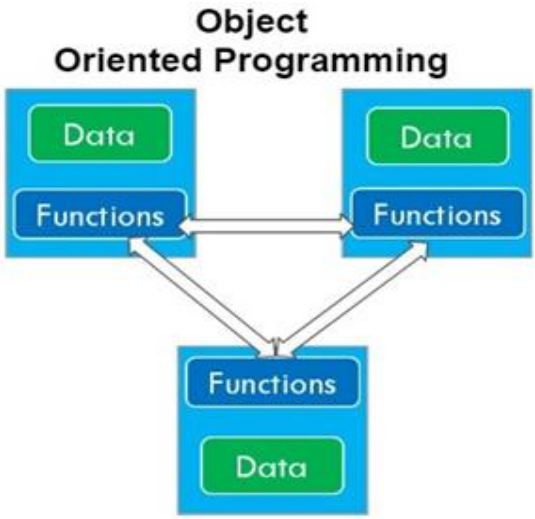
- **Name**: A destructor has the same name as the class, preceded by a tilde (~), e.g., ~MyClass().
- **No Parameters**: Destructors take no arguments and return no value.
- **Automatic Invocation**: Destructors are called automatically when an object goes out of scope or is explicitly deleted.
- **Single Destructor**: Each class can have only one destructor.
- **No Overloading**: Destructors cannot be overloaded.

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

class Student{
public:
string name;
double* cgpaPtr;

Student(string name, double cgpa){
    this->name = name;
    cgpaPtr = new double;
    *cgpaPtr = cgpa;
}

//Destructor
    ~Student(){
        cout<<"Hi! I'm destructor..I delete everything...\n";
        delete cgpaPtr;
    }
void getInfo(){
        cout<<"name: "<<name<<endl;
        cout<<"CGPA: "<<*cgpaPtr<<endl;

    }

};
int main()
{
   Student s1("Shruti",8.8);
  s1.getInfo();

    return 0;
}
```

DIFFERENCE BETWEEN POP & OOP

Feature	**Procedural Oriented Programming (POP)**	**Object Oriented Programming (OOP)**
Approach	Follows a top-down approach	Follows a bottom-up approach
Structure	Program is divided into functions	Program is divided into objects
Data Access	Data is global and shared by functions	Data is encapsulated in objects
Data Security	Less secure as data is exposed	More secure due to encapsulation
Focus	Focuses on functions	Focuses on objects
Code Reusability	Less reusability	High reusability through inheritance
Inheritance	Does not support inheritance	Supports inheritance
Polymorphism	Does not support polymorphism	Supports polymorphism
Examples	C, Pascal	C++, Java, Python





PILLARS OF OOPS

Object-Oriented Programming (OOP) is based on **four** **fundamental principles**: ***Encapsulation***, ***Inheritance***, ***Polymorphism*** and ***Abstraction***.

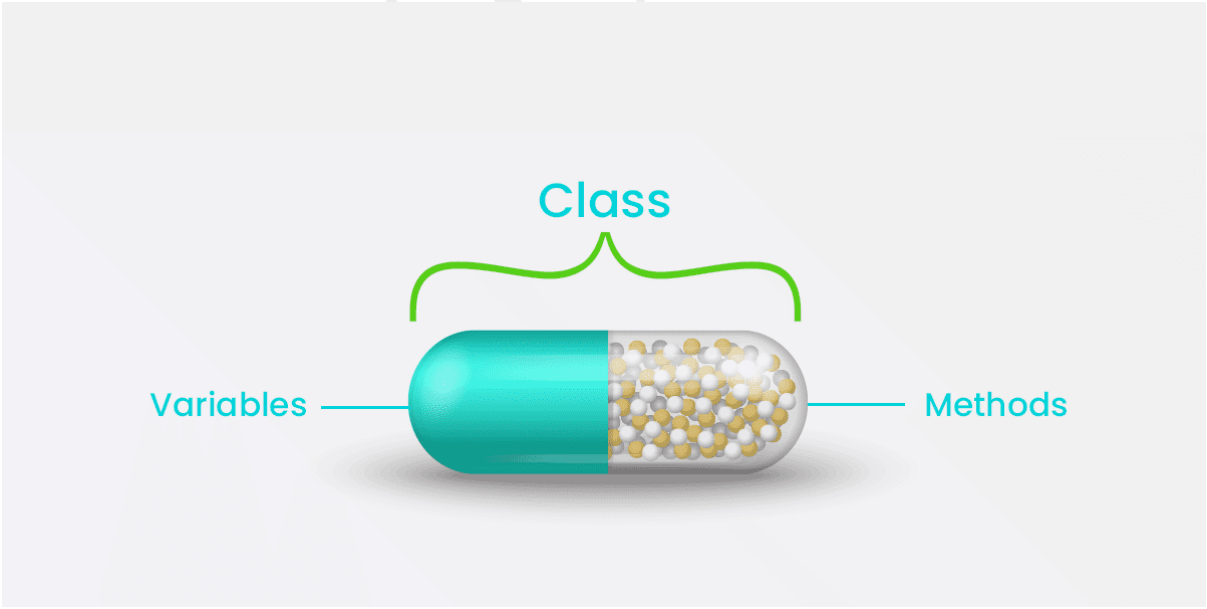
ENCAPSULATION

Definition: Encapsulation is the ***wrapping up*** ***data & member functions*** into a ***single unit*** called ***Class***.

- It helps in ***data-hiding***. It ***helps hiding sensitive information*** like ***bank balance*** of user and ***passwords*** *by* ***using access modifiers***.

- It restricts direct access to some of the object's components and can prevent the accidental modification of data.

Analogy: Think of a capsule or a pill. It encapsulates the medicine inside it. You cannot access the medicine directly; you have to take the pill to get the benefits.

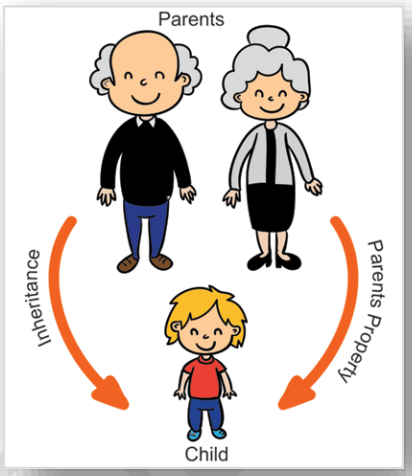


CODE EXAMPLE:

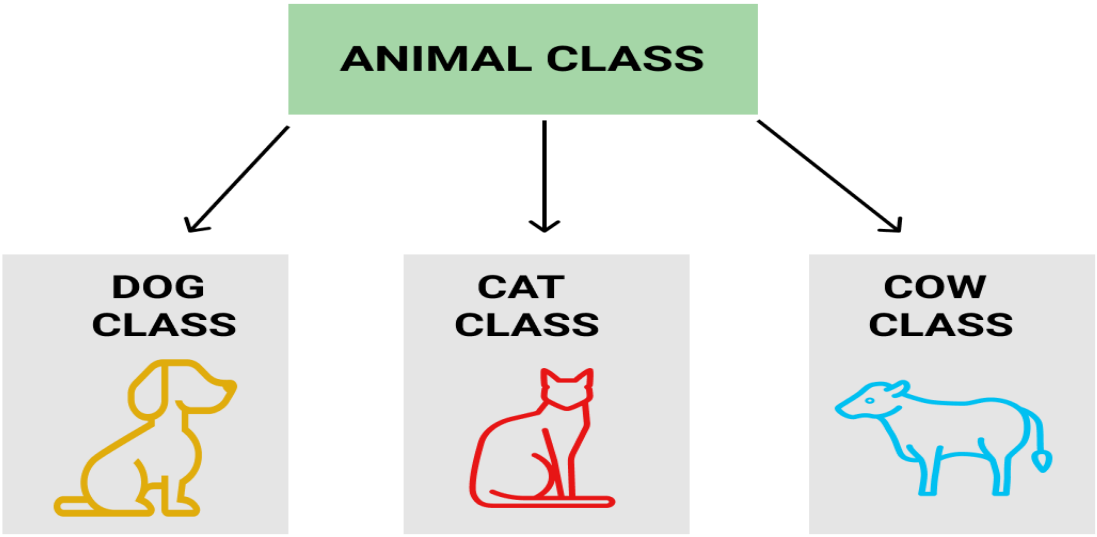
```cpp
#include <iostream>
#include <string>
using namespace std;

class Accounts {
private:
    double balance;
    string password; // data-hiding

public:
    string accountId;
    string username;

    // Constructor
    Accounts(string accId, string user, double bal, string pwd) {
        accountId = accId;
        username = user;
        balance = bal;
        password = pwd;
    }

    // Getter for balance
    double getBalance() {
        return balance;
    }

    // Setter for balance
    void setBalance(double newBalance) {
        if (newBalance >= 0) {
            balance = newBalance;
        } else {
            cout << "Invalid balance amount!" << endl;
        }
    }

    // Function to display account details (excluding password)
    void displayAccountDetails() {
        cout << "Account ID: " << accountId << endl;
        cout << "Username: " << username << endl;
        cout << "Balance: $" << balance << endl;
    }
};

int main() {
    // Creating an object of the Accounts class
    Accounts account1("123456789", "JohnDoe", 1000.0, "securepassword");

    // Displaying account details
    account1.displayAccountDetails();

    // Modifying the balance
    account1.setBalance(1500.0);
    cout << "Updated Balance: $" << account1.getBalance() << endl;
    return 0;
}
```

INHERITANCE

Definition: It is the process by which **properties** & **member function** of ***base class*** are ***passed on*** to the ***derived class***.



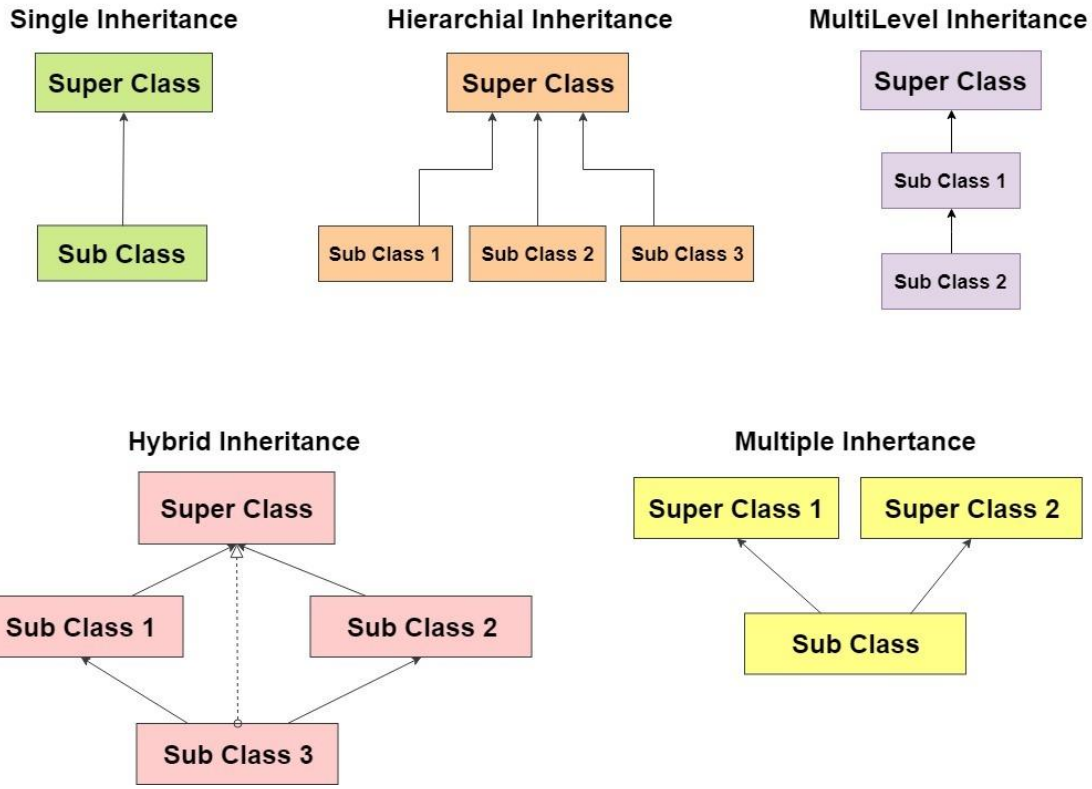
- The derived class inherits attributes and behaviours (methods) from the base class, allowing code reuse and extending functionality.
- **Analogy:** A child inherits characteristics and behaviours from their parents. For example, a child might inherit eye colour or height from their parents.



MODES OF INHERITANCE:

	Derived Class	Derived Class	Derived Class
Base Class	**Private Mode**	**Protected Mode**	**Public Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

TYPES OF INHERITANCE:



CODE EXAMPLE:

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

class Person{
public:
    string name;
    int age;
    Person()
    {
        cout << "I'm parent\n";
    }
};

class Student : public Person{
public:
    int roll;

    Student(string name,int age,int roll){
        this->name = name;
        this->age = age;
        this->roll = roll;
        cout << "I'm child\n";
    }
    void getInfo(){
        cout<< "name: "<< name<<endl;
        cout<< "age: "<< age<<endl;
        cout<< "rollNo: "<< roll<<endl;
    }
};

int main(){
    Student s1("Shruti",19,418);
    s1.getInfo();

    return 0;
}
```

POLYMORPHISM

Polymorphism is the **ability of objects** to take on ***different/multiple forms*** or behave in different ways ***depending on the context*** in which they are used.

COMPILE-TIME POLYMORPHISM:

Compile-time polymorphism is achieved using method overloading and operator overloading. It is resolved during the compilation of the program.

METHOD OVERLOADING

Method overloading allows multiple functions with the same name but different parameters to be defined.

```
#include <iostream>
using namespace std;
// Method Overloading
class Print {
public:
    void display(int i) {
        cout << "Displaying int: " << i << endl;
    }

    void display(double f) {
        cout << "Displaying float: " << f << endl;
    }

    void display(string s) {
        cout << "Displaying string: " << s << endl;
    }
};

int main() {
    Print obj;
    obj.display(5);
    obj.display(5.5);
    obj.display("Hello");

    return 0;
}
```

OPERATOR OVERLOADING

Operator overloading allows the user to redefine the way operators work for user-defined types.

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

class Complex {
private:
    float real;
    float imag;
public:
    Complex(): real(0), imag(0) {}

    Complex operator + (const Complex &obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }

    void input() {
        cout<<"Enter real and imaginary parts respectively: ";
        cin >> real >> imag;
    }

    void output() {
        if(imag < 0)
          cout<<"Output Complex number:"<<real<<imag<< "i";
        else
         cout<<"Output Complex number:"<<real<<"+"<<imag<<"i";
    }
};

int main() {
    Complex complex1, complex2, result;

    cout << "Enter first complex number:\n";
    complex1.input();

    cout << "Enter second complex number:\n";
    complex2.input();

    result = complex1 + complex2;
    result.output();

    return 0;
}
```

RUN-TIME POLYMORPHISM:

Run-time polymorphism is achieved using inheritance and virtual functions. It is resolved during the runtime of the program.

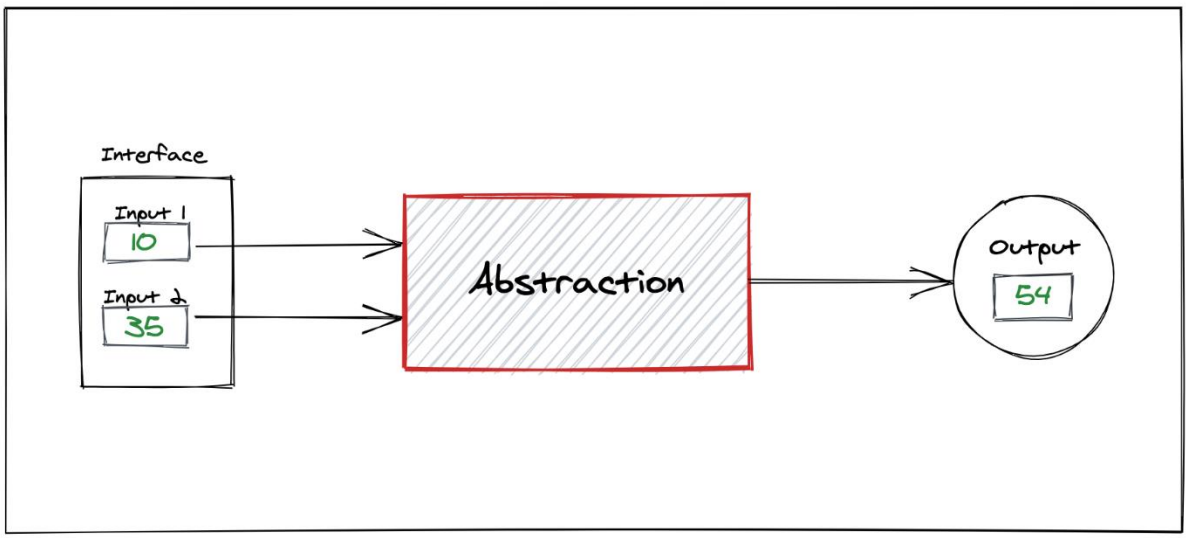
VIRTUAL FUNCTIONS

Virtual functions allow derived classes to override(inheriting) the behaviour of base class methods.

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

class Base {
public:
    virtual void show() {
        cout<<"Base class show function called"<< endl;
    }

    void display() {
        cout<<"Base class display function called"<<endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout<<"Derived class show function called"<<endl;
    }

    void display() {
      cout<<"Derived class display function called"<<endl;
    }
};

int main() {
    Base* b;
    Derived d;
    b = &d;

    // Virtual function, binded at runtime
    b->show();

    // Non-virtual function, binded at compile time
    b->display();

    return 0;
}
```

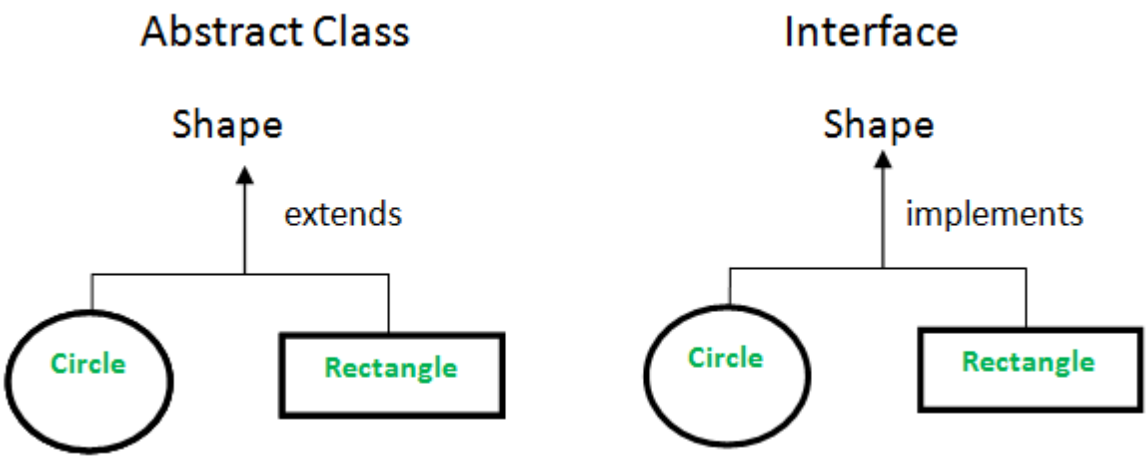
ABSTRACTION

Hiding all ***unnecessary details*** & ***showing*** only the ***important parts*.** It allows the programmer to reduce and factor out details so that one can focus on a few concepts at a time.



ABSTRACT CLASSES:

- Abstract Classes are ***base(parent) class*** from ***which other Classes*** are ***derived***. It is basically a template for creating other classes.
- They ***cannot be instantiated*** and are meant to be inherited.
- Abstract classes are typically used to ***define an interface for derived(child) classes***.



CODE EXAMPLE:

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

// Abstract class
class Shape {
public:
    virtual double getArea() const = 0; // Pure virtual function
    virtual ~Shape() {} // Virtual destructor
};

// Derived class for Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const override {return 3.14159 *radius *radius;}
};

// Derived class for Rectangle
class Rectangle : public Shape {
private:
    double length, width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double getArea() const override { return length * width; }
};

int main() {
    Shape* shapes[] = { new Circle(5.0), new Rectangle(4.0, 6.0) };
    for (Shape* shape : shapes) {
        cout << "Area: " << shape->getArea() << endl;
        delete shape;
    }
    return 0;
}
```