C++ Classes

Panchatcharam Mariappan

Associate Professor

Department of Mathematics and Statistics,
IIT Tirupati

OOPS CONCEPTS

- ✓ **Programming Paradigms**
  - Procedural
    - Modules, Data Structures, Procedures that Operate upon them
  - Objectural:
    - Objects which encapsulate data and behavior
    - Messages passed between objects
  - Functional
    - Functions, Closures, Recursion, lists, …

- ✓ Object Oriented Programming
- ✓ Programming paradigm or programming language mode
- ✓ Programs are organized around data or objects rather than functions and logic
- ✓ It compartmentalizes data into objects or data fields
- ✓ It describes the object contents and its behaviour through the declaration of classes or methods
- ✓ Encapsulation: Easier to manage, variables and states are hidden behind well-defined boundaries

- ✓ **Easy to design software**
- ✓ **Easy to maintain the software**
- ✓ **Reusable software**

- ✓ **A data type**
  - ☞ **Stores Data**
  - ☞ **+ Operations defined to act on the data**
- ✓ **Tangible Entities (Physically exists in real world)**
  - **Person, Student, Locker, Air Ticket, etc**
- ✓ **Intangible Entities (Exists logically in real world)**
  - **Bank Account, Email, Reservation**
- ✓ **Interactions between objects define the system operations**

- Take a Bank Details or Your Mobile Phone or PC
  - ✓ It is not necessary that everyone should know everything about your account
  - ✓ Manager/Administrator has a role
  - ✓ Cashier/User has a role
- ✓ Think: A piece of code as black box
  - ✓ Cannot See
  - ✓ Do not need to see
  - ✓ Do not want to see
  - ✓ High Coding details

- ✓ **Attributes or Data Attributes**
  - ✓ Characteristics or properties of an entity in a database table
  - ✓ A named piece of data or variable
  - ✓ Data members (class variables and instance variables)

- ✓ Example 1: Student has
  - ☞ Name
  - ☞ Roll Number
  - ☞ Marks
  - ☞ Branch/discipline

- ❖ Example 2: Circle has
  - ☞ Radius
  - ☞ Center
- ❖ Example 3: Rectangle has
  - ☞ Sides/Edges
  - ☞ Vertices

- ✓ **Methods or Procedural Attributes**
  - ✓ **Attributes bound to functions/behavior/operators**

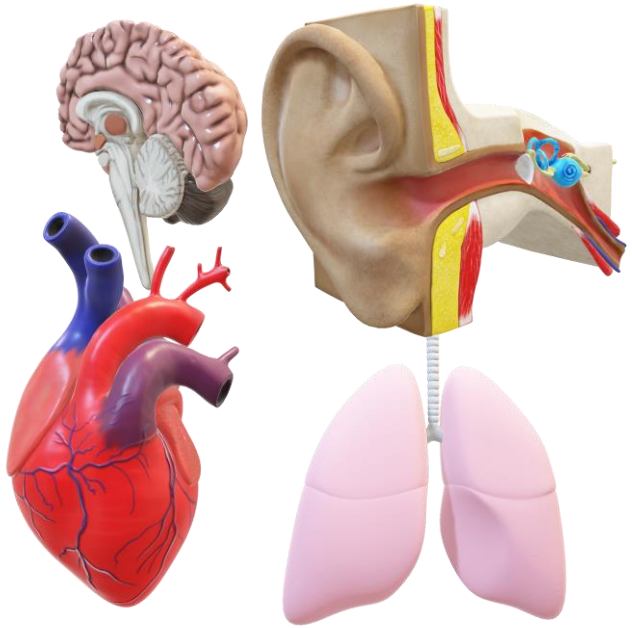
- ✓ **Example 1: Student has**
  - ☞ **Average Marks Calculation**
  - ☞ **Decide Grades**

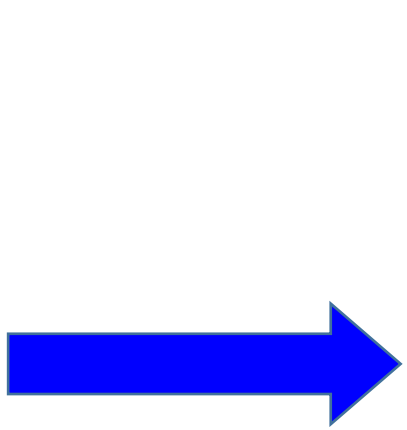
- ❖ **Example 2: Circle has**
  - ☞ **Area**
  - ☞ **Circumference**

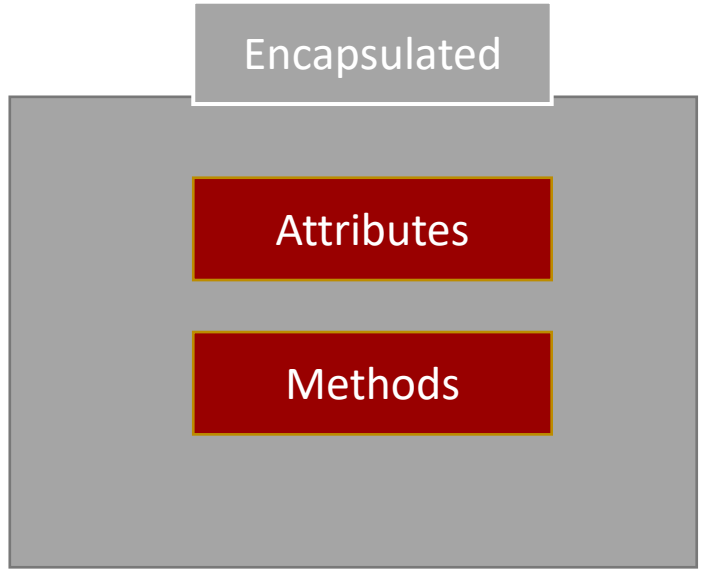
- ❖ **Example 3: Rectangle has**
  - ☞ **Area**
  - ☞ **Circumference**

- ✓ A process by which class components interact
  - Send data to another object
  - Request data from another object
  - Request object to perform some behaviour
- ✓ Implemented as methods (not called functions)
  - Functions are process that are object independent
  - Methods are dependent on the state of the object

- ✓ **Encapsulation implements the concept of abstraction**
  - ✵ **Details associated with object**
  - ✵ **End user could see the public interface, but implementation are hidden**







- ✓ **Attributes and methods are encapsulated within the logical boundary of the object entity**
  - ✵ **In procedural paradigms, data and functions are typically maintained as separate entities**
  - ✵ **In Objectural paradigms, each object has attributes (data) and methods (functions) that operates upon those attributes**

Glance at a Class

- ✓ Classes
  - A definition of objects of the same kind
  - Basic unit of OOP
  - A blueprint, template or prototype that encapsulates both static attributes and dynamic behaviours within a box
  - Defines a public interface for using these boxes
  - easily reusable
  - Combines data structures and algorithms in the same box

- ✓ Classes
  - Bundling Data
  - + Functionality

- ✓ Classes
  - Bundling Data
  - + Functionality

- ✓ Classes
  - A collection of functions and attributes
  - Attached to a specific name to represent an abstract concept

- ✓ Classes
  - User-defined prototype for an object with attributes and methods

- ✓ Classes
  - Bundling Data
  - + Functionality

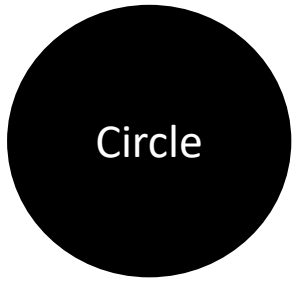
- ✓ Classes
  - A collection of functions and attributes
  - Attached to a specific name to represent an abstract concept

- ✓ Classes
  - User-defined prototype for an object with attributes and methods

- ✓ **A software item that contains variables and methods**
- ✓ **Object Oriented Design focuses on**
  - Encapsulation
    - dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - the ability to create subclasses that contain specializations of their parents

- ✓ **View class/object as factories or templates**
- ✓ **An Individual Object of a certain class**
- ✓ **Each object instance takes all the properties of the class from which it was created**

Abstract Concept

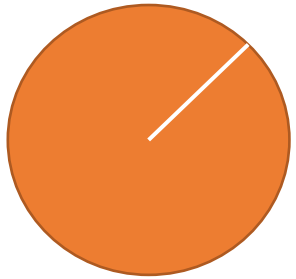


Attributes

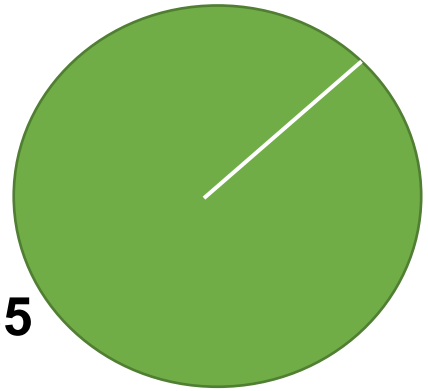
- ☞ **Radius**

Methods/Functions

- ☞ **Area()**
- ☞ **Circumference()**



Radius=3
Area()=28.27
Circumference()=18.85



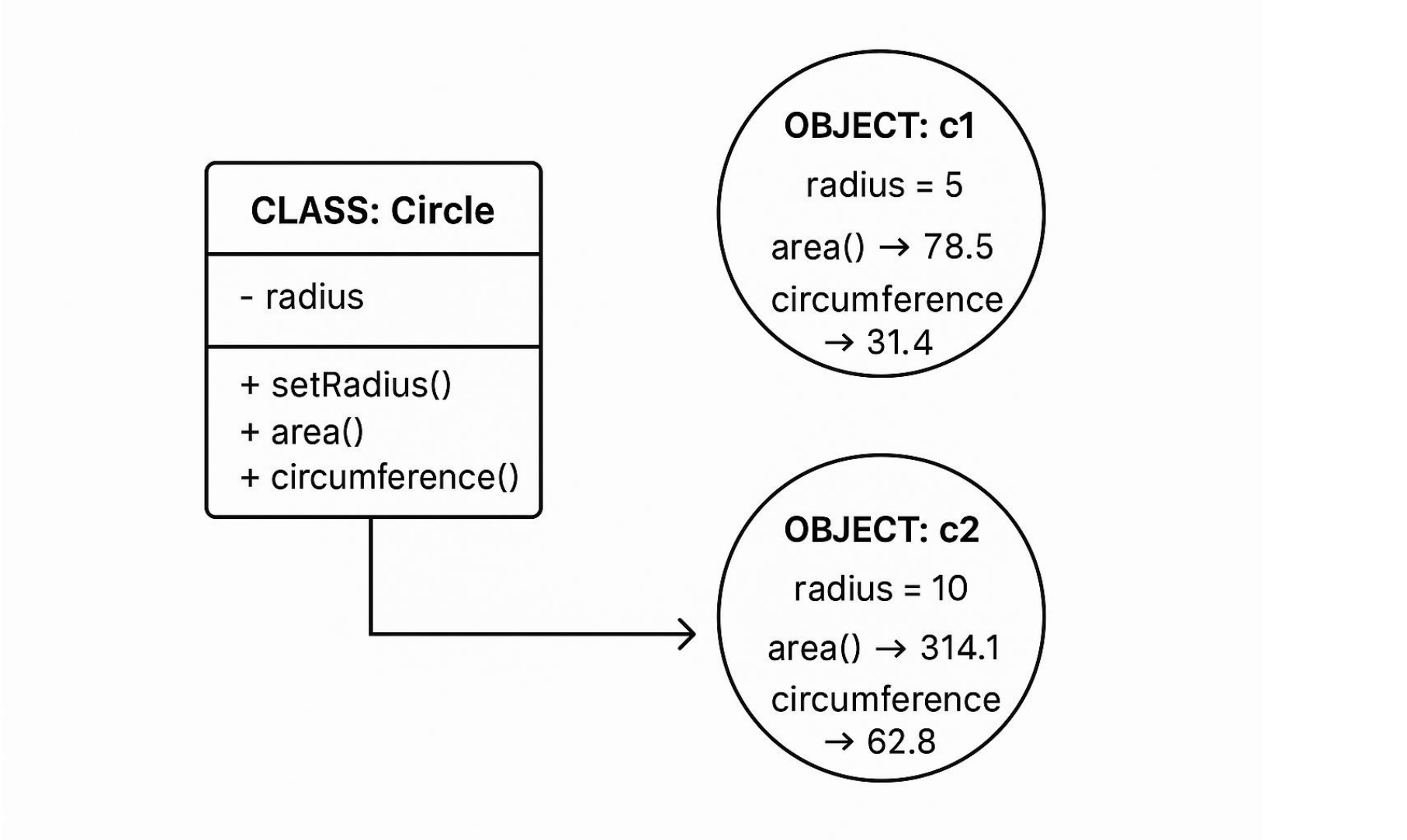
Radius=6
Area()=113.1
Circumference()=37.7

Class Vs Instance of a Class

- ✓ class name is the **type**
  - ➢ `class circle(object)`
- ✓ Defined generically
  - ➢ `area=pi*r*r`
- ✓ Defines data and methods **common across all instances**

- ✓ Instance is **one specific** object
  - ➢ `Circle C1;`
  - ➢ `Circle C2;`
- ✓ Data varies between instances
  - ➢ `C1.setRadius(5.0)`
  - ➢ `C2.setRadius(10.0)`
  - ➢ `C1.r` and `C2.r` are different
- ✓ Instance has the **structure of the class**
  - ➢ `C1.area()` → `78.5398`
  - ➢ `C2.area()` → `314.159`

Class Vs Instance of a Class



Class Vs Instance of a Class

Concept	Description	Example
Class	Blueprint describing what all circles have/do	class Circle { ... };
Instance/Object	Actual circle with its own radius	Circle c1; Circle c2;
Data	Each object stores its own radius	c1.radius = 5, c2.radius = 10
Function	All objects can call the same methods	c1.area(), c2.area()

- **Object**
  - A unique instance of data structured defined by its class
  - Contains Data Members
    - Class Variables
    - Instance Variables
    - Methods

- **Class**
  - User-defined prototype for an object
  - Set of attributes to characterize any object of the class
  - Attributes
    - Data Members(Class Variables, Instance Variables)
    - Methods
    - Accessed via dot notation
- **Instance**
  - An individual object of a certain class
- **Instantiation**
  - Creation of an instance of a class

- **Class Variable**
  - Shared by all instance of a class
  - Defined within a class
  - Outside any of the class' methods
  - Not use as frequently as instance variable

- **Instance Variable**
  - Defined inside a method
  - Belongs to only to the current instance of the class

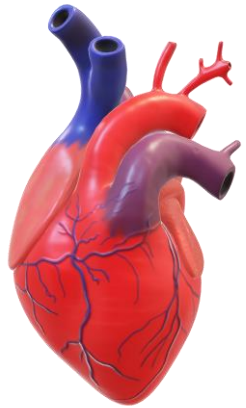
- **Data Member**
  - A class variable
  - Instance Variable
  - Holds data associated with a class and its objects

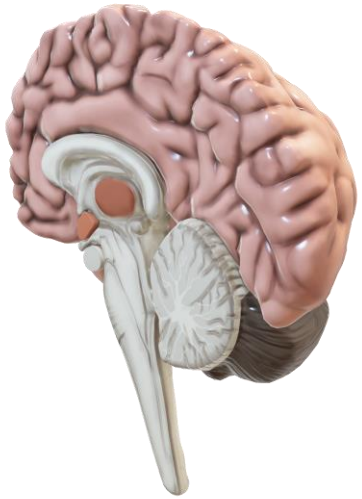
- **Method**
  - A special kind of function that is defined in a class definition

- **Function Overloading**
  - ☞ Assignment of more than one behaviour to a particular function
- **Operator Overloading**
  - ☞ Assignment of more than one function to a particular operator
- **Inheritance**
  - ☞ Transfer of the characteristic of a class to other classes that are derived from it

- **Overriding**
  - When inheriting from a class, we can alter behaviour of the parent class by overriding function
  - Declaring functions in the subclass with the same name
  - More precedence over parent class
- **Polymorphism**
  - Two objects of different classes
  - Supports same set of functions
  - Attributes can be treated identically
  - Implementation are different, but appears to be same

- Do you know or care how a smartphone or TV or washing machine or any electrical appliances or your own body??
  - ✕ No. As long as you are the user of the appliances and the appliance functions well




Why Use OOP and Classes of Objects

- Group different object of the same type
  - Classes and objects are more like the real world
  - Mimic the real world
  - Minimize the semantic gap by modelling the real world
- Semantic Gap:
  - Difference Between the real world and the representation in a computer
- Allow you to define an interface to some object and its operations
  - Use it without knowing the internals
- Modularize the program into multiple objects that work together, each has its own purpose

C++ CLASSES

Syntax for Classes

```
class ClassName
{
        //Data members
        private:
        varType1 var1;
        varType2 var2;
        varType3 *var3;
        varType4 var4[100];
        //Member Functions
        public:
        ClassName(varType1 var1, varType2 var2, varType3 *var3, varType4 var4[100]); //Constructor
        Print();
        void Function1();
        varType1 Function2(varType2 var2);
        varType2 Function3(varType3 *var);
}
```

```
class Course
{
        private:
        string code;
        string name;
        float Marks;
        char Grade;
        public:
        Course(string c="MA0001",string n="AVBACA",float m=0.0,char G='S')
        {
                code=c;
                name=n;
                Marks=m;
                Grade=G;
        }
        void CalculateGrade();
        void Print();

};
```

Class definition

name

```
class Course
{
        private:
        string code;
        string name;
        float Marks;
        char Grade;
        public:
        Course(string c="MA0001",string n="AVBACA",float m=0.0,char G='S')
        {
                code=c;
                name=n;
                Marks=m;
                Grade=G;
        }
        void CalculateGrade();
        void Print();

};
```

Members of the Class

Constructors of the Class

Functions of the Class

Class definition

name

```
class Course
{
        private:
        string code;
        string name;
        float Marks;
        char Grade;
        public:
        Course(string c="MA0001",string
n="AVBACA",float m=0.0,char G='S')
        {
                code=c;
                name=n;
                Marks=m;
                Grade=G;
        }
        void CalculateGrade();
        void Print();

};
```

Members of the Class

Constructors of the Class

Functions of the Class

```
int main()
{
        Course Course1("MA5191",
"Programming Laboratory",95.0,'A');
        Course1.CalculateGrade();
        Course1.Print();
        return 0;

}
```

Instance of the Class

Mapped to c, n, m and G

c->"MA5191"

n->"Programming Laboratory"

m->95.0

G->'A'

Download the CPP File

- A constructor in C++ is a special member function of a class that is automatically called when an object is created.
- Its main purpose is to initialize the object's data members.

- Has the same as the class
- Has no return type, not even void.
- Is automatically invoked when you create an object.

- A constructor in C++ is a special member function of a class that is automatically called when an object is created.
- Its main purpose is to initialize the object's data members.

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

class Student {
public:
    string name;
    int age;
    // Constructor
    Student(string n, int a)
    {
        name = n;
        age = a;
    }

    void display() {
        cout << "Name: " << name << ", Age: "
<< age << endl;
    }
};
```

```
int main() {
    Student s1("Raja", 20); // constructor
called automatically
    s1.display();
}
```

Implicit vs Explicit Constructor Calls

- A constructor in C++ is a special member function of a class that is automatically called when an object is created.
- Its main purpose is to initialize the object's data members.

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

class Student {
public:
    string name;
    int age;
    // Constructor
    Student(string n, int a)
    {
        name = n;
        age = a;
    }

    void display() {
        cout << "Name: " << name << ", Age: "
<< age << endl;
    }
};
```

```
int main() {
    Student s1("Raja", 20); // implicit call
    Student s2=Student("Ravi",19);/Explicit call
    s1.display();
}
```

Download the CPP File

❖ A constructor with no arguments.

```
class Box {
public:
   int length;
   Box() {   // default constructor
      length = 10;
   }
};
```

- **A constructor with no arguments.**

```
class Box {
public:
    int length;
    Box(int l) {    // constructor
        length = l;
    }
};
```

- **Used to create a new object as a copy of an existing object.**

```
class Box {
public:
    int length;
    Box(const Box &b) {    // copy constructor
        length = b.length;
    }
};
```

- **Used to create a new object as a copy of an existing object.**

```
class Rectangle {
public:
    int width, height;

    Rectangle() { width = 1; height = 1; }            // default
    Rectangle(int w, int h) { width = w; height = h; } // parameterized
};
```

- **A faster and preferred way to initialize members.**

```
class Point {
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {} // initialization list
};
```

- Constructors cannot return a value.
- You can’t call a constructor like a normal function.
- If no constructor is defined, the compiler provides a default one automatically.
- Constructors can be private, useful in design patterns like Singleton.

Scope (::) Operator

- **The :: (scope resolution operator) connects a name to its scope — it tells the compiler where to find or define something.**

Scope (::) Operator

- When we **declare** a member function **inside a class**, we can later **define** it **outside** using the :: operator.

:: Operator

Class definition

name

```
class Course
{
        private:
        string code;
        string name;
        float Marks;
        char Grade;
        public:
        Course(string c="MA0001",string n="AVBACA",float m=0.0,char G='S')
        {
                code=c;
                name=n;
                Marks=m;
                Grade=G;
        }
        void CalculateGrade();
        void Print();

};
```

Members of the Class

Constructors of the Class

Only functions are declared

functions are defined outside the class

```
void Course::Print()
{
            cout<<"Course Code   : "<<this->code<<endl;
            cout<<"Course Name   : "<<this->name<<endl;
            cout<<"Marks Scored  : "<<this->Marks<<endl;
            cout<<"Grade : "<<this->Grade<<endl;
}

void Course::CalculateGrade()
{
            if(this->Marks>90)
            this->Grade='S';
            else if(this->Marks>80 && this->Marks<90)
            this->Grade='A';
            //Complete the rest
            else
            this->Grade='F';

}
```

```
int main()
{
            Course Course1("MA5191", "Programming Laboratory",95.0,'A');
            Course1.CalculateGrade();
            Course1.Print();
            return 0;

}
```

this → pointer

- **In C++, every non-static member function of a class has access to a special hidden pointer called this.**
- It points to **the object that is currently calling the function**.
- **Differentiate between class data members and local variables with the same name.**
- **Return the object itself (for function chaining).**
- **Pass the current object as an argument to another function.**

- **Think of this as the word "myself" inside the class.**

this → pointer

Class definition

name

```
class Course
{
        private:
        string code;
        string name;
        float Marks;
        char Grade;
        public:
        Course(string c="MA0001",string n="AVBACA",float m=0.0,char G='S')
        {
                code=c;
                name=n;
                Marks=m;
                Grade=G;
        }
        void CalculateGrade();
        void Print();

};
```

Members of the Class

Constructors of the Class

Only functions are declared

functions are defined outside the class

```
void Course::Print()
{
        cout<<"Course Code   : "<<this->code<<endl;
        cout<<"Course Name   : "<<this->name<<endl;
        cout<<"Marks Scored  : "<<this->Marks<<endl;
        cout<<"Grade : "<<this->Grade<<endl;
}

void Course::CalculateGrade()
{
        if(this->Marks>90)
        this->Grade='S';
        else if(this->Marks>80 && this->Marks<90)
        this->Grade='A';
        //Complete the rest
        else
        this->Grade='F';

}
```

"the name variable belonging to this object".

"the Marks variable belonging to this object".

```
int main()
{
        Course Course1("MA5191", "Programming Laboratory",95.0,'A');
        Course1.CalculateGrade();
        Course1.Print();
        return 0;

}
```

this → pointer

```
class Box {
    int length;
public:
    Box(int l=0) { length = l; }
    Box& setLength(int l) {
        this->length = l;
        return *this;  // return the current object
    }
    void show() { cout << "Length = " << length << endl; }
};

int main() {
    Box b;
    b.setLength(10).show(); // method chaining
}
```

friend function

```
class Vector
{
        private:
        int length;
        double *values;
        public:
        Vector(int N=0, double=0);                                  //Constructor
        Vector(const Vector&);                          // copy constructor
        ~Vector(){ delete [] values; }                  // destructor is defined inline
        Vector& operator=(const Vector&);               // overload assignment
        void LinSpace(double xl,double xu,int n);   //Linspace generates the list of equidistant n points between
xl and xu
        Vector operator+(const Vector &x);                          //+ operator overloading
        friend Vector operator+(const Vector &x, const Vector &y); //To add two vector
        friend Vector operator-(const Vector &x, const Vector &y); //Subtract x-y
        friend Vector operator*(const Vector &x, double c);                     //c*x
        friend Vector operator*(double c, const Vector &x);                     //x*yy
        double& operator[](int i) const { return values[i]; } // eg v[i] = 10
        double norm();
        double norm2();
        double norm(int p);
        double infnorm();
        int size() const { return length; }               // return length of vector
        void print();                                   //prints the vector
        void save(char *);                              //prints the vector in a file
};
```

`friend` function

A **friend function** is a function **that is not a member** of a class but is **allowed to access the class's private and protected members**.

👉 In short, **it's an outsider with special access** granted by the class.

Normally, private members of a class cannot be accessed directly from outside.
But sometimes, we want **two or more classes** or **a standalone function** to **work closely together** and share internal data.
Example use cases:

- Operator overloading (+, ==, etc.)
- Accessing data from two different classes simultaneously
- Debugging or utility functions that need internal details

`friend` function

A **friend function** is a function **that is not a member** of a class but is **allowed to access the class's private and protected members**.

👉 In short, **it's an outsider with special access** granted by the class.

Think of a **friend function** like a **trusted guest** —
It's **not part of the family (class)**,
but it's **trusted enough to enter private rooms (private data)**.

`friend` function

A **friend function** is a function **that is not a member** of a class but is **allowed to access the class's private and protected members**.

👉 In short, **it's an outsider with special access** granted by the class.

Rule	Description
Declared using friend keyword	Inside class definition
Not called with dot (.) operator	Called like a normal function
Accesses private/protected data	If declared as a friend
Friendship is **not mutual**	If A is friend of B, B isn't automatically friend of A
Friendship is **not inherited**	Derived class doesn't inherit friendship

`friend` function

A **friend function** is a function **that is not a member** of a class but is **allowed to access the class's private and protected members**.

👉 In short, **it's an outsider with special access** granted by the class.

Feature	Friend Function	Member Function
Belongs to class	❌ No	✅ Yes
Access to private data	✅ Yes (if declared friend)	✅ Yes
Called using object	❌ No	✅ Yes
Uses this pointer	❌ No	✅ Yes

friend function

```
class Vector
{
                private:
                int length;
                double *values;
                public:
                Vector(int N=0, double=0);                                          //Constructor
                Vector(const Vector&);                      // copy constructor
                ~Vector(){ delete [] values; }              // destructor is defined inline
                Vector& operator=(const Vector&);           // overload assignment
                void LinSpace(double xl,double xu,int n);   //Linspace generates the list of equidistant n points between xl and xu
                Vector operator+(const Vector &x);                                  //+ operator overloading
                friend Vector operator+(const Vector &x, const Vector &y); //To add two vector
                friend Vector operator-(const Vector &x, const Vector &y); //Subtract x-y
                friend Vector operator*(const Vector &x, double c);                   //c*x
                friend Vector operator*(double c, const Vector &x);                   //x*yy
                double& operator[](int i) const { return values[i]; } // eg v[i] = 10
                double norm();
                double norm2();
                double norm(int p);
                double infnorm();
                int size() const { return length; }                 // return length of vector
                void print();                                               //prints the vector
                void save(char *);                                          //prints the vector in a file
};
```

- They are **not member functions**.
- They **can access private members** (length and values) of Vector.
- They allow **flexible operations**, e.g., 2.0 * v1 (scalar first) which **cannot be done easily with member functions**.

friend function

```
Vector operator+(const Vector &x, const Vector &y) {
    // assume same length
    Vector result(x.length);          // create a vector for result
    for(int i=0; i<x.length; i++) {
        result.values[i] = x.values[i] + y.values[i]; // access private 'values'
    }
    return result;
}
```

- Even though `values` is **private**, the friend function can access it directly.
- If this were **not a friend**, you would need **getter functions** for `values`.

friend function

```
Vector operator*(const Vector &x, double c) {
    Vector result(x.length);
    for(int i=0; i<x.length; i++) {
        result.values[i] = x.values[i] * c;
    }
    return result;
}

Vector operator*(double c, const Vector &x) {
    return x * c; // reuse the previous operator
}
```

- Friend functions **create a new Vector object** for the result.
- The function directly accesses the **private values array**, so it's efficient.
- You **avoid getters and setters**, keeping syntax clean: v3 = v1 + v2;.

Getters and Setters Methods

```
class Student {
private:
    string name;
    int age;

public:
    // Setter for name
    void setName(string n) {
        name = n;
    }

    // Setter for age (with validation)
    void setAge(int a) {
        if(a > 0) age = a;
        else cout << "Age must be positive!" << endl;
    }

// Getter for name
    string getName() const {
        return name;
    }
    // Getter for age
    int getAge() const {
        return age;
    }
};
```

setter

getter

```
int main() {
    Student s;

    s.setName("Raja");   // setter
    s.setAge(20);          // setter
    cout << s.getName() << " is " << s.getAge()
<< " years old." << endl; // getter

    s.setAge(-5); // invalid, setter prevents it
    s.age=10; // compiler error
//error: 'std::string Student::name' is private
within this context
}
```

- **Getters and setters must be used outside class to access data attributes**

Point Class

```
int main()
{
        Point P1,P2;
        P1.SetPoint(1.0,2.0,3.0);

        P1.Print();
        P2.SetPoint(3.0,4.0,5.0);
        P2.Print();
        cout<<P1.distance(P1,P2)<<endl;
        cout<<P1.distanceFromOrigin(P1)<<endl;
        cout<<P2.distanceFromOrigin(P2)<<endl;
        cout<<P1.distanceFromOrigin()<<endl;
        cout<<P2.distanceFromOrigin()<<endl;
        Point P3=P1.MidPoint(P1,P2);
        P3.Print();

        //cout<<distance(P1,P2)<<endl;
        cout<<distanceFromOrigin(P1)<<endl;
        cout<<distanceFromOrigin(P2)<<endl;
        Point P4=MidPoint(P1,P2);
        Print(P4);

        Point P5=P1+P2;
        Print(P5);
        return 0;
}
```

Download the CPP File

```
class Point{
        private:
        double x, y, z;
        public:
        Point(double x=0.0, double y=0.0, double z=0.0);
        void SetPoint(double x, double y, double z);
        Point GetPoint();
        void SetX(double x);
        void SetY(double y);
        void SetZ(double z);
        double GetX();
        double GetY();
        double GetZ();
        void Print();
        double distance(Point P1, Point P2);
        double distanceFromOrigin(Point P);
        double distanceFromOrigin();
        Point MidPoint(Point P1, Point P2);
        const Point operator+(const Point &P) const;
};
```

Download the HPP File

Download the CPP File

```
class Vector
{
        private:
        int length;
        double *values;
        public:
        Vector(int N=0, double=0);                              //Constructor
        Vector(const Vector&);                      // copy constructor
        ~Vector(){ delete [] values; }              // destructor is defined inline
        Vector& operator=(const Vector&);           // overload assignment
        void LinSpace(double xl,double xu,int n);   //Linspace generates the list of equidistant n points between xl and xu
        Vector operator+(const Vector &x);                      //+ operator overloading
        friend Vector operator+(const Vector &x, const Vector &y); //To add two vector
        friend Vector operator-(const Vector &x, const Vector &y); //Subtract x-y
        friend Vector operator*(const Vector &x, double c);                     //c*x
        friend Vector operator*(double c, const Vector &x);                     //x*yy
        double& operator[](int i) const { return values[i]; } // eg v[i] = 10
        double norm();
        double norm2();
        double norm(int p);
        double infnorm();
        int size() const { return length; }                 // return length of vector
        void print();                                                           //prints the vector
        void save(char *);  //prints the vector in a file
};
```

```
class Matrix
{
        private:
        double *values;
        int row, col;
        public:
        Matrix(const Matrix& A);
        Matrix(double *val, int r, int c)//Constructor
        {
                row=r;
                col=c;
                values=new double[row*col];
                for(int i=0;i<r;i++)
                for(int j=0;j<c;j++)
                values[i*c+j]=val[i*c+j];
        }
        Matrix(int M=0, int N=0, double x=0);

        friend Matrix operator*(      Matrix &A,          Matrix &B);
        friend Matrix operator+(  const Matrix &A, const Matrix &B);
        friend Matrix operator-(  const Matrix &A,  const Matrix &B);
        friend Matrix operator*(double c,   const Matrix &A);
        double& operator[](int i) const;
        friend Vector operator*(const Matrix &A,   Vector &x);
        Matrix& operator=(const Matrix& );

        ~Matrix();//destructor
}
```

[Download the HPP File]

[Download the CPP File]