1. Constant Data Member

A const data member is a member of a class whose value cannot be modified once it is ini alized.

A const data member must be ini alized at the me of its construc on. This can be done using:

- Constructor ini aliza list.
- Default member ini alizers

The value of a const data member cannot be modified a er ini aliza on.

Each instance of the class can have its own unique value for the const data member, as it's initalized by the constructor.

For example :
```
#include <iostream> using

namespace std; class MyClass {     const int

myConst;  // Constant data member public:

    // Constructor to ini alize the const data member

MyClass(int value) : myConst(value) {}     void display()

const {         cout << "The value of myConst is: " <<

myConst << endl;

    }

};

int     main()     {

MyClass    obj1(10);

obj1.display()

MyClass    obj2(20);

obj2.display();

return 0;

}
```

OUTPUT FOR THIS CODE -

THE VALUE OF MYCONST IS:

10

THE VALUE OF MYCONST IS:
20

2. Constant Member func on

Constant member func ons are those func ons that are denied permission to change the values of the data members of their class. To make a member func on constant, the keyword const is appended to the func on prototype and also to the func on defini on header.

for example:

```
return_type func on_name () const
{
    //func on body
}
```

Some Important Points -

1. When a func on is declared as const, it can be called on any type of object, const object as well as non-const objects.

2. Whenever an object is declared as const, it needs to be ini alized at the me of declara on. however, the object ini aliza on while declaring is possible only with the help of constructors.

3. A func on becomes const when the const keyword is used in the func on's declara on. The idea of const func ons is not to allow them to modify the object on which they are called.

4. It is recommended prac ce to make as many func ons const as possible so that accidental changes to objects are avoided.

Example code:

```
// C++ program to demonstrate that data members can be
// updated in a member func on that is not constant.
#include  <iostream>
using  namespace  std;
class Demo {
    int x;
public:
    void set_data(int a) { x = a; }
    // non const member func on
// data can be updated     int
get_data()
    {
++x;
return x;
    }
};
main()
{
    Demo d;
   d.set_data(10);
cout   <<   d.get_data();
return 0;
}
```

Output –

11

3. Sta c data member

sta c data members and sta c member func ons are features that belong to the class rather than any specific object of the class

- A sta c data member is a variable shared among all objects of the class.
- It is allocated only once, no ma er how many objects of the class are created.
- A sta c data member is declared inside the class defini on but must be defined outside the class if it's not ini alized in the class itself.
- Shared Memory: A sta c data member is common for all instances of the class.
- Storage: Memory is allocated only once, in the global data segment.
- Access:
- Can be accessed using the class name (ClassName::member) or through an object.
- O en ini alized outside the class defini on.
- Life me: It exists throughout the life me of the program.

Code example-

```
#include <iostream>
using namespace std;
class Test {     sta c int count; //
Declara on public:
   Test() { count++; }  // Increment sta c member     sta c int
getCount() { return count; } // Accessor for sta c member
};
// Defini on of sta c data member
int Test::count = 0; int
main() {
   Test t1, t2, t3;        cout << "Number of objects: " <<
Test::getCount() << endl;     return 0;
}
```

Output- Number of objects: 3

4. Sta c Member Func on

- A sta c member func on can access only sta c data members or other sta c member func ons.
- It does not have access to the this pointer because it does not operate on an object instance.

Key Points:

1. No Object Required: Can be called using the class name or an object of the class.
2. Access: It can access only sta c members of the class.
3. U lity: O en used to perform opera ons that are independent of specific object instances.

Code example-

```
#include <iostream>
using namespace std;
class Test {     sta c int count; // Sta c
data member public:
    Test() { count++; }     sta c void displayCount() { //
Sta c member func on        cout << "Count: " << count
<< endl;
    }
};

int Test::count = 0; // Define sta c data member int
main() {
    Test t1, t2;
    Test::displayCount(); // Access using class name
t1.displayCount();    // Access using object     return
0;
}
```

Output-

```
Count: 2
```

5.

Feature	Static Data Member	Static Member Function
Purpose	Shared variable for all objects	Function that doesn't need an object instance
Storage	Global data segment	Stored in code segment
Access	Using class name or object	Using class name or object
Scope	Class-wide	Class-wide
`this` **Pointer**	Not used	Not available

6. Polymorphism

Polymorphism in C++ is a fundamental concept in object-oriented programming (OOP) that allows en es like func ons, operators, or objects to take on mul ple forms. It provides flexibility and reusability in the code by allowing a single interface to represent different underlying forms (data types or behaviors).

Types of Polymorphism in C++

1. Compile-Time Polymorphism (Sta c Polymorphism): This type of polymorphism is resolved at compile me and is achieved using:
   - Func on Overloading: Func ons with the same name but different parameter lists.

Example-
```
class
Example {
public:
   void show(int x) {        cout <<
"Integer: " << x << endl;
   }
   void show(double y) {        cout
<< "Double: " << y << endl;
   }
};
```

Operator Overloading: Customizing the behavior of operators for user-defined types.

Example-
```
class Complex { public:
   int real, imag;
   Complex operator+(const Complex& obj)
{      Complex res;      res.real = this->real
+ obj.real;         res.imag = this->imag +
obj.imag;       return res;
   }};
```

Run-Time Polymorphism (Dynamic Polymorphism): This type of polymorphism is resolved at run me and is achieved using:

- Func on Overriding: Redefining a base class func on in a derived class.
- Virtual Func ons and Pointers: Virtual func ons allow overriding in derived classes and support dynamic binding.

```
class Base {
public:
   virtual void display() {        cout << "Base
class display func on" << endl;
   }};
class Derived : public Base {
public:
   void display() override {        cout << "Derived
class display func on" << endl;
   }};
int main() {
  Base*  basePtr;        Derived  derivedObj;        basePtr  =
&derivedObj;     basePtr->display(); // Outputs: Derived class
display func on     return 0;
}
```

Output-

Derived class display func on

- Compile- me polymorphism is faster but less flexible.
- Run- me polymorphism provides flexibility via dynamic binding but incurs a slight performance overhead.
- Virtual func ons require a vtable (virtual table) mechanism.

Polymorphism enhances code maintainability, scalability, and flexibility by enabling a single interface to work with different data types or behaviors.

7. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile- me polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithme c operators may be overloaded are Complex Numbers, Frac onal Numbers, Big integers, etc.

Example:

int a; float
b,sum; sum
= a + b;

Here, variables “a” and “b” are of types “int” and “float”, which are built-in data types. Hence the addi on operator ‘+’ can easily add the contents of “a” and “b”. This is because the addi on operator “+” is predefined to add variables of built-in data type only.

Implementa on:

// C++ Program to Demonstrate

// Operator Overloading

```
#include <iostream>

using namespace std;

class Complex {

private:

    int real, imag;

public:

    Complex(int r = 0, int i = 0)

    {        real

= r;

imag = i;

    }


    // This is automa cally called when '+' is used with

    // between two Complex objects

    Complex operator+(Complex const& obj)

    {

        Complex res;

       res.real  =  real  +  obj.real;

res.imag  =  imag  +  obj.imag;

return res;

    }
```

```
    void print() { cout << real << " + i" << imag << '\n'; }
};
int main()
{
    Complex c1(10, 5), c2(2,
4);     Complex c3 = c1 + c2;
c3.print(); }
```

Output –

12+i9

Operators that can be Overloaded in C++

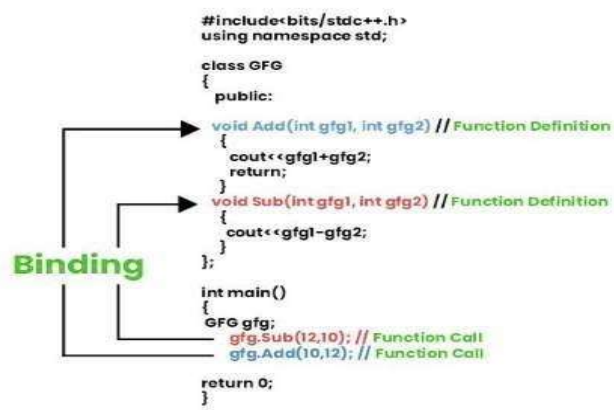
We can overload

- Unary operators
- Binary operators
- Special operators ([ ], (), etc)

8. Dynamic Binding

Dynamic binding in C++ is a prac ce of connec ng the func on calls with the func on defini ons by avoiding the issues with sta c binding, which occurred at build me. Because dynamic binding is flexible, it avoids the drawbacks of sta c binding, which connected the func on call and defini on at build me.

In simple terms, Dynamic binding is the connec on between the func on declara on and the func on call.



Usage of Dynamic Binding

It is also possible to use dynamic binding with a single func on name to handle mul ple objects. Debugging the code and errors is also made easier and complexity is reduced with the help of Dynamic Binding.

Static Binding	Dynamic Binding
It takes place at compile time which is referred to as early binding	It takes place at runtime so it is referred to as late binding
Execution of static binding is faster than dynamic binding because of all the information needed to call a function.	Execution of dynamic binding is slower than static binding because the function call is not resolved until runtime.
It takes place using normal function calls, operator overloading, and function overloading.	It takes place using virtual functions
Real objects never use static binding	Real objects use dynamic binding

9.Virtual func ons

A virtual func on is a member func on declared in a base class and re-declared in a derived class (overridden). You can execute the virtual func on of the derived class when you refer to its object using a pointer or reference to the base class. The concept of dynamic binding is implemented with the help of virtual func ons.

Example: #include

```
<iostream> using

namespace std; class

Base { public:

   // Func on that calls print     void

callFunc on() { print(); }     // Virtual

func on to be overridden     virtual

void print() {

        cout << "Prin ng the Base class content" << endl;

    }

};

// Derived class inheri ng Base publicly

class Derived : public Base { public:

    void print() override { // Derived's implementa on of print

cout << "Prin ng the Derived class content" << endl;

    }

};

int main() {
```

```
    Base baseObj;          // Crea ng an object of Base
baseObj.callFunc on(); // Calling callFunc on on Base object
Derived derivedObj;     // Crea ng an object of Derived
derivedObj.callFunc on(); // Calling callFunc on on Derived object

    return 0;
}
```

Output-

Prin ng the Base class content

Prin ng the Base class content