1. Constant Data Member

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

A const data member must be initialized at the time of its construction. This can be done using:

- Constructor initialization list.
- Default member initializers

The value of a const data member cannot be modified after initialization.

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

For example :

```
#include <iostream>

using namespace std;

class MyClass {

  const int myConst;  // Constant data member

public:

  // Constructor to initialize 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 function

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

for example:

```
return_type function_name () const

{

    //function body

}
```

Some Important Points -

1. When a function 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 initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

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

4. It is recommended practice to make as many functions 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 function that is not constant.
#include <iostream>
using namespace std;
class Demo {
   int x;
public:
   void set_data(int a) { x = a; }
   // non const member function
   // 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. Static data member

static data members and **static member functions** are features that belong to the class rather than any specific object of the class

 A **static data member** is a variable shared among all objects of the class.

 It is allocated only once, no matter how many objects of the class are created.

 A static data member is declared inside the class definition but must be defined outside the class if it's not initialized in the class itself.

 Shared Memory: A static 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.
- Often initialized outside the class definition.

 Lifetime: It exists throughout the lifetime of the program.

Code example-

```
#include <iostream>

using namespace std;

class Test {

   static int count; // Declaration

public:

   Test() { count++; }  // Increment static member

   static int getCount() { return count; } // Accessor for static member

};

// Definition of static 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. Static Member Function

- A **static member function** can access only **static data members** or other **static member functions**.
- It does not have access to the this pointer because it does not operate on an object instance.

Key Points:

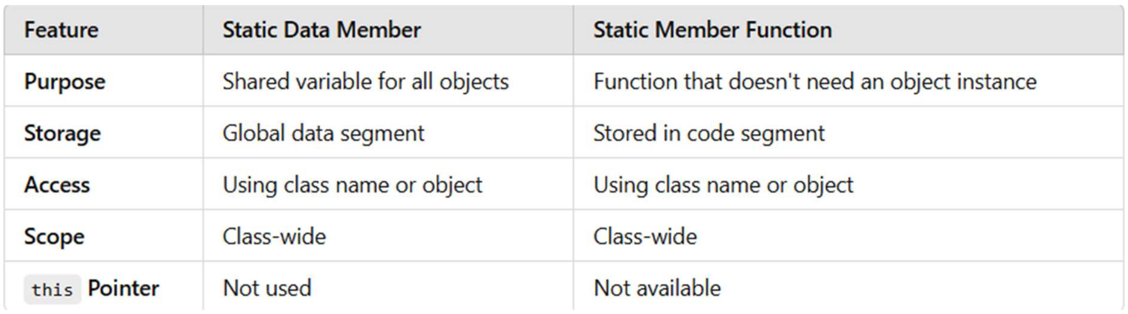
1. **No Object Required**: Can be called using the class name or an object of the class.
2. **Access**: It can access only static members of the class.
3. **Utility**: Often used to perform operations that are independent of specific object instances.

Code example-

```
#include <iostream>

using namespace std;

class Test {

  static int count; // Static data member

public:

  Test() { count++; }

  static void displayCount() { // Static member function

    cout << "Count: " << count << endl;

  }

};


int Test::count = 0; // Define static data member

int main() {

  Test t1, t2;

  Test::displayCount(); // Access using class name

  t1.displayCount();    // Access using object

  return 0;

}
```

Output-

```
Count: 2
```



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

5. Polymorphism

Polymorphism in C++ is a fundamental concept in object-oriented programming (OOP) that allows entities like functions, operators, or objects to take on multiple 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 (Static Polymorphism):** This type of polymorphism is resolved at compile time and is achieved using:
   - **Function Overloading:** Functions 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 runtime and is achieved using:

- **Function Overriding:** Redefining a base class function in a derived class.
- **Virtual Functions and Pointers:** Virtual functions allow overriding in derived classes and support dynamic binding.

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

Output-

```
Derived class display function
```

- **Compile-time polymorphism** is faster but less flexible.
- **Run-time polymorphism** provides flexibility via dynamic binding but incurs a slight performance overhead.
- Virtual functions 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.

6. 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-time 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 arithmetic operators may be overloaded are Complex Numbers, Fractional 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 addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

Implementation:

```
// 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 automatically 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++

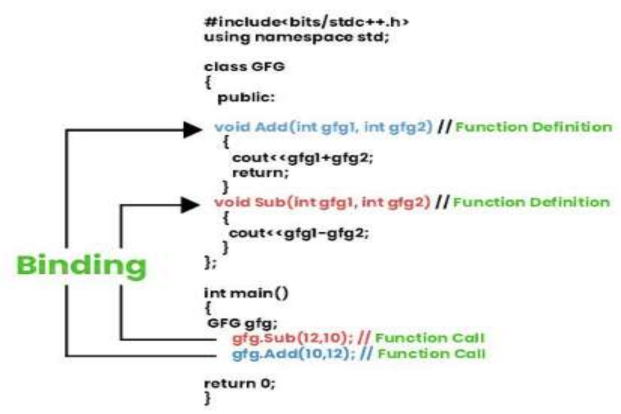
We can overload

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

7. Dynamic Binding

Dynamic binding in C++ is a practice of connecting the function calls with the function definitions by avoiding the issues with static binding, which occurred at build time. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

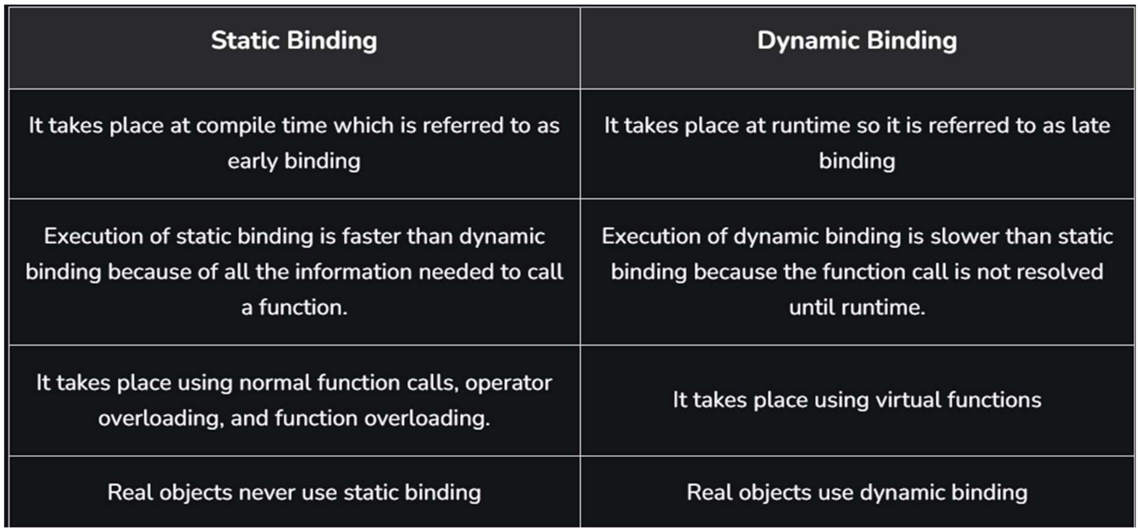
In simple terms, Dynamic binding is the connection between the function declaration and the function call.



```
#include<bits/stdc++.h>
using namespace std;

class GFG
{
  public:
  void Add(int gfg1, int gfg2) // Function Definition
   {
    cout<<gfg1+gfg2;
    return;
   }
  void Sub(int gfg1, int gfg2) // Function Definition
   {
   cout<<gfg1-gfg2;
   }
};

int main()
{
 GFG gfg;
    gfg.Sub(12,10); // Function Call
    gfg.Add(10,12); // Function Call

return 0;
}
```

Usage of Dynamic Binding

It is also possible to use dynamic binding with a single function name to handle multiple 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

8.Virtual functions

A virtual function is a member function declared in a base class and re-declared in a derived class (overridden). You can execute the virtual function 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 functions.

Example:

```
#include <iostream>

using namespace std;

class Base {

public:

   // Function that calls print

   void callFunction() { print(); }

   // Virtual function to be overridden

   virtual void print() {

      cout << "Printing the Base class content" << endl;

   }

};

// Derived class inheriting Base publicly

class Derived : public Base {

public:

   void print() override { // Derived's implementation of print

      cout << "Printing the Derived class content" << endl;

   }

};
```

```
int main() {
    Base baseObj;          // Creating an object of Base
    baseObj.callFunction(); // Calling callFunction on Base object
    Derived derivedObj;     // Creating an object of Derived
    derivedObj.callFunction(); // Calling callFunction on Derived object

    return 0;
}
```

Output-

```
Printing the Base class content
Printing the Base class content
```