MOD 7

Polymorphism in C++

- **Polymorphism** = *"many forms"* (Greek: *poly* = many, *morphos* = forms).
- Allows the **same name** to perform **multiple tasks**.
- **Real-life example**: A woman can be a daughter, sister, wife, and mother – all roles with different behaviors.

🔹 **Types of Polymorphism**

1. **Compile-Time Polymorphism (Static / Early Binding)**
    - Achieved via:
        - **Function Overloading**
        - **Operator Overloading**
2. **Run-Time Polymorphism (Dynamic Binding)**
    - Achieved via:
        - **Virtual Functions**

🔹 **Function Overloading**

- Same function name with **different parameter list**.
- Helps perform **similar operations** with different data.
- **Return type is NOT** considered for overloading.

✅ **Code Example 1: Class-based add() Overloading**

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

class Cal {
public:
   int add(int a, int b) {
      return a + b;
   }
```

```
    int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Cal C;
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}
```

✅ **Code Example 2: Global mul() Function Overloading**

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

int mul(int a, int b) {
    return a * b;
}
float mul(float x, int y) {
    return x * y;
}

int main() {
    int r1 = mul(6, 7);
    float r2 = mul(0.2f, 3);
    cout << "r1 is: " << r1 << endl;
    cout << "r2 is: " << r2 << endl;
    return 0;
}
```

🔹 **Function Call Resolution (Compile Time)**

- Compiler selects the correct overloaded function based on:
  - **Exact match**
  - **Integral promotions** (e.g., char to int)
  - **Implicit conversions** (e.g., float to double)
- If multiple matches → **Ambiguity Error**.

⚠️ **Ambiguity Example: square(10)**

long square(long n);

long square(double x);

// Call: square(10); → ambiguous: 10 can be long or double

🔹 **Advantages of Function Overloading**

- Faster execution.
- Clean, readable code.
- Easy maintenance & memory-efficient.
- Improves code reusability & flexibility.

🔹 **Disadvantages of Function Overloading**

- Can't overload by **return type only**.
- **Static** member functions can't be overloaded if another function has the same name & parameters.

🔹 **Overloading Ambiguities**

1. ❌ Type Conversion Ambiguity

#include<iostream>

using namespace std;

void function(float) { cout << "Data Type: float\n"; }

```
void function(int)   { cout << "Data Type: int\n"; }

int main() {
    function(1.0); // Ambiguous: double → float or int?
    function(1);   // Valid
    return 0;
}
```

---

2. ❌ Default Arguments Ambiguity

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

int add(int a) {
    int b = 10;
    return a + b;
}
int add(int a, int b = 10) {
    return a + b;
}

int main() {
    int a = 5;
    cout << "a + b = " << add(a) << endl; // Ambiguous
    return 0;
}
```

---

3. ❌ Pass by Reference Ambiguity

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

```
void display(int a) {
    cout << "a = " << a << endl;
}
void display(int &a) {
    cout << "a = " << a << endl;
}

int main() {
    int a = 5;
    display(a); // Ambiguous: pass-by-value or pass-by-ref?
    return 0;
}
```

- **Overloading main() Function (Yes, possible within a class)**

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

class Mainclass {
public:
    int main(int a) {
        cout << "a = " << a << endl;
        return 0;
    }
    int main(int a, int b) {
        cout << "a = " << a << "; b = " << b << endl;
        return 0;
    }
};

int main() {
```

```
    Mainclass object;
    object.main(5);
    object.main(5, 10);
    return 0;
}
```

📌 Operator Overloading - Overview

- **Definition:** Operator overloading allows C++ operators to be redefined and used for user-defined types (classes/objects).
- **Type:** Compile-time polymorphism.

⚙️ Why Use Operator Overloading?

- To make operators work with **user-defined types**, just like they do with built-in types.
- Example: You can use + with objects of a class like a1 + a2.

❌ Operators That Cannot Be Overloaded

1. Scope resolution ::
2. sizeof
3. Member selector .
4. Pointer-to-member selector .*
5. Ternary conditional ?:

Syntax

```
return_type class_name::operator op(argument_list) {
  // function body
}
```

✅ Rules for Operator Overloading

- Only existing operators can be overloaded.
- At least one operand must be a **user-defined** type.
- **Unary Operators**:

- If overloaded as member → no arguments
- If overloaded as friend → one argument

- **Binary Operators**:
  - As member → one argument
  - As friend → two arguments

📋 **Examples**

1. Unary Operator Overload

```
void operator++()       // Prefix
void operator++(int)    // Postfix (int is a dummy parameter)
```

2. Binary Operator (+) Overload

```
void A::operator+(A a) {
  int m = x + a.x;
  cout << "Addition: " << m;
}
```

3. I/O Operator Overloading

- **Insertion (<<)** using ostream:

```
friend ostream& operator<<(ostream &out, const student &s);
```

- **Extraction (>>)** using istream:

```
friend istream& operator>>(istream &in, employee &e);
```

4. Comparison Operator <

```
int operator<(const distance& d) {
  return (feet == d.feet && inches < d.inches);
}
```

5. Subscript Operator []

```
int& operator[](int i) {
```

```
  if (i > SIZE) {
    cout << "Index out of bounds";
    return arr[0];
  }
  return arr[i];
}
```

6. Function Call Operator ()

```
int operator()(int x) {
  return x * factor;
}
```

🚫 **Operators That Cannot Be Overloaded with friend Functions**

- =
- ()
- []
- ->

✅ **1. Unary Operator Overloading (++)**

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

class Test {
private:
    int num;

public:
    Test() : num(8) {}

    // Overloading prefix increment
    void operator++() {
```

```
        num = num + 2;
    }

    // Overloading postfix increment
    void operator++(int) {
        num = num + 2;
    }

    void Print() {
        cout << "The Count is: " << num << endl;
    }
};

int main() {
    Test tt;
    ++tt;       // Calls prefix
    tt.Print();
    tt++;       // Calls postfix
    tt.Print();
    return 0;
}
```

✅ **2. Binary Operator Overloading (+)**

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

class A {
private:
    int x;
```

```
public:
    A() : x(0) {}
    A(int i) : x(i) {}

    void operator+(A a) {
        int m = x + a.x;
        cout << "Addition of two objects is: " << m << endl;
    }

    void display() {
        cout << "x = " << x << endl;
    }
};

int main() {
    A a1(5), a2(4);
    a1 + a2;  // Equivalent to: a1.operator+(a2)
    return 0;
}
```

✅ 3. Overloading I/O Operators (<< and >>)

📘 a. Student (Output only)

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

class Student {
private:
    int roll_no;
    float marks;
```

```
public:
    Student(int r, float m) : roll_no(r), marks(m) {}

    friend ostream& operator<<(ostream& out, Student& st) {
        out << "Roll No: " << st.roll_no << endl;
        out << "Marks: " << st.marks << endl;
        return out;
    }
};

int main() {
    Student s1(1, 490), s2(2, 600);
    cout << "Record of Student 1:\n" << s1;
    cout << "Record of Student 2:\n" << s2;
    return 0;
}
```

📘 **b. Employee (Input and Output)**

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

class Employee {
private:
    int emp_no;
    char name[10], address[40];

public:
    friend istream& operator>>(istream& in, Employee& emp) {
        cout << "Enter Employee No: ";
        in >> emp.emp_no;
        cout << "Enter Employee Name: ";
```

```
        in >> emp.name;
        cout << "Enter Employee Address: ";
        in >> emp.address;
        return in;
    }

    friend ostream& operator<<(ostream& out, Employee& emp) {
        out << "Emp No: " << emp.emp_no << endl;
        out << "Name: " << emp.name << endl;
        out << "Address: " << emp.address << endl;
        return out;
    }
};

int main() {
    Employee e1, e2;
    cout << "Enter record for Employee 1:\n";
    cin >> e1;
    cout << "Enter record for Employee 2:\n";
    cin >> e2;

    cout << "\nRecord of Employee 1:\n" << e1;
    cout << "Record of Employee 2:\n" << e2;
    return 0;
}
```

✅ **4. Overloading Comparison Operator (<)**

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

```
class Distance {
private:
    int feet;
    int inches;

public:
    Distance(int f, int i) : feet(f), inches(i) {}

    void display() {
        cout << "Feet: " << feet << "\nInches: " << inches << endl;
    }

    int operator<(const Distance& d) {
        return (feet == d.feet && inches < d.inches);
    }
};

int main() {
    Distance d1(11, 10), d2(5, 11);
    cout << "Values of d1:\n";
    d1.display();
    cout << "Values of d2:\n";
    d2.display();

    if (d1 < d2)
        cout << "d1 is less than d2\n";
    else
        cout << "d2 is less than d1\n";

    return 0;
```

```
}
```

✅ 5. Overloading Subscript Operator ([])

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

const int SIZE = 10;

class SaferArray {
private:
    int arr[SIZE];

public:
    SaferArray() {
        for (int i = 0; i < SIZE; i++)
            arr[i] = i;
    }

    int& operator[](int i) {
        if (i >= SIZE) {
            cout << "Index out of bounds!" << endl;
            return arr[0]; // Safe fallback
        }
        return arr[i];
    }
};

int main() {
    SaferArray a;
    cout << "Value of a[2]: " << a[2] << endl;
```

```
    cout << "Value of a[5]: " << a[5] << endl;
    cout << "Value of a[12]: " << a[12] << endl;
    return 0;
}
```

✅ 6. Overloading Function Call Operator (())

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

class Multiply {
private:
    int factor;

public:
    Multiply(int f) : factor(f) {}

    // Overload function call operator
    int operator()(int x) {
        return x * factor;
    }
};

int main() {
    Multiply m1(5);
    cout << "5 * 3 = " << m1(3) << endl;
    cout << "5 * 10 = " << m1(10) << endl;
    return 0;
}
```

❗ Operators That Cannot Be Overloaded

- :: (Scope Resolution)
- sizeof
- . (Member Selector)
- .* (Member Pointer Selector)
- ?: (Ternary Operator)

Dynamic (Run-Time) Polymorphism

🟦 Static Binding

- **Overloaded functions** are selected at compile time by matching the number and type of arguments.
- This is called:
  - Early Binding
  - Static Linking
  - Compile-Time Polymorphism

🟦 Dynamic Binding (Run-Time Polymorphism)

- Function to be executed is determined at **runtime**.
- Achieved through:
  1. **Function Overriding**
  2. **Base Class Pointers or References**

🟩 Example: Without Virtual Function (No Polymorphism)

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

class Base {
public:
    void show() {
        cout << "In Base\n";
```

```
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "In Derived\n";
    }
};

int main() {
    Base* bp = new Derived;
    bp->show();  // Output: In Base (No polymorphism)
    return 0;
}
```

🟩 **Example: With Virtual Function (Run-Time Polymorphism)**

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

class Base {
public:
    virtual void show() {
        cout << "In Base\n";
    }
};

class Derived : public Base {
public:
    void show() override {
```

```
        cout << "In Derived\n";
    }
};

int main() {
    Base* bp = new Derived;
    bp->show();  // Output: In Derived
    return 0;
}
```

🟦 Pointers to Functions (Callback Functions)

- Function pointers allow selection of functions dynamically at run-time.
- Useful for:
    - Event-driven programming
    - Callbacks
    - Storing function pointers in arrays

Syntax

```
data_type (*pointer_name)(parameter_list);
```

Example

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

int add(int a, int b) {
    return a + b;
}

int main() {
    int (*funcptr)(int, int) = add;
    int sum = funcptr(5, 5);
    cout << "Value of sum is: " << sum;
```

```
    return 0;
}
```

🟦 Pointer to Objects

```
class Apple {
public:
    int x;
};

int main() {
    Apple a;
    Apple* aptr = &a;
    aptr->x = 10;       // Using arrow operator
    (*aptr).x = 20;     // Using dereference

    Apple* bptr = new Apple;
    bptr->x = 40;

    Apple* cptr = new Apple[10];
    cptr[0].x = 25;

    return 0;
}
```

🟦 Constant Pointers

```
int x = 10;
int* const pv = &x;  // Constant pointer (can't change address)

int y = 20;
const int* pv2 = &y; // Pointer to constant (can't change value)
```

🟦 Virtual Functions / Function Overriding

- A **virtual function** is a base class function overridden by the derived class.
- Enables **runtime polymorphism** using base class pointers/references.

Rules

- Must be declared virtual in the base class.
- Must have same name, return type, and parameters in derived class.
- Called using **base class pointer/reference**.

🟩 Virtual Function Example

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

class Base {
public:
    virtual void print() {
        cout << "Print Base Class\n";
    }

    void show() {
        cout << "Show Base Class\n";
    }
};

class Derived : public Base {
public:
    void print() override {
        cout << "Print Derived Class\n";
    }
```

```
    void show() {
        cout << "Show Derived Class\n";
    }
};

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

    bptr->print(); // Runtime binding
    bptr->show();  // Compile-time binding

    return 0;
}
```

🟦 Abstract Class

- A class that **cannot be instantiated**.
- Contains at least **one pure virtual function**.

Syntax

```
class Base {
public:
    virtual void fun() = 0;  // Pure virtual function
};
```

Example

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

class Base {
public:
```

```
    virtual void fun() = 0;
};

class Derived : public Base {
public:
    void fun() override {
        cout << "fun() called\n";
    }
};

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

🟦 Function Overloading vs. Overriding

Aspect	Function Overloading	Function Overriding
Definition	Same name, different parameters	Same name, parameters, return type
Resolution Time	Compile-time	Run-time
Virtual Keyword	Not required	Required in base class
Base Class Pointer	Not needed	Required for polymorphism

🟦 Virtual Base Class (Hybrid Inheritance Ambiguity Resolution)

- Solves ambiguity in multiple inheritance using virtual keyword.

Structure

```
  A
 / \
B   C
```

```
  \ /
   D
```

Example

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

class A {
public:
   void show() {
      cout << "Class A\n";
   }
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
   D obj;
   obj.show();  // No ambiguity
   return 0;
}
```

MOD 8

🧠 **Generic Programming in C++**

✅ **What is Generic Programming?**

- Programming paradigm where **generic types** are used in algorithms to work with **multiple data types** and data structures.
- Implemented using **Templates** in C++.

📦 Templates

➤ Definition:

- A **template** is a **blueprint** or **formula** to create functions or classes for different data types.
- Works like macros, but with **type safety** and **compiler-time expansion**.

🧰 Types of Templates

1. **Function Template**
2. **Class Template**

✅ Advantages

- Type-safe
- More reliable than macros
- Cleaner, reusable code
- Compile-time expansion

❌ Disadvantages

- Nested templates are not well supported
- Can expose all code (no encapsulation)
- Poor error messages in some compilers
- Difficult to debug and maintain

✅ 1. Function Template (Basic Example)

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

template <class X>
X func(X a, X b) {
    return a; // You can modify to return b or any operation
}

int main() {
```

```
    cout << func(15, 8) << endl;        // func(int, int)
    cout << func('p', 'q') << endl;     // func(char, char)
    cout << func(7.5, 9.2) << endl;     // func(double, double)
    return 0;
}
```

✅ 2. Function Template with Multiple Parameters

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

template <class T1, class T2>
void display(T1 x, T2 y) {
    cout << x << " " << y << "\n";
}

int main() {
    display(1999, "EBG");
    display(12.34, 1234);
    return 0;
}
```

✅ 3. Overloading of Template Functions

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

template <class T>
void display(T x) {
    cout << "Template display: " << x << "\n";
}
```

```
void display() {
    cout << "Explicit display:\n";
}

int main() {
    display(100);
    display(12.34);
    display('C');
    display();
    return 0;
}
```

✅ 4. Member Function Template (Dot Product Example)

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

template <class T>
class vector {
    T* v;
    int size;
public:
    vector(int n);              // Constructor with size
    vector(T* a, int n);        // Constructor with array
    T operator*(vector& y);     // Dot product
    ~vector();                  // Destructor
};

template <class T>
vector<T>::vector(int m) {
```

```
    size = m;
    v = new T[size];
    for (int i = 0; i < size; i++)
        v[i] = 0;
}

template <class T>
vector<T>::vector(T* a, int n) {
    size = n;
    v = new T[size];
    for (int i = 0; i < size; i++)
        v[i] = a[i];
}

template <class T>
T vector<T>::operator*(vector& y) {
    T sum = 0;
    for (int i = 0; i < size; i++)
        sum += this->v[i] * y.v[i];
    return sum;
}

template <class T>
vector<T>::~vector() {
    delete[] v;
}

int main() {
    int a[] = {1, 2, 3};
    int b[] = {4, 5, 6};
```

```
    vector<int> v1(a, 3);
    vector<int> v2(b, 3);
    cout << "Dot product using array constructor: " << v1 * v2 << endl;
    return 0;
}
```

✅ 5. Bubble Sort Using Template

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

template <class T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

template <class T>
void bubblesort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
                swapValues(a[j], a[j - 1]);
}

int main() {
    int x[5] = {10, 50, 30, 40, 20};
    float y[5] = {1.1, 5.5, 3.2, 4.6, 7.2};

    bubblesort(x, 5);
```

```
    bubblesort(y, 5);

    cout << "Sorted x-array: ";
    for (int i = 0; i < 5; i++)
        cout << x[i] << " ";
    cout << endl;

    cout << "Sorted y-array: ";
    for (int j = 0; j < 5; j++)
        cout << y[j] << " ";
    cout << endl;

    return 0;
}
```

✅ **6. Non-type Template Argument Example**

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

template <class T, int size>
class Array {
    T a[size];
public:
    void setValue(int index, T value) {
        if (index >= 0 && index < size)
            a[index] = value;
    }

    T getValue(int index) {
        if (index >= 0 && index < size)
```

```
            return a[index];
        return T(); // default value
    }
};

int main() {
    Array<int, 5> a1;
    a1.setValue(0, 10);
    cout << "First value: " << a1.getValue(0) << endl;

    Array<char, 3> a2;
    a2.setValue(1, 'Z');
    cout << "Second value: " << a2.getValue(1) << endl;

    return 0;
}
```

✅ **7. Class Template with Single Type Parameter**

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

template <class T>
class XYZ {
    T a, b;
public:
    XYZ(T x, T y) {
        a = x;
        b = y;
    }
```

```
    void show() {
        cout << "The addition of " << a << " and " << b << " is " << add() << endl;
    }

    T add() {
        return a + b;
    }
};

int main() {
    XYZ<int> addInt(8, 6);
    XYZ<float> addFloat(3.5, 2.6);
    XYZ<double> addDouble(2.156, 5.234);

    addInt.show();
    addFloat.show();
    addDouble.show();

    return 0;
}
```

✅ 8. Class Template with Multiple Type Parameters

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

template <class T1, class T2>
class Test {
    T1 a;
    T2 b;
public:
```

```
    Test(T1 x, T2 y) {
        a = x;
        b = y;
    }

    void show() {
        cout << a << " and " << b << "\n";
    }
};

int main() {
    Test<float, int> test1(1.23, 123);
    Test<int, char> test2(100, 'W');

    test1.show();
    test2.show();

    return 0;
}
```

MOD 8

🧩 STL Components

✅ 1. Containers

Used to **store and organize data**.

Types:

- **Sequence Containers** – vector, list, deque
- **Associative Containers** – set, map, multiset, multimap
- **Derived Containers (Adapters)** – stack, queue, priority_queue

✅ 2. Algorithms

Procedures that operate on containers using iterators (e.g., sort(), find()).

✅ 3. Iterators

Bridge between containers and algorithms. They **behave like pointers** and allow navigation.

Sequence Containers – Code Examples

✳️ Vector

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

int main() {
    vector<int> v = {1, 2, 3, 4};

    v.push_back(5);
    cout << "Vector elements: ";
    for (int x : v)
        cout << x << " ";
    return 0;
}
```

✳️ List

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

int main() {
    list<string> fruits = {"apple", "banana", "cherry"};
    fruits.push_front("mango");

    for (auto it = fruits.begin(); it != fruits.end(); ++it)
        cout << *it << " ";
    return 0;
```

```
}
```

✳️ **Deque**

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

int main() {
    deque<int> dq = {10, 20, 30};
    dq.push_front(5);
    dq.push_back(40);

    for (int x : dq)
        cout << x << " ";
    return 0;
}
```

🔹 **Associative Containers – Code Examples**

✳️ **Set**

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

int main() {
    set<int> s = {1, 2, 2, 3, 4};
    s.insert(5);

    for (int x : s)
        cout << x << " ";  // Unique and sorted
```

```
    return 0;
}
```

✳ **Multiset**

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

int main() {
    multiset<int> ms = {1, 2, 2, 3};
    ms.insert(2);

    for (int x : ms)
        cout << x << " ";  // Allows duplicates
    return 0;
}
```

✳ **Map**

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

int main() {
    map<int, string> m;
    m[1] = "One";
    m[2] = "Two";

    for (auto pair : m)
        cout << pair.first << " => " << pair.second << endl;
    return 0;
}
```

✳ **Multimap**

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

int main() {
    multimap<int, string> mm;
    mm.insert({1, "apple"});
    mm.insert({1, "banana"});

    for (auto &pair : mm)
        cout << pair.first << " => " << pair.second << endl;
    return 0;
}
```

🔹 **Derived Containers (Adapters)**

✳️ **Stack**

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

int main() {
    stack<int> s;
    s.push(10);
    s.push(20);
    s.pop();

    cout << "Top: " << s.top();
    return 0;
}
```

✳️ **Queue**

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

int main() {
    queue<string> q;
    q.push("first");
    q.push("second");
    q.pop();

    cout << "Front: " << q.front();
    return 0;
}
```

✳️ **Priority Queue**

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

int main() {
    priority_queue<int> pq;
    pq.push(30);
    pq.push(10);
    pq.push(20);

    cout << "Highest priority: " << pq.top();
    return 0;
}
```

🔹 **Iterators Example**

```
#include <iostream>
```

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

int main() {
    vector<int> v = {10, 20, 30};
    vector<int>::iterator it;

    for (it = v.begin(); it != v.end(); ++it)
        cout << *it << " ";
    return 0;
}
```

🔹 **Algorithms in STL**

✳️ **Sort, Reverse, Find**

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

int main() {
    vector<int> v = {4, 2, 5, 3, 1};

    sort(v.begin(), v.end());
    cout << "Sorted: ";
    for (int x : v) cout << x << " ";

    reverse(v.begin(), v.end());
    cout << "\nReversed: ";
    for (int x : v) cout << x << " ";
```

```
    if (find(v.begin(), v.end(), 3) != v.end())
        cout << "\n3 found!";
    else
        cout << "\n3 not found!";

    return 0;
}
```

🎯 **Summary**

Component	Function	Header
Vector	Dynamic array	<vector>
List	Doubly linked list	<list>
Set	Unique sorted values	<set>
Map	Key-value pair	<map>
Stack	LIFO stack	<stack>
Queue	FIFO queue	<queue>
Priority Queue	Max-heap based priority container	<queue>
Algorithm	Generic algorithms	<algorithm>
Iterator	Traverse through containers	<iterator>

STL - VECTOR

Vector

- **Widely used container**
- Stores elements in **contiguous memory**
- Enables **direct access** using subscript []
- Can dynamically change size and allocate memory at runtime

Definition

- A **vector** is a sequence container class that implements a **dynamic array**.
- It stores elements in **contiguous memory locations** and allocates memory as needed at runtime.

Features

- Supports **random access iterators**
- Constructors:
  - vector<int> v1; → **size 0**
  - vector<double> v2(10); → **size 10** (double type)
  - vector<int> v3(v4); → **copy** from another vector
  - vector<int> v(5, 2); → **size 5** with initial value 2
- Supports member functions and STL algorithms

Vector Functions

Function	Description
at()	Reference to element at specified index
back()	Reference to the last element
front()	Reference to the first element
swap()	Exchanges elements between two vectors
push_back()	Adds element to the end
pop_back()	Removes the last element
empty()	Checks if the vector is empty
insert()	Inserts element at a specified position
erase()	Deletes specified element
resize()	Changes the size of the vector
clear()	Removes all elements from the vector
size()	Returns the number of elements in the vector
capacity()	Returns the current capacity of the vector

Function	Description
assign()	Assigns new values to the vector
operator=()	Assigns new values to the vector container
operator[]()	Accesses a specified element
end()	Past-the-last element
emplace()	Inserts before the specified position
emplace_back()	Inserts at the end
rend()	Reverse end (reverse iterator)
rbegin()	Reverse begin (reverse iterator)
begin()	Points to the first element
max_size()	Maximum possible size
cend()	Constant past-the-last element
cbegin()	Constant first element
crbegin()	Constant reverse begin
crend()	Constant reverse end
data()	Pointer to array
shrink_to_fit()	Reduces capacity to match the current size of the vector

---

Code Example 1: Basic Operations

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

void display(vector<int> &v) {
   for(int i = 0; i < v.size(); i++) {
      cout << v[i] << " ";
   }
```

```
    cout << "\n";
}

int main() {
    vector<int> vec;
    int i;

    cout << "vector size = " << vec.size() << endl;

    for(i = 0; i < 5; i++) {
        vec.push_back(i);
    }

    cout << "extended vector size = " << vec.size() << endl;
    cout << "Current contents = ";
    display(vec);

    vector<int>::iterator myv = vec.begin();
    while(myv != vec.end()) {
        cout << "value of v = " << *myv << endl;
        myv++;
    }

    vec.insert(vec.begin(), 3, 9);
    cout << "After Insertion: ";
    display(vec);

    vec.erase(vec.begin()+3, vec.begin()+5);
    cout << "After Deletion: ";
    display(vec);
```

```
    return 0;
}
```

Output

```
vector size = 0
extended vector size = 5
Current contents = 0 1 2 3 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
After Insertion: 9 9 9 0 1 2 3 4
After Deletion: 9 9 9 2 3 4
```

Code Example 2: Sort

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

int main() {
    vector<int> test;
    test.push_back(1);
    test.push_back(8);
    test.push_back(4);

    sort(test.begin(), test.end());
```

```
    for (int i = 0; i < test.size(); i++) {
        cout << test[i] << endl;
    }

    return 0;
}
```

Output

1

4

8

Code Example 3: Search

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

int main() {
    vector<int> test;
    test.push_back(1);
    test.push_back(8);
    test.push_back(4);

    int k = 8;
    vector<int>::iterator itr = find(test.begin(), test.end(), k);

    if (itr == test.end()) {
        cout << "element not found";
    } else {
```

```
        cout << "element found";
    }

    return 0;
}
```

Output

element found

Code Example 4: Count

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

int main() {
    vector<int> test;
    test.push_back(1);
    test.push_back(8);
    test.push_back(4);
    test.push_back(1);

    int ct = count(test.begin(), test.end(), 1);
    cout << ct;

    return 0;
}
```

Output

2