1. What is the difference between malloc() vs new and free() vs delete?

- `malloc()` **vs.** `new`**:**
  - **Type:**
    - `malloc()`: malloc() is a function. Declared in `<stdlib.h>`.
    - `new`: new is an operator. No separate header file needed.
  - **Initialization:**
    - `malloc()`: Memory allocation only. Doesn't call constructors for objects.
    - `new`: Memory allocation and initialization. Calls constructors for objects.
  - **Usage with Arrays:**
    - `malloc()`: No special handling for arrays.
    - `new`: Supports array allocation. `new[]` is used for arrays, which can later be deallocated with `delete[]`.
  - **Return Type:**
    - `malloc()`: Returns a `void*` pointer.
    - `new`: Returns a pointer to the type of object being allocated.
  - **Type Safety:**
    - `malloc()`: Not type-safe. Requires explicit casting.
    - `new`: Type-safe. No need for explicit casting.
  - **Overloading:**
    - `new`: Supports overloading to customize memory allocation behavior.
    - `malloc()`: malloc() is not meant to be overloaded.
  - **Usage in C++:**
    - `malloc()`: Can be used in C++ but not recommended due to lack of support for constructors.
    - `new`: Preferred in C++ for dynamic memory allocation because it supports constructors.

- `free()` **vs.** `delete`**:**
  - **Type:**
    - `free()`: free() is a function. Declared in `<stdlib.h>`.
    - `delete`: delete is an operator. No separate header file needed.

- **Type of Memory:**
  - `free()`: Used to deallocate memory allocated with `malloc()` or `calloc()`.
  - `delete`: Used to deallocate memory allocated with `new`.
- **De-Initialization:**
  - `free()`: Only deallocates memory. Doesn't call destructors for objects.
  - `delete`: Deallocates memory and calls destructors for objects.
- **Usage with Arrays:**
  - `free()`: No special handling for arrays.
  - `delete`: Used with `delete[]` to deallocate memory allocated for arrays.
- **Overloading:**
  - `delete`: Supports overloading to customize memory deallocation behavior.
  - `free()`: free() is not meant to be overloaded.
- **Usage in C++:**
  - `free()`: Not used in C++. Deallocating memory allocated with `malloc()` or `calloc()` using `free()` in C++ can lead to undefined behavior if the object has non-trivial constructors or destructors.
  - `delete`: Preferred in C++ for deallocating memory allocated with `new` because it properly calls destructors for objects.

2. Write a code to allocate and deallocate memory for multi-dimensional array using new and delete?

```CPP
#include <iostream>

int** allocateArray(int rows, int cols) {
    int** arr = new int*[rows];  // Allocate memory for array of row pointers
    for (int i = 0; i < rows; ++i) {
        arr[i] = new int[cols];  // Allocate memory for each row
    }
    return arr;
}

void deallocateArray(int** arr, int rows) {
```

```
    for (int i = 0; i < rows; ++i) {
        delete[] arr[i];  // Deallocate memory for each row
    }
    delete[] arr;  // Deallocate memory for array of row pointers
}

int main() {
    int rows = 3;
    int cols = 4;
    int** arr = allocateArray(rows, cols);

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            arr[i][j] = i * cols + j;  // Example initialization
        }
    }

    std::cout << "Array elements:" << std::endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << arr[i][j] << " ";
        }
        std::cout << std::endl;
    }

    deallocateArray(arr, rows);
    return 0;
}
```

3. What is "this" pointer? Is it available for static, virtual, const and friend functions?

* The "this" pointer in C++ is a pointer that holds the address of the current object. It is a keyword that can be used within the member functions of a class to refer to the object on which the member function is called.
* The "this" pointer is implicitly available within the scope of non-static member functions of a class.
* It is commonly used to access member variables and member functions of the current object. For example:

```cpp
class Example {
private:
    int x;
public:
    void setX(int x) {
        this->x = x; // "this" pointer is used to differentiate between member variable and function parameter
    }
};
```

```cpp
int main() {
    Example ex;
    ex.setX(3);
}
```

* **Static Member Functions:** "this" pointer is not available within static member functions because static member functions do not operate on specific class instances/objects.
* **Virtual Member Functions:** "this" pointer is available in virtual member functions. When a virtual function is called through a pointer or reference to a base class, the "this" pointer points to the actual object type.
* **Const Member Functions:** "this" pointer is available in const member functions. However, when a member function is declared as const, the type of "this" pointer is considered as a pointer to a constant object, so you cannot modify member variables inside const member functions.
* **Friend Functions:** "this" pointer is not available within global friend functions because friend functions are not member functions of the class, even though they have access to the class's private and protected members. However, if friend function is member function of another class, it will have "this" pointer that will point to object of that class.

4. Why we can not declare static member function constant or virtual?

```
* In C++, static member functions cannot be declared as const or virtual.
* **Const Member Functions:**
    - Const member functions are functions that inteded not to modify the state of the object on which they are called. They are declared using the `const` keyword at the end of the function declaration.
    - Static member functions do not operate on specific class instances. They are not associated with any particular object, so it doesn't make sense to declare them as const because there's no object state to be modified or not modified.
* **Virtual Member Functions:**
    - Virtual member functions are used in polymorphism to achieve dynamic binding, allowing a derived class's implementation to be called through a base class pointer or reference.
    - Static member functions do not participate in polymorphism because they are resolved at compile time based on the class name, not at runtime based on the object's type.
    - Additionally, virtual functions are associated with specific objects and their behavior can vary depending on the object's type, which is not applicable to static member functions.
```

5. What is the need to write user defined destructor? When it should be declared as "virtual"?

```
* If base class pointer points, to dynamically allocated derived class object and derived class destructor is releasing any resource/memory, then base class destructor should be declared as virtual.
* The virtual destructor will ensure that first derived class destructor is called, and then the base class destructor is called. Finally, it will release the memory of dynamically allocated derived class object too. This will release all resources from base as well as derived classes.
* If destructor is not declared virtual, it will call base class destructor and release the memory of dynamically allocated derived class object. However, derived class destructor is not called; so resource/memory deallocated by derived class destructor will leak.

```cpp
#include <iostream>
class Base {
```
```

```
    int *bptr;
public:
    Base() {
        bptr = new int[3];
        std::cout << "Base constructor called" << std::endl;
    }
    virtual ~Base() {
        delete[] bptr;
        std::cout << "Base destructor called" << std::endl;
    }
};
class Derived : public Base {
    int *dptr;
public:
    Derived() {
        dptr = new int[3];
        std::cout << "Derived constructor called" << std::endl;
    }
    ~Derived() {
        delete[] dptr;
        std::cout << "Derived destructor called" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();  // Using base class pointer to point to derived class object
    delete ptr;  // Deleting object through base class pointer
    return 0;
}
```

6. Why constructor cannot be declared as virtual?

```
* In C++, constructors cannot be declared as virtual because the virtual mechanism relies on the existence of a valid object.
* When an object is being constructed, its virtual table (vtable) is not yet fully constructed or initialized. Consequently, the mechanism required to resolve virtual functions is not available during the construction process.
* Also, virtual functions are designed to be called depending on type of object via pointer/reference. However, constructor is never called on pointer. It is called when object of the class is created (class name is given in code).
```

7. Explain dynamic_cast operator. When it is required? Explain with example.

```
* The `dynamic_cast` operator in C++ is used for performing safe downcasting (casting from a base class pointer or reference to a derived class pointer or reference). It allows you to check whether the object being cast is of a specific type at runtime, and if so, perform the cast safely.
* The `dynamic_cast` operator can only be used with pointers or references to polymorphic classes. A polymorphic class is a class that has at least one virtual function declared within it.
* Syntax:
    ```cpp
    dynamic_cast<DerivedType*>(BaseType*);
    ```
* Example:
    ```cpp
    #include <iostream>
    class Base {
        int b;
    public:
        virtual void fun1() { }
        virtual void fun2() { }
        virtual ~Base() {}
    };
    class Derived : public Base {
        int d;
```
```

```
    public:
        virtual void fun1() { }
        void derivedMethod() {
            std::cout << "Derived method called" << std::endl;
        }
    };
    int main() {
        Base* basePtr = new Derived; // Upcasting: Derived object assigned to Base pointer
        //basePtr->derivedMethod(); // compiler error: due to Object slicing

        // Attempting to downcast using dynamic_cast
        Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
        // Check if downcast was successful
        if (derivedPtr != NULL) {
            // Downcast successful, call derived class method
            derivedPtr->derivedMethod();
        } else {
            // Downcast failed, handle accordingly
            std::cout << "Dynamic cast failed" << std::endl;
        }

        delete basePtr; // Clean up allocated memory

        return 0;
    }
    ```
* If dynamic_cast fails for the pointer, it returns NULL; However, if dynamic_cast fails for the references, it throws bad_cast exception.
```

8. How virtual function affects on size of object? How it is affected in single and multiple inheritance?
```

* The presence of virtual functions in a class can affect the size of the object, primarily due to the addition of a virtual function table pointer (`vptr`).
* **Single Inheritance:**
    * In single inheritance, where a class has only one direct base class, the addition of virtual functions typically increases the size of the object by the size of a pointer (usually 4 or 8 bytes depending on the architecture) to the virtual function table (`vtable`).
    * Example:
        ```cpp
        class Base {
        public:
            virtual void foo() {}
        };
        class Derived : public Base {
        public:
            virtual void bar() {}
        };
        int main() {
            std::cout << "Size of Base: " << sizeof(Base) << std::endl;       // Output: Size of Base: 8
            std::cout << "Size of Derived: " << sizeof(Derived) << std::endl; // Output: Size of Derived: 8 i.e. vptr of Base is inherited to the Derived.
            return 0;
        }
        ```
* **Multiple Inheritance:**
    * In multiple inheritance, where a class has multiple direct base classes, each base class will have its own virtual function table pointer (`vptr`) and all these vptr will be inherited to the Derived class.
    * Example:
        ```cpp
        class Base1 {
        public:
            virtual void foo() {}
        };
        ```

```
class Base2 {
public:
    virtual void bar() {}
};

class Derived : public Base1, public Base2 {
public:
    virtual void baz() {}
};

int main() {
    std::cout << "Size of Base1: " << sizeof(Base1) << std::endl;     // Output: Size of Base1: 8
    std::cout << "Size of Base2: " << sizeof(Base2) << std::endl;     // Output: Size of Base2: 8
    std::cout << "Size of Derived: " << sizeof(Derived) << std::endl; // Output: Size of Derived: 16
    return 0;
}
```

9. What is the need to overload index operator? Explain with example?

```
* Overloading the index operator (`[]`) in C++ allows objects of a class to be treated like arrays, providing a way to
access elements of the object using array syntax. This can make the code more intuitive and easier to read, especially when
dealing with custom data structures or classes that represent collections of elements.
* Example:
    ```cpp
    #include <iostream>

    class MyArray {
    private:
        int data[5]; // Internal array to store data
```
```

```
    public:
        // Overloading the index operator
        int& operator[](int index) {
            // can check array bounds (index) here
            return data[index];
        }
    };

    int main() {
        MyArray arr;
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        std::cout << "arr[0]: " << arr[0] << std::endl;
        std::cout << "arr[1]: " << arr[1] << std::endl;
        std::cout << "arr[2]: " << arr[2] << std::endl;
        return 0;
    }
    ```
* Since operator return type is by reference, it can be used on left-side or right-side of the assignment operator.
    ```cpp
    y = arr[i];
        // y = arr.operator[](i);
    arr[i] = x;
        // arr.operator[](i) = x;
    ```
```

## 10. What is the difference between pointer and reference?

Pointers and references in C++ are both used to indirectly access objects, but they have some key differences: * **Syntax:** - **Pointer:** Pointers are declared using the `*` symbol. They store the memory address of another object. `cpp int x = 10; int* ptr = &x;` - **Reference:** References are declared using the `&` symbol. They are aliases for other objects and must be initialized when declared. `cpp int x = 10; int& ref = x;` * **Null Assignment:** - **Pointer:** Pointers can be assigned
```

the value `NULL` to indicate that they are not pointing to any valid memory location. - **Reference:** References cannot be `NULL`. They must be initialized to refer to a valid object upon declaration. * **Reassignment:** - **Pointer:** Pointers can be reassigned to point to different objects (if not const pointer). - **Reference:** References cannot be reassigned to refer to a different object after initialization. They are fixed aliases. * **Dereferencing:** - **Pointer:** Pointers are dereferenced using the `*` operator to access the value stored at the memory address they point to. `cpp int value = *ptr;` - **Reference:** References are automatically dereferenced when used. They behave syntactically like the object they refer to. `cpp int value = ref; // Cannot explicitly dereference -- Auto dereferenced` * **Memory Management:** - **Pointer:** Pointers require manual memory management, such as allocation and deallocation using `new` and `delete`. - **Reference:** References do not involve memory management. They are just aliases for existing objects. * **Arithmetic Operations:** - **Pointer:** Pointers support arithmetic operations such as incrementing and decrementing to move to different memory locations. - **Reference:** References do not support arithmetic operations. They are fixed aliases and cannot be manipulated like pointers. * **Function Parameters:** - **Pointer:** Pointers are commonly used to pass parameters to functions by address, allowing the function to modify the original object. - **Reference:** References can also be used to pass parameters by reference. They provide a cleaner syntax compared to pointers. * However, internally references are pointer (that are auto-dereferenced) and size of reference is always 8 bytes. ```cpp char z = 'A'; // global var

```
class Test {
    char &x;
public:
    Test() : x(z) { ... }
};

int main() {
    cout << sizeof(Test) << endl; // 8 bytes
    return 0;
}
```

11. What is diamond problem? How to solve it?

```
* The diamond problem is a common issue that arises in multiple inheritance when a class inherits from two or more classes
that have a common ancestor. This situation creates ambiguity in the inheritance hierarchy, as there can be multiple paths
```

to access the same member from the common ancestor.
* Consider the following inheritance hierarchy:
    ```
         A
        / \
       B   C
        \ /
         D
    ```
* If class `A` has a member function `foo()`, then it is inherited to class `D` from two paths i.e. class `B` and `C`.
    ```cpp
    class A {
        public:
            void foo() { std::cout << "A::foo()" << std::endl; }
        };

        class B : public A {
        public:
        };

        class C : public A {
        public:
        };

        class D : public B, public C {
        };

        int main() {
            D d;
            d.foo();  // Call is ambiguous
            return 0;
        }
    ```

* Calling foo() on D class object leads to ambiguity error.
* There are two ways to solve the diamond problem in C++:
1. **Virtual Inheritance:**
    - Virtual inheritance is a feature in C++ that allows a class to inherit from a common base class only once, regardless of how many times it appears in the inheritance hierarchy.
    - By using virtual inheritance, ambiguity is avoided, and there is only one instance of the common base class shared among all derived classes.
    * Example:
    ```cpp
    class A {
    public:
        void foo() { std::cout << "A::foo()" << std::endl; }
    };

    class B : virtual public A {
    };

    class C : virtual public A {
    };

    class D : public B, public C {
    };

    int main() {
        D d;
        d.foo();  // Output: A::foo()
        return 0;
    }
    ```

2. **Scope Resolution Operator:**
    - You can use the scope resolution operator (`::`) to explicitly specify which version of the member function should be called.
    - This approach is less preferred as it does not resolve the underlying ambiguity and may lead to code maintenance

````
issues.
    - Example:
    ```cpp
    int main() {
        D d;
        d.B::foo();  // Output: A::foo()
        return 0;
    }
    ```
````

12. What is shallow copy and deep copy? How it is implemented in C++? Explain with example.

```
* Shallow copy and deep copy are two ways of copying objects in C++.

1. **Shallow Copy:**
    - Shallow copy creates a new object and copies all the data members of the original object to the new object.
    - If the data members of the original object include pointers, shallow copy only copies the addresses of the pointers,
not the data pointed to.
    - As a result, both the original object and the copied object share the same dynamically allocated memory.
    - Changes made to the data pointed to by one object affect the other object.

2. **Deep Copy:**
    - Deep copy creates a new object and copies all the data members of the original object to the new object.
    - If the data members of the original object include pointers, deep copy allocates new memory for the pointers and
copies the data pointed to by the pointers to the new memory locations.
    - As a result, each object has its own dynamically allocated memory, and changes made to one object do not affect the
other object.

* In C++, shallow copy and deep copy can be implemented manually by overloading the copy constructor and the assignment
operator. The default copy constructor and assignment operator does shallow copy,
* Here's an example demonstrating shallow copy and deep copy:
```

```cpp
#include <iostream>
class DynamicArray {
private:
    int* data;
    int size;
public:
    DynamicArray(int sz) {
        size = sz;
        data = new int[size];
        for (int i = 0; i < size; ++i)
            data[i] = i + 1;
    }
    DynamicArray(const DynamicArray& other) {
        size = other.size;
        data = new int[size];
        for (int i = 0; i < size; ++i)
            data[i] = other.data[i];
    }
    DynamicArray& operator=(const DynamicArray& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            for (int i = 0; i < size; ++i)
                data[i] = other.data[i];
        }
        return *this;
    }

    ~DynamicArray() {
        delete[] data;
    }
```

```
    void display() {
        for (int i = 0; i < size; ++i) {
            std::cout << data[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    DynamicArray arr1(5);
    DynamicArray arr2 = arr1; // copy constructor
    DynamicArray arr3(arr1); // copy constructor

    DynamicArray arr4(4);
    arr3 = arr1;  // assignment operator

    std::cout << "Original array: ";
    arr1.display();
    std::cout << "Copied array: ";
    arr2.display();
    std::cout << "Assigned array: ";
    arr4.display();
    return 0;
}
```

13. What is conversion function? Which are conversion functions in C++?

```
* A conversion function in C++ is a member function of a class that is used to convert an object of that class to another
type.
```

* It allows objects of a user-defined type to be implicitly or explicitly converted to another type.
* There are two types of conversion functions in C++ i.e. conversion operator and single argument constructor.

```cpp
class Rational {
public:
    Rational(int num, int den) {
        numerator = num;
        denominator = den;
    }
    Rational(int num) {
        numerator = num;
        denominator = 1;
    }
    operator double() const {
        return (double)numerator / denominator;
    }
private:
    int numerator;
    int denominator;
};

int main() {
    Rational r(3, 2);
    double d = r;  // Implicit conversion to double
        // d = r.operator double();
    int x = 5;
    Rational s;
    s = x; // Implicit conversion from int to Rational
        // x --> anonymous Rational object
        // s = anonymous object (assignment)
        // anonymous object destroyed
    return 0;
}
```

````
* Note that, both conversion functions are called automatically (in above example). To avoid this implicit conversion use
`explicit` keyword before these conversion functions. Now they will be invoked only when casting is done.
    ```cpp
    int main() {
        Rational r(3, 2);
        double d = static_cast<double>(r);  // Explicit conversion to double
            // d = r.operator double();
        int x = 5;
        Rational s;
        s = static_cast<Rational>(x); // Explicit conversion from int to Rational
            // x --> anonymous Rational object
            // s = anonymous object (assignment)
            // anonymous object destroyed
        return 0;
    }
    ```
````

14. What is smart pointer? Which are smart pointers in C++?

````
* Smart pointers are objects that behave like pointers but provide automatic memory management.
* They help prevent memory leaks and manage the lifetime of dynamically allocated objects.
* Smart pointers are part of the C++ Standard Library and are implemented as template classes.
* The main smart pointers in C++ are:

1. **std::unique_ptr:**
- `std::unique_ptr` is a smart pointer that owns a dynamically allocated object and ensures that the object is deleted when
the `unique_ptr` goes out of scope or is reset.
- It is unique in that it cannot be copied or shared. It can only be moved.
- It is lightweight and efficient because it does not incur the overhead of reference counting.
    ```cpp
    #include <memory>
````
````

```cpp
#include <iostream>

int main() {
    std::unique_ptr<int> ptr(new int(42));
    std::cout << *ptr << std::endl;
    return 0;  // ptr automatically deletes the allocated memory when it goes out of scope
}
```

2. **std::shared_ptr:**
- `std::shared_ptr` is a smart pointer that manages the ownership of a dynamically allocated object using reference counting.
- It allows multiple `shared_ptr` objects to share ownership of the same dynamically allocated object.
- The object is deleted when the last `shared_ptr` pointing to it is destroyed or reset.

```cpp
#include <memory>
#include <iostream>

int main() {
    std::shared_ptr<int> ptr1(new int(42));
    std::shared_ptr<int> ptr2 = ptr1;
    std::cout << *ptr1 << " " << *ptr2 << std::endl;
    return 0;  // Both ptr1 and ptr2 share ownership of the allocated memory
}
```

3. **std::weak_ptr:**
- `std::weak_ptr` is a smart pointer that provides a non-owning reference to an object managed by `std::shared_ptr`.
- It does not participate in reference counting and does not keep the object alive.
- It is used to break cyclic dependencies between `std::shared_ptr` objects.

```cpp
#include <memory>
#include <iostream>
```

```
int main() {
    std::shared_ptr<int> ptr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = ptr;
    if (auto sharedPtr = weakPtr.lock()) {
        std::cout << *sharedPtr << std::endl;  // Access the object if it still exists
    }
    return 0;  // weakPtr does not affect the lifetime of the allocated memory
}
```

15. What is STL? Explain different components in STL with examples?

* STL stands for Standard Template Library. It is a powerful set of C++ template classes and functions that provide reusable, efficient, and flexible algorithms and data structures. The STL is part of the C++ Standard Library and is widely used for various programming tasks, such as data manipulation, sorting, searching, and more.

* The main components of the STL include:
    * Containers
    * Iterators
    * Algorithms

* **Containers:**
    * Containers are data structures that store and manage collections of objects. There are several types of containers in the STL, each with different characteristics and use cases. Some common containers include:
        - **vector:** Dynamic array that can resize itself automatically.
        - **list:** Doubly linked list that allows for efficient insertion and deletion of elements.
        - **deque:** Double-ended queue that supports insertion and deletion at both ends.
        - **stack:** LIFO (Last In, First Out) data structure.
        - **queue:** FIFO (First In, First Out) data structure.
        - **set:** Collection of unique, sorted elements.
        - **map:** Collection of key-value pairs, where each key is unique.

    * Example:
    ```cpp
    #include <iostream>
    #include <vector>
    #include <set>

    int main() {
        // Vector example
        std::vector<int> vec = {1, 2, 3, 4, 5};
        vec.push_back(6);
        std::cout << "Vector size: " << vec.size() << std::endl;

        // Set example
        std::set<int> mySet = {3, 1, 4, 1, 5, 9, 2, 6, 5};
        std::cout << "Set size: " << mySet.size() << std::endl;

        return 0;
    }
    ```

2. **Iterators:**
Iterators are objects that provide a way to iterate over the elements of a container in a sequential manner. They act as pointers to elements within the container and support various operations like dereferencing, incrementing, and decrementing.

Example:
```cpp
#include <iostream>
#include <vector>

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

    // Using iterators to iterate over the vector
```

```
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

3. **Algorithms:**
Algorithms are functions that operate on containers and perform various operations such as sorting, searching, transforming, and modifying elements. STL provides a wide range of algorithms that can be used with different types of containers.

Example:
```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {5, 2, 7, 1, 9, 3};

    // Sorting the vector using std::sort algorithm
    std::sort(vec.begin(), vec.end());

    // Displaying the sorted vector
    for (auto elem : vec) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

These are some of the main components of the STL. By leveraging the containers, iterators, and algorithms provided by the STL, C++ programmers can write efficient and expressive code for a wide range of tasks.

16. What is difference between set, vector and map? How to traverse them? Explain with code.

The main differences between `set`, `vector`, and `map` in C++ STL lie in their characteristics, usage, and underlying data structures:

1. **Set:**
- A set is an associative container that stores unique elements in sorted order.
- It does not allow duplicate elements.
- Implemented using a self-balancing binary search tree (usually Red-Black Tree).
- Provides efficient insertion, deletion, and search operations.

* **Traversing a Set:**
    - Sets are typically traversed using iterators.
    - Since sets are sorted, elements are traversed in sorted order.

    ```cpp
    #include <iostream>
    #include <set>

    int main() {
        std::set<int> mySet = {3, 1, 4, 1, 5};

        // Traversing set using iterators
        for (auto it = mySet.begin(); it != mySet.end(); ++it) {
            std::cout << *it << " ";
        }
        std::cout << std::endl;
    ```

```
        return 0;
    }
    ```

2. **Vector:**
- A vector is a dynamic array that can resize itself automatically.
- Elements are stored in contiguous memory locations, allowing efficient random access.
- Allows duplicate elements.
- Provides efficient insertion and deletion at the end of the vector, but less efficient for insertion and deletion in the middle.

* **Traversing a Vector:**
    - Vectors can be traversed using iterators or index-based loops.
    - Iterators provide a flexible way to traverse vectors.

    ```cpp
    #include <iostream>
    #include <vector>

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

        // Traversing vector using iterators
        for (auto it = vec.begin(); it != vec.end(); ++it) {
            std::cout << *it << " ";
        }
        std::cout << std::endl;

        // Traversing vector using index-based loop
        for (size_t i = 0; i < vec.size(); ++i) {
            std::cout << vec[i] << " ";
        }
        std::cout << std::endl;
```

```
        return 0;
    }
    ```

3. **Map:**
- A map is an associative container that stores key-value pairs.
- Each key is unique, and the elements are sorted based on the keys.
- Implemented using a self-balancing binary search tree or hash table.
- Provides efficient insertion, deletion, and search operations based on keys.
* **Traversing a Map:**
    - Maps are typically traversed using iterators, which provide access to both keys and values.

    ```cpp
    #include <iostream>
    #include <map>

    int main() {
        std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}, {3, "three"}}; // C++' 17

        // Traversing map using iterators
        for (auto it = myMap.begin(); it != myMap.end(); ++it) {
            std::cout << it->first << ": " << it->second << std::endl;
        }

        return 0;
    }
    ```