Welcome

Object Oriented Programming (UTA018)

(Session: 2026-2027 ODD)

Instructor: Dr. Suresh Raikwar

Department of Computer Science and Engineering,

Thapar Institute of Engineering and Technology, Patiala, Punjab, India

Important Instruction(s):

- This material is intended for guidance and conceptual understanding. It does not guarantee that examination questions will be drawn directly from it.
- Merely studying this material is not sufficient for achieving good performance.
- Sessional Tests, MST, and EST will assess your understanding of Object-Oriented Programming concepts through **logical reasoning, analytical thinking, and problem-solving**.
- Students are strongly encouraged to attend regular classes and practice programming problems from multiple sources to develop a deeper conceptual understanding and improve problem-solving skills.

UTA 018: OOPs
Exception Handling



What is Exception Handling?

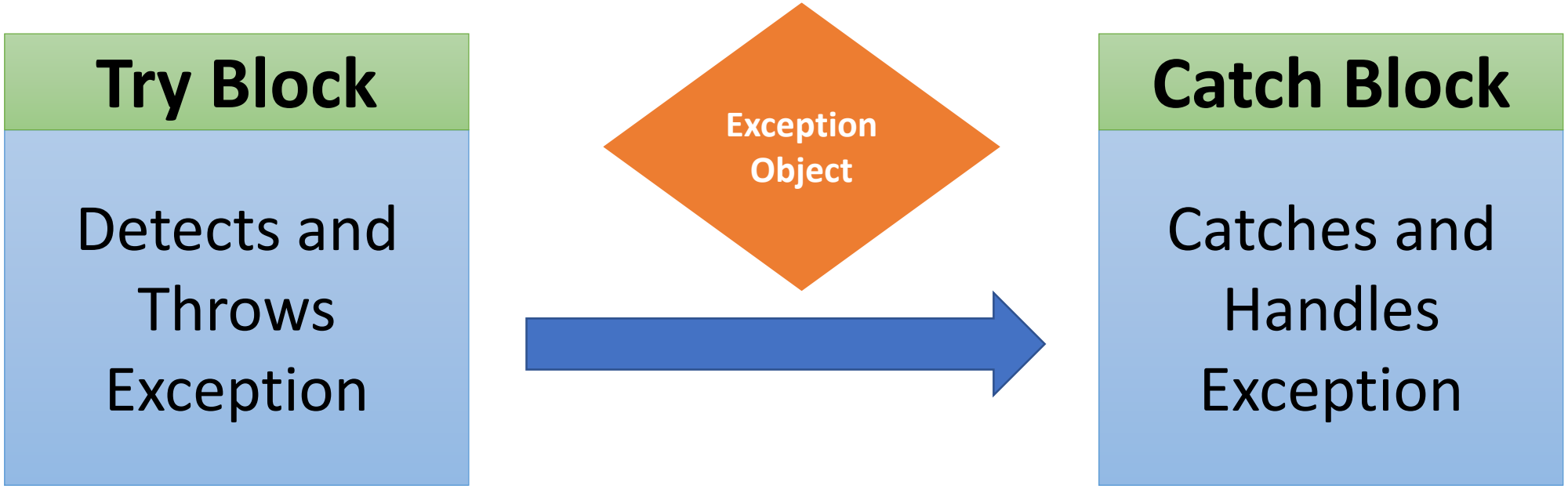
Exception handling is a mechanism in C++ to handle runtime errors or "exceptions" that may occur during the execution of a program. These exceptions could include invalid user input, divide-by-zero errors, memory allocation failures, file I/O errors, etc.

Exception Handling Motivation

The **motivation** behind exception handling is to:

- **Separate error-handling code from the main logic** of the program.
- **Increase robustness**: Handle errors gracefully without crashing the program.
- **Provide a way to recover** from unforeseen runtime conditions.
- Without proper error handling, programs can terminate abruptly, leading to poor user experiences and system instability.
- Exception is a run-time error, such as divide by zero, array out of bounds, file not found etc.
- It helps to continue the normal flow of the application even in the case of runtime errors
- It makes the code simpler, cleaner, and less likely to miss errors

Exception Handling Mechanism



Basic Syntax of Exception Handling

C++ provides a structured way to handle exceptions using three main keywords:

- **try**: Block of code that may throw an exception.
- **throw**: Used to throw an exception when an error occurs.
- **catch**: Block of code that handles the exception.

```
try {
    // Code that may cause an exception
    if (/* some error condition */) {
        throw exception;  // Throw an exception
    }
}
catch (type_of_exception) {
    // Code to handle the exception
}
```

```
try{
fun1();
fun2();
fun3();
} catch(error){
// … do something if error happens in any of the functions
}
```

// if any of the fun() is a constructor then you have to use *if-else* in the constructor to handle the error since it cannot return anything, which is not neat practice.

Exception thrown and caught

- Exception is thrown at runtime.
- If an exception is not handled (caught), it prints exception message and terminates the program.
- Programmer's job is to handle possible exceptions using *try*, *throw* and *catch* to avoid abnormal program termination

Catching Multiple Exceptions

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

int main() {
    try {
        int choice;
        cout << "Enter choice (1 for int, 2 for float): ";
        cin >> choice;
        if (choice == 1)
            throw 100;
        else if (choice == 2)
            throw 2.5f;
        else
            throw "Invalid choice!";
    }
    catch (int e) {
        cout << "Caught an integer: " << e << endl;
    }
    catch (float e) {
        cout << "Caught a float: " << e << endl;
    }
    catch (const char* e) {
        cout << "Caught a string: " << e << endl;
    }

    return 0;
}
```

Exception Handling with Class Objects

C++ allows you to throw and catch objects of user-defined types (classes). This is useful for passing detailed error information.

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

class DivisionByZero {
public:
    void errorMessage() {
        cout << "Error: Division by zero!" << endl;
    }
};
```

```
int main() {
    int numerator = 10, denominator = 0;

    try {
        if (denominator == 0) {
            DivisionByZero ex;
            throw ex;  // Throw an object of the class
        }
        cout << "Result: " << numerator / denominator << endl;
    }
    catch (DivisionByZero ex) {
        ex.errorMessage();
    }

    return 0;
}
```

Catching All Exceptions (catch(...))

C++ provides a special syntax to catch any type of exception. The catch(...) block acts as a **catch-all handler** and can be used when the type of exception is unknown.

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

int main() {
    try {
        throw 10;
    }
    catch (...) {  // Catch any type of exception
        cout << "Caught an unknown exception!" << endl;
    }

    return 0;
}
```

```
int main() {
    try  {
        throw 10;
    }
    catch (char *e)  { cout << "Caught " << e; }
    catch (...)  { cout << "Default Exception\n"; }
    return 0;
}
```

// What will be the output?

```
int main() {
   try  {
      throw "10";
   }
   catch (const char *e)  { cout << "Caught " << e; }
   catch (...)  { cout << "Default Exception\n"; }

 }
```

Output

Caught 10

Throwing Exceptions from Functions

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

int divide(int a, int b) {
    if (b == 0)
        throw "Division by zero!";
    return a / b;
}

int main() {
    int x = 10, y = 0;

    try {
        cout << "Result: " << divide(x, y) << endl;
    }
    catch (const char* e) {
        cout << "Error: " << e << endl;
    }

    return 0;
}
```

Multiple catch is possible

```
try {
//some code
throw exception;
}
catch(specific type exception) {//some code }
…
catch(generic type exception) { //some code }
```

Catch blocks must be ***ordered*** so that most specific exceptions get caught before generic exceptions.

Exception Specification

- C++ provides a mechanism to ensure that a given function is limited to throw only a specified list of exceptions.

```
void translate() throw(unknown_word,bad_grammar) {
/* ... */
}
```

- It will not throw any exception other than *unknown _word* or *bad_grammar*

```
class unknown_word{};
class bad_grammar{};
void translate(int a) throw(unknown_word,bad_grammar) {
if (a==0)   throw unknown_word();
else if(a==1) throw bad_grammar();
else throw 10;
}
int main(){
try{ translate(22);}
 catch(unknown_word u){cout<<" unknown word";}
 catch(bad_grammar g){cout<<" bad grammar";}
 catch(...) {cout<<"Default catch";}
}
```

Output

terminate called after throwing an instance of 'int'

Aborted (core dumped)

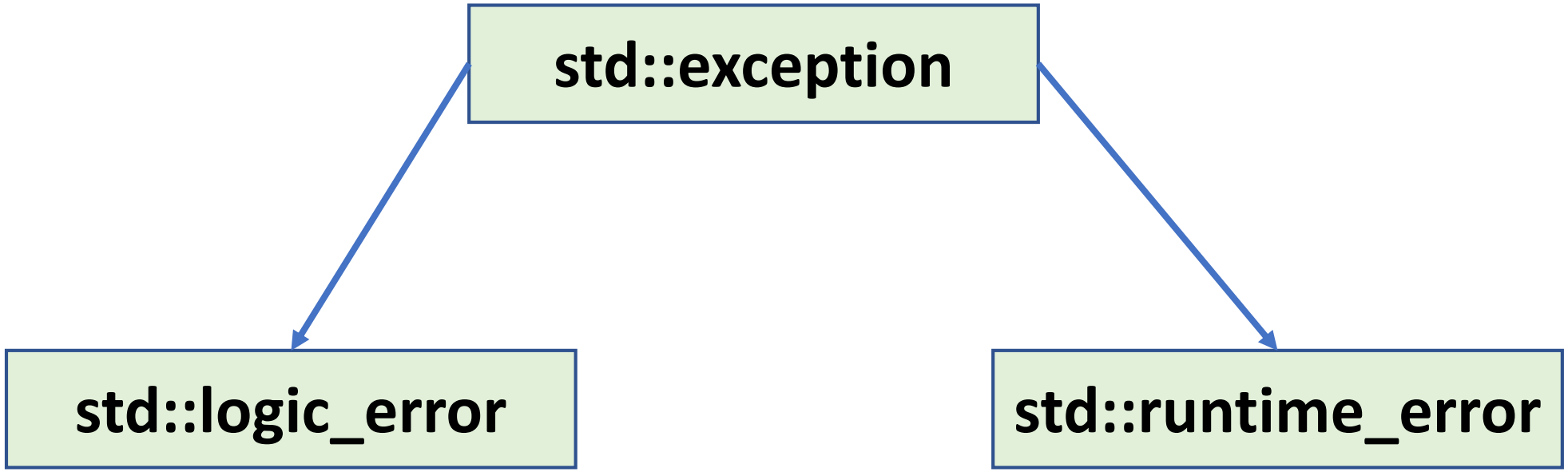
Standard Library Exceptions in C++

In C++, exceptions can be thrown and handled using the **Standard Library Exceptions**. These exceptions are part of the <exception> and <stdexcept> headers and represent a collection of predefined exception classes for common error conditions like runtime errors, out-of-range errors, bad memory allocation, etc.

C++ provides these **standard exception classes** to help handle specific types of errors more effectively and systematically. These classes follow a hierarchy, with std::exception being the base class from which all standard exceptions are derived.

Standard Exception Class Hierarchy

At the top of the hierarchy is the **std::exception** class, from which all other standard exception types are derived. Below are the commonly used standard exception classes:



+-- std::invalid_argument

+-- std::domain_error

+-- std::length_error

+-- std::out_of_range

+-- std::range_error

+-- std::overflow_error

+-- std::underflow_error

std::exception

Base Class for All Exceptions:-

std::exception is the base class for all standard exceptions. It provides the virtual member function **what()** that returns an error message.

You can catch any exception derived from **std::exception** by catching a reference to **std::exception.**

Example

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

int main() {
    try {
        throw std::exception();
    }
    catch (const std::exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

Output:
Caught exception: std::exception

std::exception::what() returns a generic message "std::exception"

std::logic_error

- Std::logic_error is a derived class of std::exception used for errors in the logic of the program (e.g., domain errors, invalid arguments).
- These errors represent conditions that could be prevented with proper input validation.

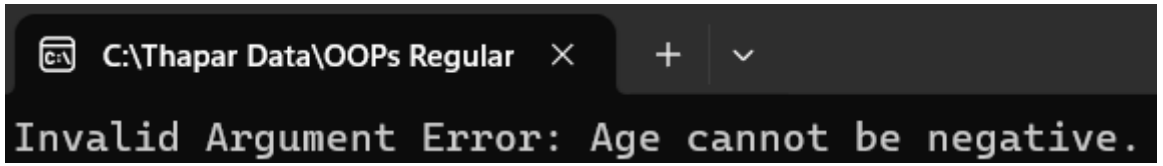
Derived Classes:

- **std::invalid_argument**: Thrown when an argument is invalid.
- **std::domain_error**: Thrown when a mathematical domain error occurs.
- **std::length_error**: Thrown when an object exceeds its maximum size.
- **std::out_of_range**: Thrown when an attempt is made to access an element outside the valid range.

Example

```cpp
#include <iostream>
#include <stdexcept>  // For standard exceptions
#include <cmath>      // For sqrt()
#include <vector>
#include <string>

using namespace std;

// Function to demonstrate std::invalid_argument
void checkAge(int age) {
    if (age < 0) {
        throw invalid_argument("Age cannot be negative.");
    } else {
        cout << "Valid age: " << age << endl;
    }
}
// Function to demonstrate std::domain_error
double calculateSquareRoot(double num) {
    if (num < 0) {
        throw domain_error("Cannot compute the square root of a negative number.");
    }
    return sqrt(num);
}
```

```cpp
// Function to demonstrate std::length_error
void createLargeString(size_t size) {
    string s;
    if (size > s.max_size()) {
        throw length_error("String size exceeds maximum allowable size.");
    }
    s.resize(size);  // Try to resize the string
    cout << "String of size " << size << " created successfully." << endl;
}
// Function to demonstrate std::out_of_range
void accessVectorElement(const vector<int>& v, size_t index) {
    if (index >= v.size()) {
        throw out_of_range("Index is out of range.");
    }
    cout << "Element at index " << index << ": " << v[index] << endl;
}

int main() {
    // Demonstrate std::invalid_argument
    try {
        checkAge(-5);  // Invalid age
    }
    catch (const invalid_argument& e) {
        cout << "Invalid Argument Error: " << e.what() << endl;
    }
```



Example

```
    // Demonstrate std::domain_error
    try {
        double result = calculateSquareRoot(-9.0);  // Invalid square root operation
        cout << "Square root: " << result << endl;
    }
    catch (const domain_error& e) {
        cout << "Domain Error: " << e.what() << endl;
    }
    // Demonstrate std::length_error
    try {
        createLargeString(100000000000);  // Exceeds maximum size of string
    }
    catch (const length_error& e) {
        cout << "Length Error: " << e.what() << endl;
    }
    // Demonstrate std::out_of_range
    vector<int> vec = {1, 2, 3, 4, 5};
    try {
        accessVectorElement(vec, 10);  // Index is out of range
    }
    catch (const out_of_range& e) {
        cout << "Out of Range Error: " << e.what() << endl;
    }
    return 0;
}
```

```
Domain Error: Cannot compute the square root of a negative number.
```

std::runtime_error

- Std::runtime_error is another derived class of std::exception and is used for errors that occur during the runtime of the program.
- These errors cannot usually be predicted or checked at compile time.

Derived Classes:

- **std::range_error**: Thrown when a result of a computation is out of range.
- **std::overflow_error**: Thrown when an arithmetic overflow occurs.
- **std::underflow_error**: Thrown when an arithmetic underflow occurs.

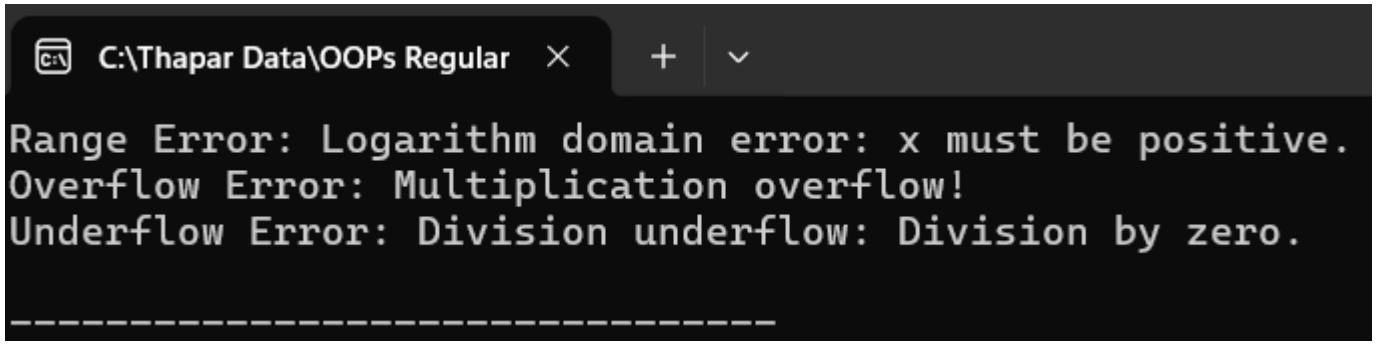
Example

```cpp
#include <iostream>
#include <stdexcept>  // For standard exceptions
#include <limits>     // For numeric limits
#include <cmath>      // For pow() function

using namespace std;

// Function to demonstrate std::range_error
double computeLog(double x) {
    if (x <= 0) {
        throw range_error("Logarithm domain error: x must be positive.");
    }
    return log(x);  // Compute natural logarithm
}

// Function to demonstrate std::overflow_error
int multiplyLargeNumbers(int a, int b) {
    if (a > 0 && b > 0 && a > (numeric_limits<int>::max() / b)) {
        throw overflow_error("Multiplication overflow!");
    }
    return a * b;
}
```

```cpp
// Function to demonstrate std::underflow_error
double divideSmallNumbers(double a, double b) {
    if (b == 0) {
        throw underflow_error("Division underflow: Division by zero.");
    }
    return a / b;
}
int main() {
    // Demonstrate std::range_error
    try {
        cout << "Logarithm of -1: " << computeLog(-1) << endl;  // Invalid input for log
    }
    catch (const range_error& e) {
        cout << "Range Error: " << e.what() << endl;
    }

    // Demonstrate std::overflow_error
    try {
        int a = numeric_limits<int>::max() / 2;
        int b = 3;
        cout << "Multiplication of large numbers: " << multiplyLargeNumbers(a, b) << endl;
    }
    catch (const overflow_error& e) {
        cout << "Overflow Error: " << e.what() << endl;
    }
```

Example

```
    // Demonstrate std::underflow_error
    try {
        double verySmallNumber = numeric_limits<double>::min();
        double divisor = 0;  // Division by zero
        cout << "Division result: " << divideSmallNumbers(verySmallNumber, divisor) << endl;
    }
    catch (const underflow_error& e) {
        cout << "Underflow Error: " << e.what() << endl;
    }

    return 0;
}
```



UTA 018: OOPs
Templates in C++



Introduction

Templates in C++ allow you to write **generic code** that can work with any data type. Instead of writing multiple versions of the same function or class for different data types, templates let you create **type-independent functions and classes** that automatically adapt to the type of data they work with.

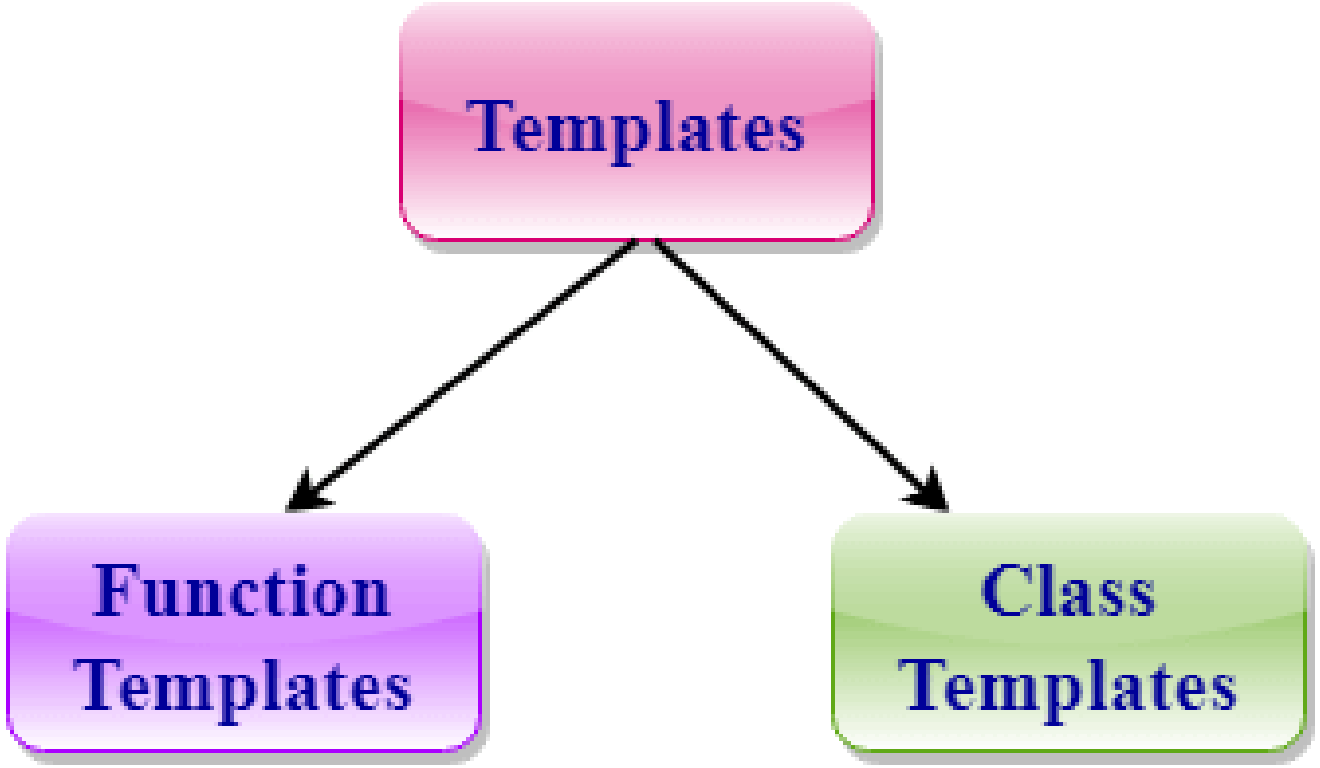
- Templates come in two main forms:
  - **Function Templates**: Used to create functions that can operate on any data type.
  - **Class Templates**: Used to create classes that can store, manipulate, or process data of any type.

Templates are a cornerstone of **generic programming** in C++, providing flexibility, reusability, and type safety.

Need of Template

- To reduce code *duplication* when supporting numerous data types

```
int MaxElement (int x, int y){ return x > y ? x : y ; }
float MaxElement (float x, float y){ return x > y ? x : y ; }
char MaxElement (char x, char y){ return x > y ? x : y ; }

T MaxElement (T x, T y){ return x>y?x:y;}
```



We can write a generic code for a function e.g. add() for integers, double, float etc.

We can write a generic code for a class, to manipulate group of member variables & functions e.g. linked list of strings, integers etc.

Function Template

A **template function** is a function that works with any data type. Instead of specifying a particular data type, you use a **placeholder type**. When the function is called, the compiler replaces the placeholder type with the actual data type of the argument(s).

Syntax of a Template Function:

```
template <class/typename T>
T functionName(T parameter1, T parameter2,…T Parameter_n)
{
        // Function implementation
}
```

Example:1

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

template <class T>
T MaxElement(T x, T y){ return x>y?x:y;}

int main(){
cout<<MaxElement (10,20)<<endl;
cout<<MaxElement ('a','z')<<endl;
cout<<MaxElement (-2.5,7.7)<<endl;
}
```

Defining Function Template

Calling Function Template

Example:2

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

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

int main() {
    int x = 5, y = 10;
    double p = 3.5, q = 7.8;

    swapValues(x, y);     // Swaps two integers
    swapValues(p, q);     // Swaps two doubles

    cout << "Swapped integers: " << x << ", " << y << endl;
    cout << "Swapped doubles: " << p << ", " << q << endl;

    return 0;
}
```

Example: Bubble Sort without Template

```
void bubbleSort(int 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])
              swap(a[j], a[j - 1]);
}
```

Example: Bubble Sort with Template

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

Function Template with Multiple Parameters

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

template <typename T1, typename T2>
void displayPair(T1 a, T2 b) {
    cout << "First: " << a << ", Second: " << b << endl;
}

int main() {
    displayPair(10, 15.5);      // Integer and double
    displayPair("Hello", 20);  // String and integer

    return 0;
}
```

Example: Write a template for

```
int main(){
show(100,"hello hello");
show('k',1500);
show(1.23,2987);
}
```

Example: Write a template for

```
template <class T1, class T2>
void show(T1 a, T2 b){
cout<<a<<“, “<<b<<endl;
}
```

```
int main(){
show(100,"hello hello");
show('k',1500);
show(1.23,2987);
}
```

Function template with non-type parameter

- Non-type parameter is not a type (datatype) but a value e.g. 100
- They are used to initialize a class or to specify the sizes of class members

- template <class T, int size> // size is the non-type parameter

Function template with non-type parameter

```
#include<iostream>
using namespace std;
template <class T, int size>
void show(T a){cout<<a<<", "<<size;}
int main(){
show <char,10> ('c');
}
```

Overloading function templates

```
template <class T1, class T2>
void show(T1 a, T2 b){
        cout<<a<<", "<<b<<endl;
}
void show(int a, int b)    {
        cout<<”For integer cases";
}

 int main(){
 show(100,"hello hello");
 show(3,3);
 }
```

Usage of Template Argument

- **No-argument template function**

```
Template <class T>
T void (void)                    //error: T is not used as an argument
{
//…..
Return parameter;
}
```

- **Template-type Argument Unused**

```
Template <class T>
Void test (int x)                    //error: T is not used as an argument
{
//…..
Return parameter;
}
```

- **Usage of partial Number of Template Argument**

```
Template <class T, class U>
Void insert (T &x)                    //error: U is not used as an argument
{
//…..
Return parameter;
}
```

Class Template

```
// Template class example
template <class/typename T> class Test {
T var;
public:
Test (T i) {var=i;}
T divideBy2 () {return var/2;}
};
int main(){
Test <int> t1(50);
Test <double> t2(-10.20);
cout<<t1.divideBy2()<<endl;
cout<<t2.divideBy2()<<endl;
}
```

```
// Template function  example
#include <iostream>
using namespace std;
template <class T>
T MaxElement(T x, T y){
        return x>y?x:y;
}
int main(){
cout<<MaxElement(10,20)<<endl;
cout<<MaxElement('a','z')<<endl;
}
```

Defining function outside the template class

```
template <class/typename T>
class Test {
T var;
public:
Test (T i) {var=i;}
T divideBy2 ();
};
template <class T>
T Test<T> :: divideBy2(){return var/2;}
```

template class with non-type parameter

```
template <class/typename T, int n>
class Test {
T var;
public:
Test () {var = n; cout<<"n = "<< n <<endl;}
T divideBy2 () {return var/2;}
};
int main(){
Test <int,10> t1;
Test <double,20> t2;
cout<<t1.divideBy2()<<" "<<t2.divideBy2()<<endl;
}
```

Output:

```
n = 10
n = 20
5 10
```

Class Template with Multiple Parameters

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

template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) : first(a), second(b) {}

    void display() const {
        cout << "First: " << first << ", Second: " << second << endl;
    }
};

int main() {
    Pair<int, double> p1(10, 20.5);       // Pair of int and double
    Pair<string, char> p2("Hello", 'A'); // Pair of string and char

    p1.display();
    p2.display();

    return 0;
}
```

Advantages of Using Templates

- **Code Reusability**: Templates allow you to write code once and use it for different data types without duplication.
- **Type Safety**: The compiler enforces type checking at compile time, ensuring the correct data type is used.
- **Generic Programming**: Templates facilitate writing generic algorithms and data structures, making code more flexible and adaptable.

Template Specialization

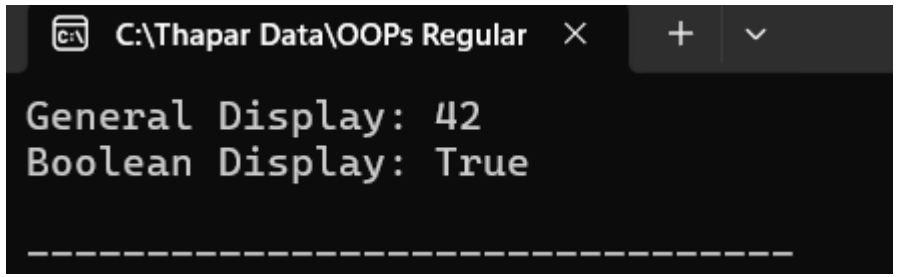
Template specialization allows you to define a specific implementation of a template function or class for a particular type. This is useful when you need specialized behavior for a certain data type, while still having a general template for other types.

Example of Class Template Specialization

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

template <typename T>
class Display {
public:
    void show(T value) {
        cout << "General Display: " << value << endl;
    }
};

// Specialization for `bool` type
template <>
class Display<bool> {
public:
    void show(bool value) {
        cout << "Boolean Display: " << (value ? "True" : "False") << endl;
    }
};
```

Example of Class Template Specialization

```
int main() {
    Display<int> intDisplay;
    intDisplay.show(42);    // Uses the general template

    Display<bool> boolDisplay;
    boolDisplay.show(true); // Uses the specialized template

    return 0;
}
```



- Display<int> uses the general Display template, while Display<bool> uses the specialized version specifically designed for bool.

- This feature is particularly useful when you need customized behavior for certain data types, such as printing true/false instead of 1/0.

Partial Specialization (Class Templates Only)

Partial specialization is only available for **class templates** (not function templates) and allows you to specialize a template for a subset of its template parameters.

Partial Specialization (Class Templates Only)

```cpp
template <typename T, typename U>
class Pair {
public:
    void show() {
        cout << "General Pair" << endl;
    }
};
// Partial specialization when both types are the same
template <typename T>
class Pair<T, T> {
public:
    void show() {
        cout << "Specialized Pair for the same types" << endl;
    }
};
int main() {
    Pair<int, double> p1;
    p1.show();  // Uses general template

    Pair<int, int> p2;
    p2.show();  // Uses partially specialized template

    return 0;
}
```

Note:

Pair<int, double> uses the general Pair template, while Pair<int, int> uses the partially specialized version where both template parameters are the same.

Variadic Templates

- Variadic templates allow a template to accept **an arbitrary number of template parameters**. This is useful for functions like printf-style functions or containers that handle multiple elements of varying types.

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

template <typename T>
void print(T value) {
    cout << value << endl;
}

template <typename T, typename... Args>
void print(T value, Args... args) {
    cout << value << " ";
    print(args...);
    // Recursive call with reduced argument pack
}

int main() {
    print(1, 2.5, "Hello", 'A');
    print(5);
    print("OOPs");
    print("Todays Topic", "Function and Class Template");
    return 0;
}
```

Note:

- The first print function is the base case for recursion.

- The second print function is the recursive function that processes one argument at a time.

- Variadic templates enable flexible function calls with any number and type of arguments, making them powerful for generic programming.

```cpp
#include<iostream>
using namespace std;
struct A{
    int x,y;
};
struct B{
    int x;
    double y;
};
template <typename T>
void Assign_A(T a, T b, A&s1){
    s1.x=a;
    s1.y=b;
}
template<typename T, typename U>
void Assign_B(T a, U b, B&s2){
    s2.x=a;
    s2.y=b;
}
```

```cpp
int main(void){
    A s1;
    B s2;
    Assign_A(4, 5, s1);
    cout<<"Struct A:"<<s1.x<<", "<<s1.y<<endl;
    Assign_B(3.4, 4.3, s2);
    cout<<"Struct B:"<<s2.x<<", "<<s2.y<<endl;
    return 0;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T>
class vector{
    T *v;
    int size;
    public:
        vector(int vector_size){
            size=vector_size;
            v= new T[size];
        }
        ~vector(){
            delete v;
        }
        T &elem(int i){
            if(i>=size)
                cout<<"error: out of range\n";
            return v[i];
        }
        void show();
};
template <class T>
void vector<T>::show(){
    for(int i=0;i<size;i++)
        cout<<elem(i)<<", ";
}
```

```cpp
int main(void){
    int i;
    vector<int> int_vect(5);
    vector<float> float_vect(4);
    for(i=0;i<5;i++)
        int_vect.elem(i)=i+1;
    for(i=0;i<4;i++)
        float_vect.elem(i)=float(i+1.5);
    cout<<"integer vector:"<<endl;
    int_vect.show();cout<<endl;

    cout<<"float vector:"<<endl;
    float_vect.show();
    return 0;
}
```

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

template<typename T>
int Search(T data[], T search_item, int low, int high){
    if(low>high)
        return -1;
    int mid=(low+high)/2;

    if(search_item<data[mid])
        return Search(data, search_item,low,mid-1);
    else
        if(search_item>data[mid])
            return Search(data,search_item,mid+1,high);
        return mid;
}
```

```cpp
int  main(void){
    int element, size, num[5],index;
    cout<<"How many elements:"<<endl;
    cin>>size;
    cout<<"enter the elements in ascending order:\n";
    for(int i=0;i<size;i++)
        cin>>num[i];
    cout<<"enter the elements to be searched:\n";
    cin>>element;
    index=Search(num,element,0,size);
    if(index==-1)
        cout<<"Element is not found\n";
    else
        cout<<"element foud at position:"<<index;
    return 0;
}
```

Inheritance of Class Template

Use of templates with respect to inheritance involves the following:

1. Derive a class template from a base class, which is a template class.
2. Derive a class template from a base class, which is a template class and add more template members in the derived class.
3. Derive a class from a base class which is not a template, and add template member to that class.
4. Derive a class from a base class which is a template class and restrict the template feature, so that the derived class and its derivatives do not have the template feature.

Syntax

```
template<class T>
class Base{
        //template type data and functions
};
template<class T1, …>
class Derive: public Base<T1, …>{
        //template type data and functions.
};
```

Example:01

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Derived class template inheriting from Base
template <typename T>
class Derived : public Base<T> {
public:
    Derived(T val) : Base<T>(val) {}
    void display() const {
        cout << "Derived value: " << this->value << endl;
    }
};
```

```cpp
int main() {
    Derived<int> obj(10);
    obj.show();     // Call Base class function
    obj.display();  // Call Derived class function

    return 0;
}
```

Example:02

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Derived class template with two template parameters
template <typename T, typename U>
class Derived : public Base<T> {
public:
    U extraValue;
    Derived(T val, U extra) : Base<T>(val), extraValue(extra) {}

    void display() const {
        cout << "Derived value: " << this->value << ", Extra value: " << extraValue << endl;
    }
};
```

```cpp
int main() {
    Derived<int, double> obj(10, 20.5);
    obj.show();      // Call Base class function
    obj.display();   // Call Derived class function

    return 0;
}
```

Example:03

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

// Non-template base class
class Base {
public:
    void show() const {
        cout << "Base class show function" << endl;
    }
};

// Derived class with a template member function
class Derived : public Base {
public:
    template <typename T>
    void display(T value) const {
        cout << "Derived class display with value: " << value << endl;
    }
};

int main() {
    Derived obj;
    obj.show();          // Call Base class function
    obj.display(10);     // Call Derived class template function with an int
    obj.display(3.14);   // Call Derived class template function with a double

    return 0;
}
```

Example:04

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

// Base class template
template <typename T>
class Base {
public:
    T value;
    Base(T val) : value(val) {}
    void show() const {
        cout << "Base value: " << value << endl;
    }
};

// Non-template Derived class inheriting from a specific instantiation of Base
class Derived : public Base<int> {  // `int` is specified, so Derived is not a template
public:
    Derived(int val) : Base<int>(val) {}
    void display() const {
        cout << "Derived class with fixed int type: " << this->value << endl;
    }
};

int main() {
    Derived obj(10);
    obj.show();       // Call Base class function
    obj.display();    // Call Derived class function

    return 0;
}
```

UTA 018: OOPs

Standard Template Library (STL)



Standard Template Library (STL)

The **Standard Template Library (STL)** is a powerful library in C++ that provides generic classes and functions designed to manage collections of data and perform common operations on them. STL allows developers to write efficient, reusable, and type-safe code. It enables the use of **containers, algorithms, and iterators** to work with data structures in a generic manner, making C++ more versatile for solving a wide range of problems.

STL provide general-purpose classes & functions to implement commonly used algorithms and data structures like vectors, lists, queues, and stacks.

Key Features of STL

- **Generic Programming**: STL uses templates to provide generic data structures and algorithms.
- **Efficiency**: STL algorithms and containers are highly optimized for performance.
- **Type Safety**: STL templates enforce type compatibility at compile-time.
- **Reusability**: The same code can be reused for different data types without modification.

Components of the STL

- Containers – Classes that store collections of elements. Eg. Vector

- Algorithms – Functions that perform operations on data in containers. Eg. sort()

- Iterator – Objects that enable traversal of elements in containers. Generalised concept of pointers to point an element in a container Eg. forward, bidirectional etc.

These components work together, with **containers** holding the data, **iterators** providing a way to access the data, and **algorithms** performing operations on the data.

Containers in STL

- **Containers** are objects that store collections of data. The STL provides different types of containers, each optimized for specific tasks. STL containers are divided into three categories:

- **Sequence Containers**: Maintain the order of elements based on insertion.
- **Associative Containers**: Automatically sort and store elements in a key-value fashion.
- **Container Adapters**: Provide a modified interface for existing containers.

Sequence Containers

Sequence containers store elements in a linear arrangement, preserving the order in which elements are inserted.

Container	Description	Example Use Case
vector	Dynamic array that supports fast random access and efficient resizing at the end.	Storing lists of data with frequent appends.
deque	Double-ended queue, allows insertion and deletion at both ends.	Managing data with access at both ends.
list	Doubly linked list, allows efficient insertion and deletion anywhere in the list.	Use when frequent insertions/deletions are needed.

Sequence Containers: vector

- A vector is a dynamic array that can grow or shrink in size. It provides random access to elements and supports efficient insertion and deletion at the end.

Library: <vector>

Constructors: vector(), vector(size), vector(size, value)

Capacity Functions:

Function	Description	Function	Description
size()	Returns the number of elements	capacity()	Returns the total allocated memory
empty()	Check if the vector is empty	resize(new_size)	Resizes the vector
reserve(capacity)	Allocates space for the specified capacity.		

Sequence Containers: vector

Modifiers Functions:

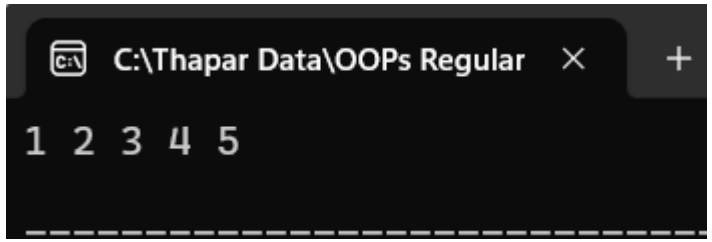
Function	Description
push_back()	Adds an element to the end.
pop_back()	Removes the last element.
insert(position, value)	Inserts an element at a given position.
erase(position)	Removes an element from a specified position.
clear()	Removes all elements.
emplace_back(arg…)	Constructs and adds an element at the end.

Access Functions:

Function	Description
at(index)	Accesses the element at the specified position (bounds-checked).
front()	Accesses the first element.
back()	Accesses the last element.
data()	Returns a pointer to the underlying array.

Sequence Containers: vector (Example)

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

int main() {
    vector<int> numbers = {1, 2, 3};
    numbers.push_back(4);  // Add element to the end
    numbers.push_back(5);

    for (int n : numbers) {
        cout << n << " ";
    }
    cout << endl;

    return 0;
}
```



Sequence Containers: deque

- A deque is a double-ended queue that allows insertion and deletion at both the beginning and the end. It is implemented as a dynamic array of fixed-size arrays, so accessing elements is almost as fast as in a vector.

Library: <deque>

Constructors: deque(), deque(size), deque(size, value)

Capacity Functions:

size(), empty()

Sequence Containers: deque

Modifier Functions:

Function	Description
push_back(value), push_front(value)	Adds elements to the end or beginning.
Pop_back(value), pop_front(value)	Removes elements from the end or beginning.
insert(position, value)	Inserts an element at a given position.
erase(position)	Removes an element from a specified position.
clear()	Removes all elements.

Access Functions:

Function	Description
at(index)	Accesses the element at the specified position (bounds-checked).
front()	Accesses the first element.
back()	Accesses the last element.
data()	Returns a pointer to the underlying array.

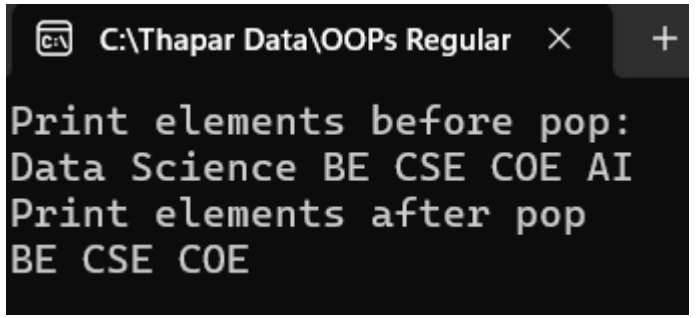
Sequence Containers: deque (Example)

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

int main() {
    deque<int> deq = {10, 20, 30};
    deq.push_front(5);   // Insert at the beginning
    deq.push_back(40);   // Insert at the end
    cout<<"Print elements before pop:\n";
    for (int n : deq) {
        cout << n << " ";
    }
    cout << endl;

    deq.pop_front();     // Remove from the beginning
    deq.pop_back();      // Remove from the end
    cout<<"Print elements after pop\n";
    for (int n : deq) {
        cout << n << " ";
    }
    cout << endl;

    deq.erase(deq.begin()+1);
    for (int n : deq) {
        cout << n << " ";
    }
    cout << endl;

    return 0;
}
```

```
Print elements before pop:
5 10 20 30 40
Print elements after pop
10 20 30
10 30
```

Sequence Containers: deque (Example)

```cpp
#include <iostream>
#include <deque>
#include<string>
using namespace std;
int main() {
    deque<string> deq = {"BE", "CSE", "COE"};
    deq.push_front("Data Science");   // Insert at the beginning
    deq.push_back("AI");   // Insert at the end
    cout<<"Print elements before pop:\n";
    for (string n : deq) {
        cout << n << " ";
    }
    cout << endl;

    deq.pop_front();     // Remove from the beginning
    deq.pop_back();      // Remove from the end
    cout<<"Print elements after pop\n";
    for (string n : deq) {
        cout << n << " ";
    }
    cout << endl;

    return 0;
}
```



Sequence Containers: list

- A list is a doubly linked list that allows efficient insertion and deletion from anywhere in the list. However, it does not provide random access to elements (like vectors and deques do).

Library: <list>

Constructors: list(), list(size), list(size, value)

Access Functions:

Function	Description
front()	Accesses the first element.
back()	Accesses the last element.

Sequence Containers: list

Modifier Functions:

Function	Description
push_back(value), push_front(value)	Adds elements to the end or beginning.
Pop_back(value), pop_front(value)	Removes elements from the end or beginning.
insert(position, value)	Inserts an element at a given position.
erase(position)	Removes an element from a specified position.
clear()	Removes all elements.
remove(value)	Removes all elements equal to a value.
sort()	Sorts the list.
reverse()	Reverse the list

Sequence Containers: list (Example)

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

int main() {
    list<int> lst = {1, 2, 3};
    lst.push_front(0);  // Add element to the beginning
    lst.push_back(4);   // Add element to the end
    cout<<"List elements:\n";
    for (int n : lst) {
        cout << n << " ";
    }
    lst.reverse();      // Reverse the list
    cout<<endl<<"List elements after using reverse function:\n";
    for (int n : lst) {
        cout << n << " ";
    }
    cout << endl;

    return 0;
}
```

```
C:\Thapar Data\OOPs Regular
List elements:
0 1 2 3 4
List elements after using reverse function:
4 3 2 1 0
```

Differences Between deque and list

Feature	List	Deque
Underlying Structure	Doubly linked list	Array of fixed-size arrays
Random Access	Not supported	Supported with [] and at().
Insertion/Deletion	Efficient at any position (O(1) at known position)	Efficient at front and back, slower in the middle
Memory Usage	Higher due to pointers per node	More compact, contiguous blocks of memory
Best Use Case	Frequent middle insertions/deletions, no need for random access	Frequent front/back operations and occasional random access

Associative Containers

Associative containers store elements in a way that enables fast retrieval based on keys. They use a binary tree or hash-based structure to keep elements sorted and quickly searchable.

Container	Description	Example Use Case
set	Stores unique elements in a sorted order.	Keeping a unique list of IDs or sorted items.
Map	Stores key-value pairs with unique keys, sorted by key.	Dictionary-like structure for fast lookups.
Multiset	Stores sorted elements, allowing duplicates.	Use when multiple identical elements are needed.
Multimap	Stores key-value pairs, allowing duplicate keys.	Multi-value dictionary-like structure.

Associative Containers: set

A set is an associative container that stores unique elements in a sorted order. It does not allow duplicate elements.

Library: <set>

Constructors: set(), set(initializer_list)

Capacity Functions: size(), empty()

Modifier:

insert(value): Inserts a unique value.

erase(value), clear()

Lookup Functions:

find(value): finds an element.

count(value): check if an element is exist.

lower_bound(value), upper_bound(value)

Associative Containers: set (Example)

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

int main() {
    // 1. Initialization
    set<int> s1 = {10, 20, 30};       // Initializer list
    set<int> s2(s1);                  // Copy constructor
    set<int> s3(s1.begin(), s1.end()); // Range-based initialization

    cout << "Initial set s1: ";
    for (int n : s1) cout << n << " ";
    cout << "\nCopy of s1 in s2: ";
    for (int n : s2) cout << n << " ";
    cout << "\nRange-based initialization of s3: ";
    for (int n : s3) cout << n << " ";
    cout << endl;

    // 2. Inserting Elements
    s1.insert(25); // Insert a single element
    s1.insert(20); // Attempt to insert a duplicate element (ignored)

    cout << "\ns1 after insertion of 25 and duplicate 20: ";
    for (int n : s1) cout << n << " ";

    // Inserting using hint
    auto it = s1.insert(s1.end(), 35); // Inserting with hint
    cout << "\nElement inserted using hint (35): " << *it;

    // 3. Deleting Elements
    s1.erase(10); // Erase element with value 10
    cout << "\ns1 after erasing 10: ";
    for (int n : s1) cout << n << " ";

    s1.erase(s1.begin()); // Erase first element
    cout << "\ns1 after erasing first element: ";
    for (int n : s1) cout << n << " ";

    // 4. Find and Count
    if (s1.find(25) != s1.end()) {
        cout << "\nElement 25 found in s1.";
    } else {
        cout << "\nElement 25 not found in s1.";
    }
    // count() returns 1 if found, otherwise 0
    cout << "\nCount of 30 in s1: " << s1.count(30);
```

Associative Containers: set (Example)

```cpp
    // 5. Iterating with Lower and Upper Bound
    cout << "\n\nIterating s1 using lower_bound and upper_bound:" << endl;
    for (auto it = s1.lower_bound(20); it != s1.upper_bound(35); ++it) {
        cout << *it << " ";
    }

    // 6. Clearing All Elements
    s1.clear();
    cout << "\n\ns1 after clear(): Size = ";
    cout<< s1.size() << ", Is empty? " << (s1.empty() ? "Yes" : "No") << endl

    return 0;
}
```

```
Initial set s1: 10 20 30
Copy of s1 in s2: 10 20 30
Range-based initialization of s3: 10 20 30

s1 after insertion of 25 and duplicate 20: 10 20 25 30
Element inserted using hint (35): 35
s1 after erasing 10: 20 25 30 35
s1 after erasing first element: 25 30 35
Element 25 found in s1.
Count of 30 in s1: 1

Iterating s1 using lower_bound and upper_bound:
25 30 35

s1 after clear(): Size = 0, Is empty? Yes
```

Associative Containers: map

A map is an associative container that stores key-value pairs in a sorted order by key. It does not allow duplicate keys.

Library: <map>

Constructors: map(), map(initializer_list)

Capacity Functions: size(), empty()

Modifier:

insert(pair): Inserts a key-value pair.

erase(key), clear()

Lookup Functions:

find(key): finds a key.

count(key): check if a key is exist,

at(key): access the value by key,

operator[]: inserts or access the value by key

Associative Containers: map (Example)

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

int main() {
    map<string, int> ages;
    ages["Alice"] = 25;
    ages["Bob"] = 30;

    for (const auto& pair : ages) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}
```

```
Alice: 25
Bob: 30
```

Associative Containers: map (Example)

Example in DevC++

Container Adaptors

Container adapters provide a restricted interface to the underlying container. They adapt existing containers like deque and vector to provide specialized behavior such as stack or queue operations.

Container	Description	Underlying Container Used
stack	LIFO (Last In, First Out) container	deque
queue	FIFO (First In, First Out) container	deque
priority_queue	Elements are ordered by priority	vector or deque

Example of stack

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

int main() {
    stack<int> stk;
    stk.push(1);
    stk.push(2);
    stk.push(3);

    while (!stk.empty()) {
        cout << stk.top() << " ";
        stk.pop();
    }
    cout << endl;

    return 0;
}
```

Iterators in STL

Iterators provide a way to access elements in containers. They behave like pointers, allowing traversal of container elements. Iterators can be used with containers to make code generic and flexible.

In C++, an **iterator** is an object that points to an element within a container. Think of an iterator as a pointer that “points” to the location of an element but doesn’t directly hold the element's value. Therefore, to access the actual value stored at the iterator’s position, you need to **dereference** the iterator.

Types of Iterators

- **Input Iterator**: Can read data sequentially.
- **Output Iterator**: Can write data sequentially.
- **Forward Iterator**: Can read or write in a single direction.
- **Bidirectional Iterator**: Can move both forward and backward.
- **Random Access Iterator**: Can jump to any element in constant time.

Using Iterators to Traverse Containers

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

int main() {
    vector<int> vec = {10, 20, 30, 40};

    // Using iterator to traverse
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Traverse Containers using **auto**

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

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

    // Using auto to iterate over vector elements
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Algorithms in STL

The STL provides a variety of **algorithms** for operations such as sorting, searching, counting, and modifying data. These algorithms work with iterators, making them compatible with any container.

Algorithm	Description
std::sort	Sorts elements in a range in ascending order.
std::find	Finds the first occurrence of a value in a range.
std::count	Counts occurrences of a specific value in a range.
std::reverse	Reverses the order of elements in a range.
std::accumulate	Computes the sum of elements in a range.

Example of Using Algorithms

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>  // For accumulate
using namespace std;
int main() {
    vector<int> vec = {5, 1, 4, 2, 3};
    // Sorting the vector

    sort(vec.begin(), vec.end());
    // Counting occurrences of a value

    int count3 = count(vec.begin(), vec.end(), 3);

    // Calculating sum of all elements
    int sum = accumulate(vec.begin(), vec.end(), 0);
    cout << "Sorted elements: ";
    for (int n : vec) {
        cout << n << " ";
    }
    cout <<endl<< "Count of 3: " << count3 << endl;
    cout << "Sum of elements: " << sum << endl;
    return 0;
}
```

```
Sorted elements: 1 2 3 4 5
Count of 3: 1
Sum of elements: 15
```

Function Objects (Functors)

- A **function object** or **functor** is an object that acts like a function. You can create function objects by overloading the **operator()**. Functors are often used as custom comparison functions in STL algorithms.

C++ string (as an STL) vs char array

- A string is an inbuilt C++ class that contains a char array but automatically manages it using built in functions



Basic string program

```
#include <string>
#include<iostream>
using namespace std;
int main(){
string greeting ="Hello";
cout<<greeting;
}
```

String concatenation using ‘+’

```
string firstName = "John";

string lastName = "Doe";

string fullName = firstName + " " + lastName;
```

Some basic string functions

```
string txt = ”aBcDeF";

cout << txt.length();

cout<< txt[0];  //for the first letter
```

Some basic string functions

```
getline(cin,txt);  //for including blank spaces

txt.substr(1, 3);   // substring BcD (from 1 till 3 char)

str1.compare(str2);  // compare string str1 with str2
```

Hashing for faster access & efficient storage

- Hashing maps given *value* to a particular *key*. This helps in faster access.

- Let H(x) maps the *value* at *key* **x%10** in an Array. Then *values* {11,12,13,14,15} will be stored at {1,2,3,4,5} *keys*.

- Hashing is one way encryption which means you cannot go from *key* to *value*. (1 to 11 for example)

Header file for hashing programs

- To run the following hash programs, you need #include<bits/stdc++>
- DevCpp may not have this file so either you can download and include it using #include”bits/stdc++”
- OR use an online compiler https://www.onlinegdb.com/online_c++_compiler

```
// Character hashing


#include<iostream>
using namespace std;
int main(){
    hash <char> hash_character;  //syntax for char type
    cout << "Hash key is: " << hash_character('a');
}
```

Output

Hash key is: 97

Looks like hash key is getting calculated same as ASCII value of the character (for char case)

// Integer hashing

```
#include<iostream>
using namespace std;
int main(){
    hash <int> hash_int;
    cout << "Hash key is: " << hash_int(597)<<endl;
    cout << "Hash key is: " << hash_int(-597)<<endl;
}
```

Output

Hash key is: 597

Hash key is: 18446744073709551019

Looks like hash key for positive integer is same as its value and for negative it is a complicated code (may be t’s compliment, not sure)

```
// String hashing

#include<iostream>
using namespace std;
int main(){
    hash <string> hash_string;
    cout << "Hash key is: " << hash_string("a")<<endl;
    cout << "Hash key is: " << hash_string("apple")<<endl;
}
```

Output

Hash key is: 4993892634952068459

Hash key is: 1377659774762457284 8

For both “a” and “apple” the hash values are complicated codes calculated internally.

Vector (another STL)

- Vector is a dynamic array with a size flexibility and direct access to any element.

```
// Simple vector to insert int values
#include<vector>
#include<iostream>
using namespace std;
int main()  {
vector<int> v1;
v1.push_back(11);
v1.push_back(22);
vector<int>::iterator i;
for(i=v1.begin();i!=v1.end();++i)
  cout << *i << endl;
}
```

Output

11

22

#include<algorithm> for vector functions

By including *algorithm* header file, many inbuilt functions can be executed on vector such as:

- sort (values.begin(), values.end());
- reverse (values.begin(), values.end());
- random_shuffle (values.begin(), values.end());
- count (values.begin(), values.end(), 0); //count zeros

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main()  {
vector<int> v1;
v1.push_back(11);  v1.push_back(33); v1.push_back(22);
sort (v1.begin(), v1.end());
vector<int>::iterator i;
for(i=v1.begin();i!=v1.end();++i)
  cout<<*i<<endl;
cout<< “Max = “<<*max_element (v1.begin(), v1.end());
}
```



Output

11

22

33

Max = 33

String and Vector classes vs C-style pointers

(1) String constant of C++

In C++ one must use *const* for int

const char* str = ”This is a const string in C++";

Otherwise warning message -

main.cpp: In function ‘int main()’: main.cpp:4:13: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings] char* str = "This is a const string in C++";

(2) Cannot modify a string

```
 int main() {
    char* str = "Hello";
//warning – it should be const char *
    str[1] = 'o';
}
```

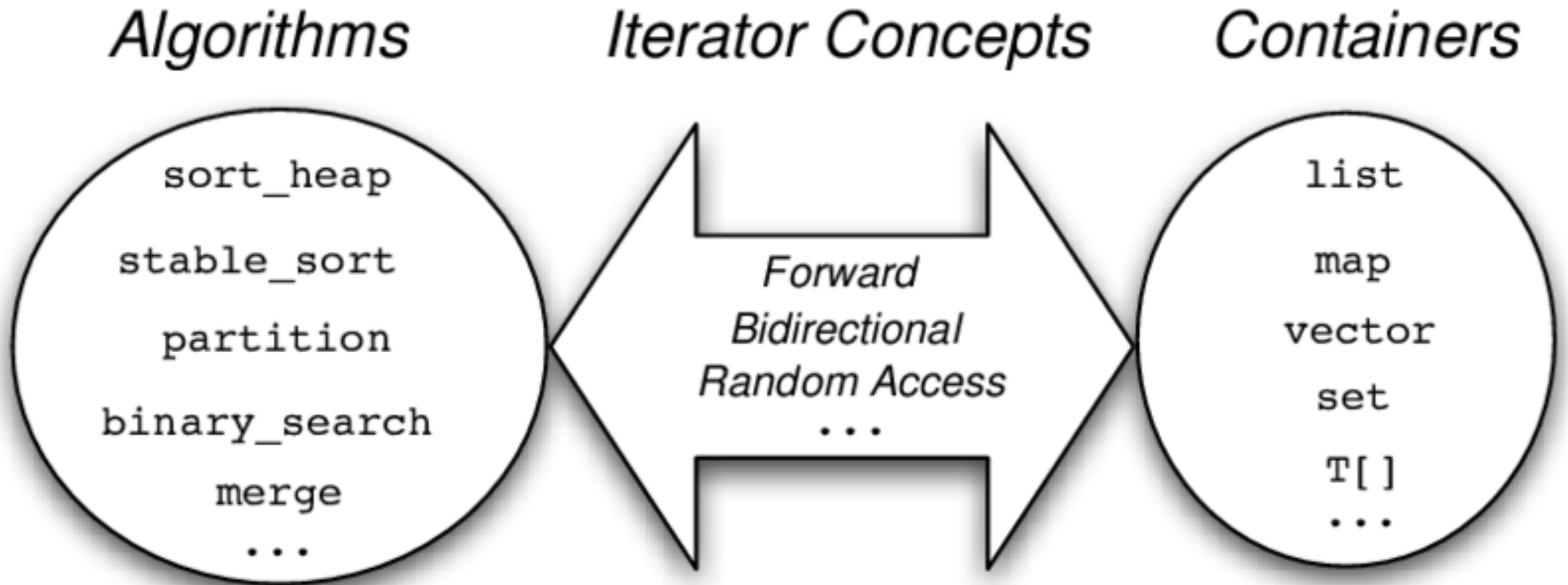
Output - Segmentation fault

Using char[] modification is possible

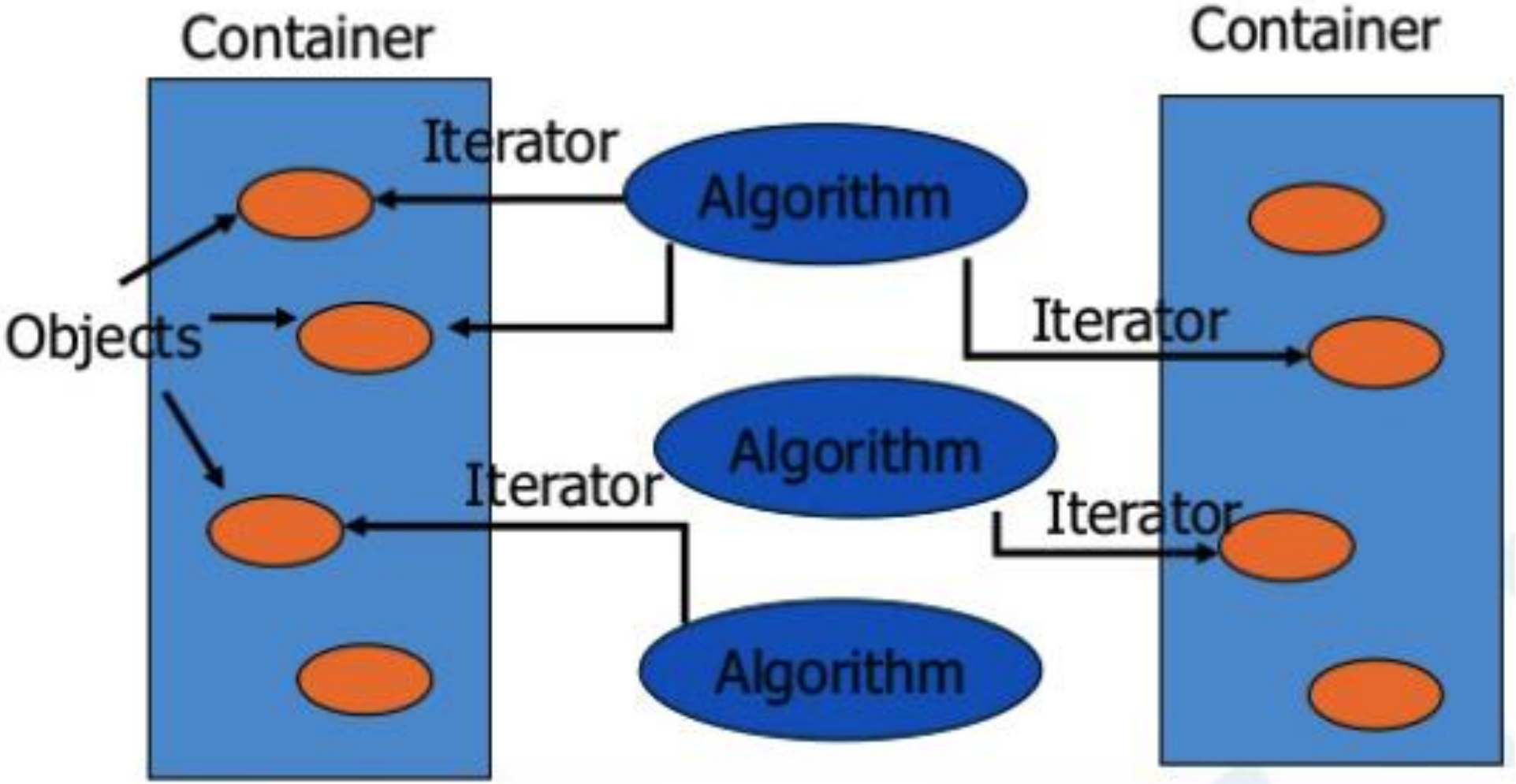
```
int main() {
  char str[] = "Hello";
    str[1] = 'o';
    cout << str << endl;
}
// Output will be Hollo
```

(3) char* is faster

- string will likely have more overhead


Algorithms
Iterator Concepts
Containers
sort_heap
stable_sort
partition
binary_search
merge
...
Forward
Bidirectional
Random Access
...
list
map
vector
set
T[]
...

Algorithms use iterators to interact with containers



Thank You