Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

TEMPLATE PROGRAMMING: FUNCTION & CLASS TEMPLATES

CHAPTER 1: INTRODUCTION TO TEMPLATE PROGRAMMING

Template programming is a powerful feature in C++ that enables **generic programming**, allowing developers to write **flexible, reusable, and type-independent** code. Instead of writing multiple versions of a function or a class for different data types, **templates** allow the definition of a single structure that works with any data type. This enhances code maintainability and reduces redundancy.

Templates are particularly useful when designing **libraries and frameworks** where functions or classes need to operate on different types of data. For instance, sorting algorithms, mathematical operations, and data structures such as **linked lists, stacks, and queues** benefit significantly from templates. Without templates, developers would need to define the same function separately for each data type, leading to **code duplication** and increased maintenance effort.

For example, consider a **swap function** that exchanges the values of two variables:

```
void swapInt(int &a, int &b) {
   int temp = a;
   a = b;
```

```
    b = temp;
}

void swapDouble(double &a, double &b) {
    double temp = a;
    a = b;
    b = temp;
}
```

Instead of defining separate functions for **int** and **double**, we can create a **function template**:

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

This single function works with **any data type**, making it more efficient and scalable.

Benefits of Template Programming

- **Code Reusability:** Write one function or class that works for multiple data types.
- **Type Safety:** The compiler checks type correctness at compile time, reducing runtime errors.

- **Reduced Code Duplication:** Eliminates the need to write separate functions/classes for each data type.
- **Scalability:** Makes it easier to extend functionality for new data types.

Templates form the foundation of **Standard Template Library (STL)** in C++, which provides powerful built-in generic data structures like **vectors, stacks, and maps**.

CHAPTER 2: FUNCTION TEMPLATES

A **function template** allows a function to work with different data types using **generic placeholders**. The function logic remains the same, but the data type adapts based on the argument type.

Syntax of Function Templates

```
template <typename T>
T add(T a, T b) {
   return a + b;
}
```

Here:

- template <typename T> tells the compiler that T is a **generic type parameter**.
- T add(T a, T b) defines a function that takes two parameters of type T and returns a value of the same type.

Example: Using Function Templates

```
#include <iostream>
```

```
using namespace std;


template <typename T>
T multiply(T a, T b) {
    return a * b;
}


int main() {
    cout << "Multiplication of integers: " << multiply(5, 10) << endl;
    cout << "Multiplication of doubles: " << multiply(3.5, 2.2) << endl;
    return 0;
}
```

Output:

Multiplication of integers: 50

Multiplication of doubles: 7.7

The function multiply() works for both int and double data types, demonstrating **code reusability**.

Advantages of Function Templates

- Enables writing **generic algorithms** that work with various data types.
- Reduces the number of function overloads, improving **code efficiency**.

- Provides **compile-time type checking**, avoiding runtime errors.

CHAPTER 3: CLASS TEMPLATES

A **class template** allows defining a class that works with any data type, making it useful for implementing **generic data structures** such as **arrays, linked lists, and stacks**.

Syntax of Class Templates

```
template <typename T>
class Box {
private:
  T value;
public:
  Box(T val) : value(val) {}

  void show() {
    cout << "Value: " << value << endl;
  }
};
```

Here:

- template <typename T> declares T as a **generic type**.
- Box<T> is a **generic class** that can store any data type.

Example: Using Class Templates

```
#include <iostream>

using namespace std;


template <typename T>

class Box {

private:

   T value;

public:

   Box(T val) : value(val) {}


   void show() {

      cout << "Value: " << value << endl;

   }

};


int main() {

   Box<int> intBox(100);

   intBox.show();


   Box<string> strBox("Template Programming");
```

```
    strBox.show();


    return 0;
}
```

Output:

Value: 100

Value: Template Programming

This demonstrates how **class templates** enable **flexible and reusable** implementations of generic types.

Advantages of Class Templates

- Allows designing **generic data structures** such as stacks and queues.
- Reduces **redundant code**, making programs more **efficient**.
- Provides **compile-time type safety**, preventing invalid operations.

CHAPTER 4: CASE STUDY – IMPLEMENTING A GENERIC STACK

Scenario

A software company wants to implement a **stack data structure** that works for multiple data types (**integers, doubles, and strings**) without rewriting the class multiple times.

Solution: Using Class Templates for a Stack

```
#include <iostream>
```

```
using namespace std;


template <typename T>
class Stack {
private:
   T arr[5]; // Fixed size stack
   int top;
public:
   Stack() : top(-1) {}


   void push(T val) {
      if (top < 4) {
         arr[++top] = val;
         cout << "Pushed: " << val << endl;
      } else {
         cout << "Stack Overflow\n";
      }
   }


   void pop() {
      if (top >= 0) {
```

```
            cout << "Popped: " << arr[top--] << endl;
        } else {
            cout << "Stack Underflow\n";
        }
    }
};

int main() {
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    intStack.pop();

    Stack<string> strStack;
    strStack.push("Hello");
    strStack.push("Templates");
    strStack.pop();

    return 0;
}
```

Output:

Pushed: 10

Pushed: 20

Popped: 20

Pushed: Hello

Pushed: Templates

Popped: Templates

This demonstrates how **templates** simplify the implementation of **generic data structures**.

CHAPTER 5: EXERCISES

Exercise 1: Function Template for Finding the Maximum of Two Numbers

Write a **function template** that takes two numbers and returns the maximum. Test it with **integers and doubles**.

Exercise 2: Class Template for a Simple Calculator

Implement a **class template** for a calculator that performs **addition, subtraction, multiplication, and division** on different data types.

Exercise 3: Implementing a Generic Linked List

Use **class templates** to create a **linked list** that can store **any data type**. Implement **insert and display operations**.

CONCLUSION

Template programming is an essential feature in C++ that enhances **code reusability, efficiency, and maintainability**. **Function templates** allow writing **generic algorithms**, while **class templates** enable implementing **generic data structures** like stacks and linked lists. **Case studies and exercises** reinforce the importance of templates in real-world applications.

STL (Standard Template Library) – Vectors, Lists, Maps

Chapter 1: Introduction to Standard Template Library (STL)

The **Standard Template Library (STL)** in C++ is a powerful collection of pre-built template classes and functions that provide efficient implementations of commonly used data structures and algorithms. STL enhances C++ programming by offering ready-to-use implementations of **dynamic arrays, linked lists, hash tables, trees, and other essential data structures**. This saves developers from writing boilerplate code, making programs more efficient and maintainable.

STL consists of three major components:

1. **Containers:** These store collections of data and come in different types such as **vectors, lists, and maps**.
2. **Algorithms:** These perform operations like **sorting, searching, and manipulation** on containers.
3. **Iterators:** These provide a uniform way to traverse elements in containers.

Using STL improves **code reusability, performance, and scalability**. For example, rather than manually implementing a linked list, developers can use std::list, which already provides all necessary operations. Similarly, std::map allows storing key-value pairs, reducing the complexity of searching and retrieving data.

By leveraging STL, programmers can write **highly optimized and bug-free** programs with minimal effort. STL is widely used in

competitive programming, software development, and system design, making it an essential tool for every C++ programmer.

CHAPTER 2: VECTORS – DYNAMIC ARRAY IMPLEMENTATION

A **vector** in STL is a **dynamic array** that can **automatically resize** itself when new elements are inserted or deleted. Unlike standard arrays, vectors manage memory dynamically, eliminating the need for manual allocation.

Key Features of Vectors

- **Dynamic Sizing:** No need to predefine size, it grows and shrinks automatically.
- **Fast Access:** Provides constant-time (O(1)) access to elements using an index.
- **Efficient Insertion and Deletion:** New elements can be added at the end in O(1) time complexity.

Example: Using Vectors in C++

```
#include <iostream>

#include <vector>

using namespace std;


int main() {

  vector<int> numbers = {10, 20, 30, 40}; // Initializing a vector

  numbers.push_back(50);  // Add element at end
```

```
    numbers.pop_back();    // Remove last element


    cout << "Vector elements: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;


    cout << "Size of vector: " << numbers.size() << endl;
    return 0;
}
```

Output:

Vector elements: 10 20 30 40

Size of vector: 4

Advantages of Vectors

1. **Dynamic resizing** eliminates the need for manual memory management.
2. **Easy traversal** using iterators and range-based loops.
3. **Efficient memory allocation** and automatic deallocation.

Vectors are widely used in applications where **fast element access and dynamic resizing** are required, such as **game development, database systems, and financial applications**.

CHAPTER 3: LISTS – LINKED LIST IMPLEMENTATION IN STL

A **list** in STL is an implementation of a **doubly linked list,** where each node stores a value and pointers to the next and previous nodes. Unlike vectors, lists provide efficient **insertion and deletion** at any position but have slower access times due to non-contiguous memory allocation.

Key Features of Lists

- **Fast Insertions/Deletions:** Adding or removing elements from any position takes O(1) time.
- **Efficient Memory Utilization:** No need for contiguous memory like arrays.
- **Supports Bidirectional Traversal:** Can be accessed forward and backward using iterators.

Example: Using Lists in C++

```
#include <iostream>

#include <list>

using namespace std;


int main() {

  list<int> myList = {10, 20, 30};  // Initializing a list

  myList.push_front(5);   // Insert at front

  myList.push_back(40);   // Insert at end
```

```
    myList.pop_front();    // Remove first element


    cout << "List elements: ";
    for (int num : myList) {
        cout << num << " ";
    }
    cout << endl;


    return 0;
}
```

Output:

List elements: 10 20 30 40

Advantages of Lists

1. **Efficient insertions and deletions** compared to vectors.
2. **No reallocation issues** since memory is dynamically allocated per node.
3. **Supports bidirectional traversal** using begin() and end() iterators.

Lists are ideal for **task scheduling, undo-redo functionality, and real-time event handling,** where frequent insertions and deletions are required.

CHAPTER 4: MAPS – KEY-VALUE PAIR IMPLEMENTATION

A **map** in STL is a data structure that stores **key-value pairs**, where each key is unique. It allows **fast lookup, insertion, and deletion** using self-balancing trees (Red-Black Trees).

Key Features of Maps

- **Ordered Storage:** Elements are stored in **sorted** order based on keys.
- **Efficient Searching:** Lookup time complexity is O(log N).
- **Key-Value Pairs:** Keys are unique, and values are associated with them.

Example: Using Maps in C++

```
#include <iostream>

#include <map>

using namespace std;


int main() {

   map<int, string> students;

   students[101] = "Alice";

   students[102] = "Bob";

   students[103] = "Charlie";


   cout << "Student List:" << endl;
```

```
    for (auto student : students) {
        cout << "ID: " << student.first << ", Name: " << student.second << endl;
    }

    return 0;
}
```

Output:

Student List:

ID: 101, Name: Alice

ID: 102, Name: Bob

ID: 103, Name: Charlie

Advantages of Maps

1. **Fast retrieval** using keys instead of searching through the entire dataset.
2. **Automatic sorting** based on keys, making range queries efficient.
3. **Supports complex data storage** with custom key-value mappings.

Maps are extensively used in **databases, caching mechanisms, and dictionary implementations**.

CHAPTER 5: CASE STUDY – STUDENT MANAGEMENT SYSTEM USING STL

Scenario

A university wants to develop a **student management system** that:

- Stores **student records** (ID, Name, Course) using map.
- Provides fast **search** based on student ID.
- Allows **adding, updating, and deleting** student records.

Solution: Implementing the System Using STL

```
#include <iostream>

#include <map>

using namespace std;


int main() {

  map<int, string> students;

  students[101] = "Alice - Computer Science";

  students[102] = "Bob - Mathematics";

  students[103] = "Charlie - Physics";


  int searchID;

  cout << "Enter Student ID to search: ";

  cin >> searchID;
```

```
    if (students.find(searchID) != students.end()) {
        cout << "Record Found: " << students[searchID] << endl;
    } else {
        cout << "Student not found." << endl;
    }

    return 0;
}
```

Output:

Enter Student ID to search: 102

Record Found: Bob - Mathematics

This case study demonstrates how **maps efficiently store and retrieve student data,** making them ideal for **database-driven applications**.

CHAPTER 6: EXERCISES

1. **Create a program using vector<int> that stores 10 numbers and sorts them in ascending order.**
2. **Implement a list<string> to store student names and allow adding/removing names dynamically.**
3. **Use map<string, double> to store product names with their prices and implement a search function.**

CONCLUSION

STL provides efficient **containers like vectors, lists, and maps** that simplify **dynamic memory management, searching, and data organization**. Mastering STL enhances **problem-solving skills** and enables the development of **high-performance applications**.

NETWORK PROGRAMMING WITH C++

CHAPTER 1: INTRODUCTION TO NETWORK PROGRAMMING

Network programming is a crucial domain in software development that enables communication between different systems over a network. In C++, network programming allows developers to build applications such as **web servers, chat applications, file transfer systems, and remote database connections**. It involves the use of **sockets**, which act as endpoints for communication between two nodes in a network.

Importance of Network Programming in C++

- **Client-Server Communication:** Allows multiple clients to connect to a centralized server for data exchange.
- **Internet Applications:** Used in building web servers, browsers, and APIs.
- **Distributed Computing:** Enables systems to share resources efficiently over a network.
- **Remote Control Systems:** Helps in managing devices and servers remotely.

C++ provides various libraries for network programming, including **Berkeley Sockets (POSIX), Windows Winsock, and Boost.Asio**. These libraries offer functionalities for creating, managing, and closing network connections in different environments.

Understanding network programming involves learning **protocols (TCP, UDP), IP addressing, and data transfer techniques**, which are essential for developing robust networking applications.

CHAPTER 2: BASICS OF SOCKETS IN C++

A **socket** is an endpoint for sending or receiving data across a network. Sockets use **Internet Protocol (IP)** for communication, and they support two major types of transport protocols:

1. **TCP (Transmission Control Protocol)** – Ensures reliable, ordered, and error-checked delivery of data.
2. **UDP (User Datagram Protocol)** – Provides faster, connectionless communication without guaranteed delivery.

Creating a Simple Socket in C++

To create a socket, the following steps are involved:

1. **Initialize the socket** using socket().
2. **Bind** the socket to an address and port using bind().
3. **Listen** for incoming connections (for TCP servers).
4. **Accept or connect** to a client/server.
5. **Send and receive data** using send() and recv().
6. **Close the socket** when done.

The following example demonstrates **creating a simple TCP socket** in C++.

```
#include <iostream>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

using namespace std;
```

```
int main() {
   int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
   if (serverSocket == -1) {
      cout << "Failed to create socket.\n";
      return -1;
   }

   sockaddr_in serverAddr;
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(8080);
   serverAddr.sin_addr.s_addr = INADDR_ANY;

   if (bind(serverSocket, (struct sockaddr*)&serverAddr,
sizeof(serverAddr)) == -1) {
      cout << "Failed to bind socket.\n";
      return -1;
   }

   listen(serverSocket, 5);
   cout << "Server is listening on port 8080...\n";
```

```
    int clientSocket = accept(serverSocket, nullptr, nullptr);
    if (clientSocket == -1) {
        cout << "Failed to accept client connection.\n";
        return -1;
    }

    cout << "Client connected!\n";
    close(serverSocket);
    return 0;
}
```

This example sets up a basic **TCP server** that listens for connections on port 8080.

Chapter 3: Client-Server Communication Using TCP

A **TCP client-server model** allows two computers to communicate reliably. The server waits for client requests, processes them, and sends responses.

TCP Server Example in C++

This program initializes a server that accepts connections and sends a greeting message.

```
#include <iostream>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;

int main() {
   int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
   sockaddr_in serverAddr;
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(8080);
   serverAddr.sin_addr.s_addr = INADDR_ANY;

   bind(serverSocket, (struct sockaddr*)&serverAddr,
sizeof(serverAddr));
   listen(serverSocket, 5);

   cout << "Server is running...\n";
   int clientSocket = accept(serverSocket, nullptr, nullptr);
   const char* message = "Hello from server!";
   send(clientSocket, message, strlen(message), 0);
```

```
    close(clientSocket);
    close(serverSocket);
    return 0;
}
```

TCP Client Example in C++

A client program connects to the server and receives the greeting message.

```
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
using namespace std;

int main() {
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(8080);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    connect(clientSocket, (struct sockaddr*)&serverAddr,
sizeof(serverAddr));


    char buffer[1024] = {0};

    recv(clientSocket, buffer, sizeof(buffer), 0);

    cout << "Server: " << buffer << endl;


    close(clientSocket);

    return 0;

}
```

Output

Server is running...

Client connected!

Server: Hello from server!

This demonstrates a **basic TCP connection**, where the server sends a message to the client.

CHAPTER 4: UNDERSTANDING UDP COMMUNICATION

Unlike TCP, **UDP (User Datagram Protocol)** is a **connectionless** protocol used for faster communication without guarantees of message delivery. UDP is ideal for **real-time applications like video streaming and gaming**.

UDP Server Example in C++

```
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
using namespace std;

int main() {
   int serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
   sockaddr_in serverAddr;
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(8080);
   serverAddr.sin_addr.s_addr = INADDR_ANY;

   bind(serverSocket, (struct sockaddr*)&serverAddr,
sizeof(serverAddr));
   cout << "UDP Server listening on port 8080...\n";

   char buffer[1024];
   sockaddr_in clientAddr;
   socklen_t addrLen = sizeof(clientAddr);
```

```
    recvfrom(serverSocket, buffer, sizeof(buffer), 0, (struct
sockaddr*)&clientAddr, &addrLen);

    cout << "Client: " << buffer << endl;


    close(serverSocket);

    return 0;

}
```

UDP Client Example in C++

```
#include <iostream>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <cstring>

using namespace std;


int main() {

    int clientSocket = socket(AF_INET, SOCK_DGRAM, 0);

    sockaddr_in serverAddr;

    serverAddr.sin_family = AF_INET;

    serverAddr.sin_port = htons(8080);

    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    const char* message = "Hello UDP Server!";

    sendto(clientSocket, message, strlen(message), 0, (struct
sockaddr*)&serverAddr, sizeof(serverAddr));


    close(clientSocket);

    return 0;
}
```

This program sends a **UDP datagram** to the server, which processes and displays the message.

Chapter 5: Case Study – Implementing a Chat Application

Scenario

A company wants a **basic chat system** where multiple clients can send messages to a server, which relays messages to other clients.

Solution

We can build a **multi-client chat application** using TCP sockets, where:

- The **server** listens for client connections and forwards messages.
- The **clients** send messages to the server.

Using **multi-threading**, the server can handle multiple clients simultaneously.

CHAPTER 6: EXERCISES

1. **Modify the TCP server to handle multiple clients using multi-threading.**
2. **Implement a UDP-based file transfer application.**
3. **Build a simple web client that fetches data from a URL using HTTP over TCP.**

CONCLUSION

Network programming in C++ enables **real-time communication** between applications. Using **sockets (TCP & UDP), multi-threading, and data transmission protocols**, developers can build **efficient, scalable networking applications** such as **chat systems, web servers, and distributed computing environments**.

INTEGRATING C++ WITH DATABASES (SQL, NOSQL)

CHAPTER 1: INTRODUCTION TO DATABASE INTEGRATION WITH C++

Databases are essential for storing, managing, and retrieving large amounts of structured and unstructured data efficiently. Integrating **C++ with databases** allows developers to create **data-driven applications** such as **inventory management systems, banking applications, and real-time analytics platforms**.

C++ can connect with databases using **SQL (Structured Query Language) for relational databases** and **NoSQL (Non-Relational Databases) for flexible, schema-less storage**.

Types of Databases

1. **SQL Databases:** Structured data storage using **tables, rows, and columns**. Examples: **MySQL, PostgreSQL, SQLite, MS SQL Server**.
2. **NoSQL Databases:** Schema-less data storage supporting **key-value, document-based, column-family, and graph data models**. Examples: **MongoDB, Redis, Firebase, Cassandra**.

Why Use C++ for Database Integration?

- **High performance:** C++ is used in **real-time financial applications** and **high-speed data processing systems**.
- **Scalability:** Supports handling **large datasets** efficiently.
- **Flexibility:** Can connect with **both SQL and NoSQL databases** using APIs like **ODBC, MySQL Connector, PostgreSQL libpq, and MongoDB C++ Driver**.

Understanding **database integration** is crucial for developing **enterprise applications, cloud-based services, and AI-driven analytics tools**.

CHAPTER 2: CONNECTING C++ WITH SQL DATABASES (MYSQL, SQLITE)

Using MySQL with C++

To integrate **MySQL** with C++, we use the **MySQL Connector/C++** library, which provides an API to connect, execute queries, and retrieve data.

Steps to Connect C++ with MySQL

1. **Install MySQL and MySQL Connector for C++**
2. **Include necessary headers**
3. **Establish a connection to the database**
4. **Execute SQL queries**
5. **Fetch and display results**

Example: Connecting C++ with MySQL

```
#include <iostream>

#include <mysql/mysql.h>

using namespace std;


int main() {

   MYSQL* conn;
```

```
    conn = mysql_init(nullptr);


    if (conn == nullptr) {
        cout << "MySQL initialization failed!\n";
        return 1;
    }


    if (mysql_real_connect(conn, "localhost", "root", "password",
"testdb", 3306, nullptr, 0)) {
        cout << "Connected to MySQL Database!\n";
    } else {
        cout << "Connection Failed: " << mysql_error(conn) << endl;
    }


    mysql_close(conn);
    return 0;
}
```

This program initializes MySQL, connects to a local database named testdb, and verifies the connection.

Using SQLite with C++

SQLite is a **lightweight, serverless database** that is ideal for small applications.

Example: Connecting C++ with SQLite

```
#include <iostream>

#include <sqlite3.h>

using namespace std;


int main() {

   sqlite3* db;

   int result = sqlite3_open("test.db", &db);


   if (result) {

      cout << "Error opening SQLite database: " << sqlite3_errmsg(db)
<< endl;

   } else {

      cout << "Connected to SQLite Database!\n";

   }


   sqlite3_close(db);

   return 0;

}
```

This connects to an SQLite database file test.db. If it doesn't exist, SQLite automatically creates it.

CHAPTER 3: PERFORMING SQL QUERIES IN C++

Executing SQL Queries in MySQL

Once connected, we can **insert, update, retrieve, and delete** data using SQL queries.

Example: Creating a Table and Inserting Data

```
const char* createTable = "CREATE TABLE Students (ID INT PRIMARY KEY, Name TEXT, Age INT);";

const char* insertData = "INSERT INTO Students (ID, Name, Age) VALUES (1, 'Alice', 20);";

if (mysql_query(conn, createTable) == 0) {
   cout << "Table created successfully!\n";
}


if (mysql_query(conn, insertData) == 0) {
   cout << "Data inserted successfully!\n";
}
```

Fetching Data from MySQL

```
MYSQL_RES* res;

MYSQL_ROW row;
```

```
mysql_query(conn, "SELECT * FROM Students");

res = mysql_store_result(conn);


while ((row = mysql_fetch_row(res))) {

   cout << "ID: " << row[0] << " Name: " << row[1] << " Age: " << row[2] << endl;

}
```

This retrieves all student records and prints them.

CHAPTER 4: CONNECTING C++ WITH NOSQL DATABASES (MONGODB, REDIS)

NoSQL databases like **MongoDB and Redis** store data in a **flexible format** without requiring predefined schemas.

Using MongoDB with C++

MongoDB stores data in **JSON-like documents** and provides high scalability. The **MongoDB C++ Driver** allows C++ programs to interact with a MongoDB database.

Example: Connecting C++ with MongoDB

```
#include <iostream>

#include <mongocxx/client.hpp>

#include <mongocxx/instance.hpp>

using namespace std;
```

```
using namespace mongocxx;


int main() {
    instance inst{};
    client conn{uri{"mongodb://localhost:27017"}};
    auto db = conn["testdb"];


    cout << "Connected to MongoDB!\n";
    return 0;
}
```

This establishes a connection to a MongoDB server running on localhost.

Inserting Data into MongoDB

```
document doc{};
doc << "name" << "Alice" << "age" << 20;
db["students"].insert_one(doc.view());
```

This inserts a document { "name": "Alice", "age": 20 } into the students collection.

Retrieving Data from MongoDB

```
auto cursor = db["students"].find({});
for (auto&& doc : cursor) {
    cout << bsoncxx::to_json(doc) << endl;
```

```
}
```

This retrieves and prints all student records in JSON format.

CHAPTER 5: CASE STUDY – BUILDING A STUDENT DATABASE SYSTEM

Scenario

A university wants to develop a **student management system** that:

- **Stores student records** in a database.
- **Supports CRUD operations** (Create, Read, Update, Delete).
- **Works with both SQL (MySQL) and NoSQL (MongoDB)** for flexible data storage.

Solution: Implementing a Hybrid Database System

1. **Use MySQL for structured relational data** (e.g., student grades, course enrollments).
2. **Use MongoDB for document-based storage** (e.g., student profiles, activity logs).
3. **Provide a unified C++ interface** to query and update both databases.

Hybrid Database Query Example

```
void addStudentSQL(int id, string name, int age) {
    string query = "INSERT INTO Students (ID, Name, Age) VALUES (" + to_string(id) + ", '" + name + "', " + to_string(age) + ");";
    mysql_query(conn, query.c_str());
}
```

```
void addStudentNoSQL(string name, int age) {
    document doc{};
    doc << "name" << name << "age" << age;
    db["students"].insert_one(doc.view());
}
```

This allows adding a student **to both MySQL and MongoDB**, providing a **scalable hybrid solution**.

CHAPTER 6: EXERCISES

1. **Modify the MySQL program to allow user input for adding students dynamically.**
2. **Implement a C++ program that connects to MongoDB and retrieves data based on a search query.**
3. **Create a real-time analytics dashboard that fetches data from both SQL and NoSQL databases.**

CONCLUSION

Integrating **C++ with SQL and NoSQL databases** enables the development of **scalable, high-performance applications** for diverse use cases. By leveraging **MySQL for structured data** and **MongoDB for document-based storage**, developers can build **powerful, real-time data-driven applications**.

C++ FOR GAME DEVELOPMENT & EMBEDDED SYSTEMS

CHAPTER 1: INTRODUCTION TO C++ FOR GAME DEVELOPMENT & EMBEDDED SYSTEMS

C++ is one of the most widely used programming languages in **game development and embedded systems** due to its high performance, flexibility, and direct hardware interaction capabilities. It provides the perfect balance between **low-level memory management** and **high-level object-oriented programming**, making it ideal for applications that require real-time processing.

Why Use C++ for Game Development?

- **High Performance:** C++ is faster than most high-level languages, making it perfect for rendering graphics and handling physics in real-time.
- **Memory Management:** Allows developers to **optimize memory usage** and manage game resources efficiently.
- **Game Engines:** Many popular game engines, including **Unreal Engine, CryEngine, and Unity (for performance-critical tasks)**, use C++.
- **Cross-Platform Development:** Games built in C++ can be compiled for Windows, macOS, Linux, and gaming consoles like PlayStation and Xbox.

Why Use C++ for Embedded Systems?

- **Direct Hardware Access:** Allows direct interaction with microcontrollers and processors.

- **Efficiency:** Uses minimal system resources, making it ideal for resource-constrained environments.
- **Real-time Processing:** Supports **real-time computing,** essential for embedded applications such as **automotive control systems, IoT devices, and robotics**.

Understanding **game development and embedded programming** with C++ opens doors to **exciting career opportunities** in both industries, including **game development, hardware programming, and automation systems**.

CHAPTER 2: C++ FOR GAME DEVELOPMENT

Game development requires efficient programming techniques to handle **graphics rendering, physics simulation, AI, and user input processing**. C++ provides low-level control over these aspects, making it the language of choice for most game engines.

Game Development Libraries & Frameworks in C++

- **Unreal Engine:** One of the most powerful game engines, written primarily in C++.
- **SFML (Simple and Fast Multimedia Library):** Used for **2D game development** with easy graphics, audio, and networking support.
- **SDL (Simple DirectMedia Layer):** A low-level library used for handling **graphics, sound, and input devices**.
- **OpenGL & DirectX:** Used for rendering **3D graphics** and game development on different platforms.

Example: Creating a Simple Game with SFML

The following example creates a simple **window** using SFML, which is a lightweight framework for **2D game development**.

```
#include <SFML/Graphics.hpp>


int main() {

  sf::RenderWindow window(sf::VideoMode(800, 600), "Simple Game");


  sf::CircleShape player(50); // A circular player

  player.setFillColor(sf::Color::Red);

  player.setPosition(375, 275);


  while (window.isOpen()) {

    sf::Event event;

    while (window.pollEvent(event)) {

      if (event.type == sf::Event::Closed)

        window.close();

    }


    window.clear();

    window.draw(player);

    window.display();
```

```
    }


    return 0;
}
```

Explanation:

- Creates an **800x600 window** titled "Simple Game".
- Renders a **red circle** (representing the player) in the center.
- Continuously updates the screen and handles window close events.

Key Aspects of Game Development in C++

1. **Rendering Graphics:** Using SFML, OpenGL, or DirectX for visual output.
2. **Handling User Input:** Capturing keyboard and mouse events.
3. **Game Physics:** Managing movement, collision detection, and physics simulations.
4. **AI in Games:** Implementing enemy movement, pathfinding (A* Algorithm), and decision-making logic.

With these tools, C++ is used for developing both **AAA games** and **indie projects**.

CHAPTER 3: C++ FOR EMBEDDED SYSTEMS

Embedded systems are specialized computing systems designed to perform dedicated functions within **hardware devices**, such as

automotive control systems, IoT devices, medical instruments, and industrial robots. C++ is widely used in embedded programming due to its **efficiency and ability to directly interact with hardware components**.

Key Features of Embedded Systems Programming in C++

- **Memory Efficiency:** Uses minimal RAM and processing power.
- **Hardware Interfacing:** Allows communication with sensors, actuators, and other peripherals.
- **Real-time Execution:** Ensures precise timing in critical applications.
- **Portability:** Can be used across different hardware architectures (ARM, x86, AVR, etc.).

Example: Controlling an LED with C++ on an Embedded Device (Arduino)

```
#include <Arduino.h>


const int LED_PIN = 13;  // Built-in LED pin


void setup() {
  pinMode(LED_PIN, OUTPUT); // Set pin as output
}


void loop() {
  digitalWrite(LED_PIN, HIGH); // Turn LED ON
```

```
  delay(1000);              // Wait 1 second
  digitalWrite(LED_PIN, LOW);  // Turn LED OFF
  delay(1000);              // Wait 1 second
}
```

Explanation:

- The setup() function initializes the LED pin as an output.
- The loop() function turns the LED **on and off every second**.

This simple example demonstrates how **C++ is used in real-time embedded systems**.

Embedded Systems Development Platforms

1. **Arduino:** Used for IoT projects, home automation, and robotics.
2. **Raspberry Pi:** Small computer for advanced automation tasks.
3. **ARM Cortex Processors:** Used in industrial and automotive embedded systems.

CHAPTER 4: CASE STUDY – DEVELOPING A SMART HOME SYSTEM

Scenario

A company wants to develop a **Smart Home Automation System** that:

- **Uses C++ for real-time control of embedded devices.**
- **Connects sensors (temperature, motion) with a server.**

- **Has a game-like GUI interface for user interaction.**

Solution: Combining Game Development & Embedded Systems

1. **Embedded System:** Uses an Arduino microcontroller to control smart devices.
2. **Game-Like GUI:** Uses SFML in C++ to create a **dashboard for monitoring sensors**.
3. **Network Communication:** Uses **TCP/IP** sockets to send and receive data between embedded devices and the user's PC.

Example: C++ Code for Home Automation System

```
#include <iostream>

#include <thread>

#include <chrono>

using namespace std;


void controlLight(bool state) {

  if (state)

    cout << "Light Turned ON\n";

  else

    cout << "Light Turned OFF\n";

}


int main() {
```

```
    cout << "Smart Home System\n";
    while (true) {
        char command;
        cout << "Enter 'o' to turn ON light, 'f' to turn OFF, 'q' to quit: ";
        cin >> command;

        if (command == 'o')
            controlLight(true);
        else if (command == 'f')
            controlLight(false);
        else if (command == 'q')
            break;
        else
            cout << "Invalid command.\n";
    }
    return 0;
}
```

Output:

Smart Home System

Enter 'o' to turn ON light, 'f' to turn OFF, 'q' to quit: o

Light Turned ON

This system **simulates smart home control** and can be extended to interface with **real IoT devices**.

CHAPTER 5: EXERCISES

1. **Modify the SFML game to move the circle using keyboard input (WASD keys).**
2. **Implement a simple game using OpenGL that renders a rotating cube.**
3. **Develop a C++ program for a motion sensor that detects movement and turns on an LED.**
4. **Extend the home automation system to include temperature and humidity monitoring using Arduino.**

CONCLUSION

C++ is a **powerful language** that enables both **game development** and **embedded system programming**. By leveraging **graphics libraries (SFML, OpenGL)** and **hardware control tools (Arduino, Raspberry Pi)**, developers can create **real-time interactive applications**. Whether building a **high-performance game** or an **IoT-based automation system**, C++ remains a **top choice** for performance-critical applications.

Assignment Solution: Developing a Simple Database-Driven C++ Application

Objective

This assignment involves developing a **database-driven C++ application** that performs **CRUD (Create, Read, Update, Delete) operations** on a **student database** using **MySQL**. We will use **MySQL Connector for C++** to interact with the database.

Step 1: Setting Up the Environment

Requirements

To develop this application, you need:

1. **C++ Compiler (g++/MinGW/Visual Studio C++)**
2. **MySQL Server** (Installed and running)
3. **MySQL Connector/C++ Library** (For database interaction)

Installing MySQL Connector for C++

On Linux/macOS

sudo apt-get install libmysqlclient-dev

On Windows

- Download **MySQL Connector for C++** from MySQL official site.

- Extract the files and add the lib directory to your **compiler's library path**.

Step 2: Creating a MySQL Database and Table

Open MySQL Command Line and Execute:

```
CREATE DATABASE StudentDB;

USE StudentDB;


CREATE TABLE Students (
  ID INT PRIMARY KEY AUTO_INCREMENT,
  Name VARCHAR(50),
  Age INT,
  Course VARCHAR(50)
);
```

This creates a StudentDB database with a Students table to store student records.

Step 3: Writing the C++ Code to Connect with MySQL

Including Necessary Headers

```
#include <iostream>

#include <mysql/mysql.h>

using namespace std;
```

Establishing a Connection to MySQL

```
MYSQL* connectDB() {
   MYSQL* conn = mysql_init(nullptr);

   if (conn == nullptr) {
      cout << "MySQL initialization failed!\n";
      return nullptr;
   }

   if (mysql_real_connect(conn, "localhost", "root", "password",
"StudentDB", 3306, nullptr, 0)) {
      cout << "Connected to MySQL Database!\n";
   } else {
      cout << "Connection Failed: " << mysql_error(conn) << endl;
      return nullptr;
   }

   return conn;
}
```

- Initializes MySQL connection.
- Connects to StudentDB using **username: root, password: password** (Change as per your MySQL setup).

STEP 4: IMPLEMENTING CRUD OPERATIONS

1. Function to Insert a Student Record

```
void addStudent(MYSQL* conn) {
   string name, course;
   int age;

   cout << "Enter Student Name: ";
   cin.ignore();
   getline(cin, name);
   cout << "Enter Age: ";
   cin >> age;
   cin.ignore();
   cout << "Enter Course: ";
   getline(cin, course);

   string query = "INSERT INTO Students (Name, Age, Course)
VALUES ('" + name + "', " + to_string(age) + ", '" + course + "');";

   if (mysql_query(conn, query.c_str()) == 0) {
      cout << "Student record added successfully!\n";
```

```
    } else {
        cout << "Error: " << mysql_error(conn) << endl;
    }
}
```

- Takes student details from the user.
- Constructs an SQL INSERT query and executes it using mysql_query().

---

2. Function to Retrieve All Student Records

```
void viewStudents(MYSQL* conn) {
    if (mysql_query(conn, "SELECT * FROM Students") == 0) {
        MYSQL_RES* res = mysql_store_result(conn);
        MYSQL_ROW row;

        cout << "\nID\tName\tAge\tCourse\n";
        cout << "-------------------------------\n";

        while ((row = mysql_fetch_row(res))) {
            cout << row[0] << "\t" << row[1] << "\t" << row[2] << "\t" << row[3] << endl;
        }
```

```
        mysql_free_result(res);
    } else {
        cout << "Error: " << mysql_error(conn) << endl;
    }
}
```

- Executes a SELECT query to retrieve student records.
- Displays the results in tabular format.

---

3. Function to Update a Student Record

```
void updateStudent(MYSQL* conn) {
    int id, age;
    string name, course;

    cout << "Enter Student ID to update: ";
    cin >> id;
    cin.ignore();
    cout << "Enter New Name: ";
    getline(cin, name);
    cout << "Enter New Age: ";
    cin >> age;
    cin.ignore();
```

```
    cout << "Enter New Course: ";
    getline(cin, course);

    string query = "UPDATE Students SET Name='" + name + "', Age="
+ to_string(age) + ", Course='" + course + "' WHERE ID=" +
to_string(id) + ";";

    if (mysql_query(conn, query.c_str()) == 0) {
        cout << "Student record updated successfully!\n";
    } else {
        cout << "Error: " << mysql_error(conn) << endl;
    }
}
```

- Updates student details based on **ID**.

---

4. Function to Delete a Student Record

```
void deleteStudent(MYSQL* conn) {
    int id;
    cout << "Enter Student ID to delete: ";
    cin >> id;
```

```
    string query = "DELETE FROM Students WHERE ID=" +
to_string(id) + ";";


    if (mysql_query(conn, query.c_str()) == 0) {
        cout << "Student record deleted successfully!\n";
    } else {
        cout << "Error: " << mysql_error(conn) << endl;
    }
}
```

- Deletes a record based on the provided **Student ID**.

STEP 5: CREATING THE MAIN MENU

```
int main() {
    MYSQL* conn = connectDB();
    if (conn == nullptr) return 1;

    int choice;
    while (true) {
        cout << "\nStudent Management System\n";
        cout << "-------------------------\n";
        cout << "1. Add Student\n2. View Students\n3. Update
Student\n4. Delete Student\n5. Exit\n";
```

```
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
      case 1: addStudent(conn); break;
      case 2: viewStudents(conn); break;
      case 3: updateStudent(conn); break;
      case 4: deleteStudent(conn); break;
      case 5: mysql_close(conn); cout << "Exiting...\n"; return 0;
      default: cout << "Invalid choice, try again.\n";
    }
  }
}
```

- Displays a **menu-driven system**.
- Calls appropriate functions based on user input.

STEP 6: COMPILING AND RUNNING THE PROGRAM

Compiling the Program (Linux/macOS)

```
g++ student_management.cpp -o student_management -lmysqlclient
```

```
./student_management
```

Compiling the Program (Windows)

g++ student_management.cpp -o student_management.exe -I"C:\MySQL\include" -L"C:\MySQL\lib" -lmysqlclient

student_management.exe

Ensure that MySQL **server is running** before executing the program.

STEP 7: EXPECTED OUTPUT

Student Management System

---

1. Add Student

2. View Students

3. Update Student

4. Delete Student

5. Exit

Enter your choice: 1

Enter Student Name: Alice

Enter Age: 21

Enter Course: Computer Science

Student record added successfully.

After adding multiple students, viewing records:

```
ID  Name  Age  Course
--------------------------------
1   Alice 21   Computer Science
2   Bob   22   Mathematics
```

CONCLUSION

In this assignment, we built a **simple database-driven C++ application** that performs **CRUD operations on a MySQL database**. By using **MySQL Connector for C++**, we efficiently interacted with the database, enabling **data persistence and management**. This system can be expanded to support **more complex features like authentication, analytics, and cloud integration**.

Assignment Solution: Mini Project Integrating STL, Multi-threading, and File Handling in C++

Objective

This mini project demonstrates how to integrate **STL (Standard Template Library), multi-threading, and file handling** in a C++ program. We will build a **multi-threaded Student Management System** that:

✓ Uses **STL containers (vector, map, list)** for efficient data storage.

✓ Implements **multi-threading** for concurrent operations.

✓ Utilizes **file handling** for **saving and retrieving** student data.

Step 1: Setting Up the Environment

Requirements

To build this project, you need:

- A **C++ compiler (g++/MinGW/Visual Studio C++)**
- Knowledge of **STL (vector, map, list), multi-threading, and file handling**

Project Features

1. **Add Student** – Adds a student record and saves it to a file.
2. **View Students** – Reads student records from the file and displays them.
3. **Search Student** – Searches for a student by ID using **STL map**.

4. **Multi-threading for File Handling** – Reads and writes student data using separate threads.

Step 2: Designing the Data Structure

```
#include <iostream>
#include <vector>
#include <map>
#include <list>
#include <fstream>
#include <thread>
#include <mutex>
#include <sstream>
using namespace std;
```

- Includes **STL containers** (vector, map, list).
- Uses **fstream** for file handling.
- Implements **multi-threading** with thread and mutex.

```
struct Student {
    int id;
    string name;
    int age;
    string course;
```

```
    void display() {
        cout << "ID: " << id << ", Name: " << name
             << ", Age: " << age << ", Course: " << course << endl;
    }
};
```

- Defines a **Student structure** to store **ID, Name, Age, and Course**.
- Includes a method to **display student information**.

Step 3: Implementing File Handling

A **mutex** is used to prevent **race conditions** when writing to or reading from a file.

```
mutex fileMutex;
const string filename = "students.txt";
```

Function to Save Student Records to a File

```
void saveToFile(const vector<Student>& students) {
    lock_guard<mutex> lock(fileMutex);
    ofstream file(filename, ios::trunc);

    for (const auto& student : students) {
        file << student.id << "," << student.name << ","
             << student.age << "," << student.course << "\n";
```

```
    }
    file.close();
}
```

- **lock_guard<mutex>** ensures thread-safe file access.
- Overwrites the file with updated student records.

Function to Load Students from File

```
vector<Student> loadFromFile() {
    lock_guard<mutex> lock(fileMutex);
    vector<Student> students;
    ifstream file(filename);
    string line;

    while (getline(file, line)) {
        stringstream ss(line);
        Student student;
        string temp;

        getline(ss, temp, ','); student.id = stoi(temp);
        getline(ss, student.name, ',');
        getline(ss, temp, ','); student.age = stoi(temp);
```

```
    getline(ss, student.course, ',');


    students.push_back(student);
  }
  file.close();
  return students;
}
```

- Reads student records from **file** and loads them into a **vector**.
- Uses stringstream to **parse CSV data**.

STEP 4: IMPLEMENTING STL & MULTI-THREADING

Function to Add a Student Record (Uses vector)

```
void addStudent(vector<Student>& students, map<int, Student>& studentMap) {
  Student student;
  cout << "Enter Student ID: ";
  cin >> student.id;
  cin.ignore();
  cout << "Enter Student Name: ";
  getline(cin, student.name);
  cout << "Enter Age: ";
```

```
    cin >> student.age;
    cin.ignore();
    cout << "Enter Course: ";
    getline(cin, student.course);

    students.push_back(student);
    studentMap[student.id] = student;

    thread saveThread(saveToFile, students);
    saveThread.detach(); // Runs in background

    cout << "Student record added successfully!\n";
}
```

- **Uses vector** to store student records in memory.
- **Uses map<int, Student>** for quick student lookups.
- **Runs saveToFile() in a separate thread** to save data asynchronously.

Function to Display All Students (Uses list)

```
void viewStudents(const vector<Student>& students) {
    if (students.empty()) {
```

```
        cout << "No student records available.\n";
        return;
    }

    list<Student> studentList(students.begin(), students.end());
    for (const auto& student : studentList) {
        student.display();
    }
}
```

- **Converts vector to list** for efficient traversal.
- **Displays all student records**.

Function to Search a Student by ID (Uses map)

```
void searchStudent(const map<int, Student>& studentMap) {
    int id;
    cout << "Enter Student ID to search: ";
    cin >> id;

    auto it = studentMap.find(id);
    if (it != studentMap.end()) {
        it->second.display();
```

```
    } else {
        cout << "Student not found.\n";
    }
}
```

- **Uses map<int, Student>** for fast lookup (O(1) time complexity).

Step 5: Creating the Main Menu

```
int main() {
    vector<Student> students = loadFromFile();
    map<int, Student> studentMap;

    for (const auto& student : students) {
        studentMap[student.id] = student;
    }

    int choice;
    while (true) {
        cout << "\nStudent Management System\n";
        cout << "1. Add Student\n2. View Students\n3. Search Student\n4. Exit\n";
        cout << "Enter choice: ";
```

```
        cin >> choice;

        switch (choice) {
            case 1: addStudent(students, studentMap); break;
            case 2: viewStudents(students); break;
            case 3: searchStudent(studentMap); break;
            case 4: cout << "Exiting...\n"; return 0;
            default: cout << "Invalid choice.\n";
        }
    }
}
```

- **Loads student records from file** when the program starts.
- Uses a **menu-driven system** for user interaction.

Step 6: Compiling and Running the Program

Compile the Program (Linux/macOS)

```
g++ student_manager.cpp -o student_manager -pthread
./student_manager
```

Compile the Program (Windows)

```
g++ student_manager.cpp -o student_manager.exe -lpthread
student_manager.exe
```

Make sure students.txt exists or is created when **adding students**.

STEP 7: EXPECTED OUTPUT

Student Management System

1. Add Student

2. View Students

3. Search Student

4. Exit

Enter choice: 1

Enter Student ID: 101

Enter Student Name: Alice

Enter Age: 20

Enter Course: Computer Science

Student record added successfully!

After adding multiple students, viewing records:

ID: 101, Name: Alice, Age: 20, Course: Computer Science

ID: 102, Name: Bob, Age: 22, Course: Mathematics

Searching for a student:

Enter Student ID to search: 101

ID: 101, Name: Alice, Age: 20, Course: Computer Science

CONCLUSION

This **mini project** successfully integrates **STL, multi-threading, and file handling** into a **Student Management System**. By leveraging:

✓ **STL Containers (vector, list, map)** for efficient data storage.

✓ **Multi-threading** for file operations to improve performance.

✓ **File handling** for persistent data storage.