Overview:

- **Concurrency** allows a program to execute multiple tasks simultaneously, improving performance on multi-core processors.
- C++ provides concurrency support through the **Threads** library, introduced in **C++11** and expanded in later versions (C++14, C++17, C++20/23).

Key Features:

- **Threads:** Create and manage threads.
- **Mutexes and Locks:** Prevent race conditions by synchronizing access to shared data.
- **Condition Variables:** Allow threads to wait for certain conditions.
- **Futures and Promises:** Handle asynchronous tasks and return values.

Thread Management

Creating Threads:

- Threads are created using the std::thread class.
- A thread can execute a function, a member function, or a lambda function.

Thread Lifecycle:

- **Joinable:** A thread is joinable if it is running or has finished but not yet joined.
- **Join:** Wait for the thread to finish execution.
- **Detach:** Allow the thread to run independently of the main thread.

Sample Code

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

void helloFunction() {
    cout << "Hello from function..." << endl;
}

int main() {
    cout << "Hello World from main()" << endl;
    thread t1(helloFunction); // Create a thread that runs helloFunction
    t1.join(); // Wait for the thread to finish
    cout << "Main program ends..." << endl;
    return 0;
}
```

Sample Code

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

class HelloObject {
public:
    void objectFunction() {
        cout << "Hello from object function" << endl;
    }
};

int main() {
    cout << "Hello World from main()" << endl;
    HelloObject ho;
    thread t2(&HelloObject::objectFunction, &ho); // Create a thread that runs a member functio
n
    t2.join(); // Wait for the thread to finish
    cout << "Main program ends..." << endl;
    return 0;
}
```

Sample Code

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

void helloFunction(int n, string str) {
    for (int i = 0; i < n; i++) {
        cout << "Hello from function... " << str << endl;
    }
}

int main() {
    cout << "Hello World from main()" << endl;
    thread t1(helloFunction, 3, "FEUTECH"); // Create a thread with parameters
    t1.join(); // Wait for the thread to finish
    cout << "Main program ends..." << endl;
    return 0;
}
```

Mutexes and Thread Synchronization

Overview:

- **Mutexes** are used to prevent race conditions by ensuring that only one thread can access shared data at a time.
- In C++, mutexes are implemented using the std::mutex class.

Sample Code

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

mutex mtx; // Mutex to protect shared data
int sharedData = 0;

void incrementData() {
    mtx.lock(); // Lock the mutex
    sharedData++;
    cout << "Shared Data: " << sharedData << endl;
    mtx.unlock(); // Unlock the mutex
}

int main() {
    thread t1(incrementData);
    thread t2(incrementData);
    t1.join();
    t2.join();
    return 0;
}
```

Deadlock Avoidance

lock(mtx1, mtx2);: Locks both mutexes simultaneously to avoid deadlock.

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

mutex mtx1, mtx2; // Two mutexes for synchronization

void task1(int threadId) {
    cout << "Thread " << threadId << " is attempting to lock mtx1 and mtx2." << endl;
    lock(mtx1, mtx2); // Lock both mutexes simultaneously (deadlock-safe)
    lock_guard<mutex> lock1(mtx1, adopt_lock); // Adopt the lock for mtx1
    lock_guard<mutex> lock2(mtx2, adopt_lock); // Adopt the lock for mtx2
    cout << "Thread " << threadId << " has locked mtx1 and mtx2. Task 1 is running." << endl;
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work
    cout << "Thread " << threadId << " is releasing mtx1 and mtx2." << endl;
}

void task2(int threadId) {
    cout << "Thread " << threadId << " is attempting to lock mtx1 and mtx2." << endl;
    lock(mtx1, mtx2); // Lock both mutexes simultaneously (deadlock-safe)
    lock_guard<mutex> lock1(mtx1, adopt_lock); // Adopt the lock for mtx1
    lock_guard<mutex> lock2(mtx2, adopt_lock); // Adopt the lock for mtx2
    cout << "Thread " << threadId << " has locked mtx1 and mtx2. Task 2 is running." << endl;
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work
    cout << "Thread " << threadId << " is releasing mtx1 and mtx2." << endl;
}

int main() {
    cout << "Starting threads..." << endl;
    thread t1(task1, 1); // Thread 1 with ID 1
    thread t2(task2, 2); // Thread 2 with ID 2

    t1.join();
    t2.join();

    cout << "Both threads have finished." << endl;
    return 0;
}
```

Use std::scoped_lock (C++17)

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

mutex mtx1, mtx2; // Two mutexes for synchronization

void task1(int threadId) {
    cout << "Thread " << threadId << " is attempting to lock mtx1 and mtx2." << endl;
    scoped_lock lock(mtx1, mtx2); // Automatically locks both mutexes (deadlock-safe)
    cout << "Thread " << threadId << " has locked mtx1 and mtx2. Task 1 is running." << endl;
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work
    cout << "Thread " << threadId << " is releasing mtx1 and mtx2." << endl;
}

void task2(int threadId) {
    cout << "Thread " << threadId << " is attempting to lock mtx1 and mtx2." << endl;
    scoped_lock lock(mtx1, mtx2); // Automatically locks both mutexes (deadlock-safe)
    cout << "Thread " << threadId << " has locked mtx1 and mtx2. Task 2 is running." << endl;
    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate work
    cout << "Thread " << threadId << " is releasing mtx1 and mtx2." << endl;
}

int main() {
    cout << "Starting threads..." << endl;
    thread t1(task1, 1); // Thread 1 with ID 1
    thread t2(task2, 2); // Thread 2 with ID 2

    t1.join();
    t2.join();

    cout << "Both threads have finished." << endl;
    return 0;
}
```

WITH DEADLOCK

1. **Circular Wait**: Thread 1 holds Lock A and waits for Lock B, while Thread 2 holds Lock B and waits for Lock A.
2. **No Preemption**: Locks cannot be forcibly taken away from a thread.
3. **Hold and Wait**: Threads hold locks while waiting for additional locks.
4. **Mutual Exclusion**: Only one thread can hold a lock at a time.

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

mutex mtx1, mtx2; // Two mutexes for synchronization

void task1() {
    cout << "Thread 1 is attempting to lock mtx1." << endl;
    mtx1.lock(); // Thread 1 locks mtx1
    cout << "Thread 1 has locked mtx1." << endl;

    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate some work

    cout << "Thread 1 is attempting to lock mtx2." << endl;
    mtx2.lock(); // Thread 1 waits for mtx2 (which is held by Thread 2)
    cout << "Thread 1 has locked mtx2." << endl;

    // Critical section
    cout << "Thread 1 is running." << endl;

    mtx2.unlock();
    mtx1.unlock();
}

void task2() {
    cout << "Thread 2 is attempting to lock mtx2." << endl;
    mtx2.lock(); // Thread 2 locks mtx2
    cout << "Thread 2 has locked mtx2." << endl;

    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate some work

    cout << "Thread 2 is attempting to lock mtx1." << endl;
    mtx1.lock(); // Thread 2 waits for mtx1 (which is held by Thread 1)
    cout << "Thread 2 has locked mtx1." << endl;

    // Critical section
    cout << "Thread 2 is running." << endl;

    mtx1.unlock();
    mtx2.unlock();
}

int main() {
    cout << "Starting threads..." << endl;
    thread t1(task1); // Thread 1
    thread t2(task2); // Thread 2

    t1.join();
    t2.join();

    cout << "Both threads have finished." << endl;
    return 0;
}
```

FIX

```
void task1() {
    cout << "Thread 1 is attempting to lock mtx1." << endl;
    mtx1.lock(); // Thread 1 locks mtx1
    cout << "Thread 1 has locked mtx1." << endl;

    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate some work

    cout << "Thread 1 is attempting to lock mtx2." << endl;
    mtx2.lock(); // Thread 1 locks mtx2
    cout << "Thread 1 has locked mtx2." << endl;

    // Critical section
    cout << "Thread 1 is running." << endl;

    mtx2.unlock();
    mtx1.unlock();
}

void task2() {
    cout << "Thread 2 is attempting to lock mtx1." << endl;
    mtx1.lock(); // Thread 2 locks mtx1
    cout << "Thread 2 has locked mtx1." << endl;

    this_thread::sleep_for(chrono::milliseconds(100)); // Simulate some work

    cout << "Thread 2 is attempting to lock mtx2." << endl;
    mtx2.lock(); // Thread 2 locks mtx2
    cout << "Thread 2 has locked mtx2." << endl;

    // Critical section
    cout << "Thread 2 is running." << endl;

    mtx2.unlock();
    mtx1.unlock();
}
```

1. **Deadlock occurs** when threads wait indefinitely for each other to release locks.
2. **To avoid deadlock**:
   - Always lock mutexes in the same order.
   - Use higher-level constructs like std::lock or std::scoped_lock to lock multiple mutexes safely.
3. **Debugging deadlocks** can be challenging, so it's important to design your code to avoid them from the start.

Data Sharing and Race Conditions

Race Conditions:

- A **race condition** occurs when two or more threads access shared data simultaneously, leading to unpredictable results.
- To prevent race conditions, use **mutexes** to synchronize access to shared data.

Sample Code

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

int sharedData = 0; // Shared data between threads
mutex mtx; // Mutex to protect shared data

void incrementData(int threadId) {
    for (int i = 0; i < 100000; i++) {
        mtx.lock(); // Lock the mutex before accessing shared data
        sharedData++; // Increment shared data
        cout << "Thread " << threadId << " incremented sharedData to: " << sharedData << endl;
        mtx.unlock(); // Unlock the mutex after accessing shared data
    }
}

int main() {
    // Create two threads
    thread t1(incrementData, 1); // Thread 1
    thread t2(incrementData, 2); // Thread 2

    // Wait for both threads to finish
    t1.join();
    t2.join();

    // Print the final value of sharedData
    cout << "Final Shared Data: " << sharedData << endl;
    return 0;
}
```

Sample Code

Using join() to Wait for Thread Completion

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

void threadFunction() {
    cout << "Thread is running..." << endl;
    this_thread::sleep_for(chrono::seconds(2)); // Simulate work by sleeping for 2 seconds
    cout << "Thread is finishing..." << endl;
}

int main() {
    cout << "Main thread starts..." << endl;
    thread t(threadFunction); // Create a thread that runs threadFunction
    t.join(); // Wait for the thread to finish
    cout << "Main thread ends..." << endl;
    return 0;
}
```

Sample Code

Using detach() to Run a Thread Independently

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

void threadFunction() {
    cout << "Thread is running..." << endl;
    this_thread::sleep_for(chrono::seconds(2)); // Simulate work by sleeping for 2 seconds
    cout << "Thread is finishing..." << endl;
}

int main() {
    cout << "Main thread starts..." << endl;
    thread t(threadFunction); // Create a thread that runs threadFunction
    t.detach(); // Detach the thread, allowing it to run independently
    cout << "Main thread ends..." << endl;
    return 0;
}
```

Sample Code

Using get_id() and sleep_for() to Manage Thread Execution

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

void threadFunction() {
    cout << "Thread ID: " << this_thread::get_id() << " is running..." << endl;
    this_thread::sleep_for(chrono::seconds(1)); // Simulate work by sleeping for 1 second
    cout << "Thread ID: " << this_thread::get_id() << " is finishing..." << endl;
}

int main() {
    cout << "Main thread starts..." << endl;
    thread t1(threadFunction); // Create the first thread
    thread t2(threadFunction); // Create the second thread

    cout << "Thread 1 ID: " << t1.get_id() << endl;
    cout << "Thread 2 ID: " << t2.get_id() << endl;

    t1.join(); // Wait for the first thread to finish
    t2.join(); // Wait for the second thread to finish

    cout << "Main thread ends..." << endl;
    return 0;
}
```

Sample Code

Download Manager (Multiple File Downloads)

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

void downloadFile(string fileName, int downloadTime) {
    cout << "Downloading " << fileName << "..." << endl;
    this_thread::sleep_for(chrono::seconds(downloadTime)); // Simulate download time
    cout << fileName << " downloaded!" << endl;
}

int main() {
    // Arrays to store file names and download times
    string files[] = {"file1.zip", "file2.mp4", "file3.pdf"};
    int downloadTimes[] = {3, 5, 2}; // Simulated download times for each file
    const int numFiles = 3; // Number of files

    // Array to store thread objects
    thread threads[numFiles];

    // Start a thread for each file download
    for (int i = 0; i < numFiles; i++) {
        threads[i] = thread(downloadFile, files[i], downloadTimes[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < numFiles; i++) {
        threads[i].join();
    }

    cout << "All downloads completed!" << endl;
    return 0;
}
```

Bank Account Transactions (Deposit and Withdraw)

Scenario

A bank system allows multiple users to deposit and withdraw money from their accounts simultaneously. Threads are used to handle concurrent transactions.

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

mutex mtx; // Mutex to protect shared data
int balance = 1000; // Initial account balance

void deposit(int amount, int threadId) {
    cout << "Thread " << threadId << " is attempting to deposit " << amount << "." << endl;
    mtx.lock(); // Lock the mutex before accessing shared data
    cout << "Thread " << threadId << " has locked the mutex. Depositing " << amount << "." << endl;
    balance += amount;
    cout << "Thread " << threadId << " deposited " << amount << ". New balance: " << balance << endl;
    mtx.unlock(); // Unlock the mutex after accessing shared data
    cout << "Thread " << threadId << " has released the mutex." << endl;
}

void withdraw(int amount, int threadId) {
    cout << "Thread " << threadId << " is attempting to withdraw " << amount << "." << endl;
    mtx.lock(); // Lock the mutex before accessing shared data
    cout << "Thread " << threadId << " has locked the mutex. Checking balance for withdrawal." << endl;
    if (balance >= amount) {
        balance -= amount;
        cout << "Thread " << threadId << " withdrawn " << amount << ". New balance: " << balance << endl;
    } else {
        cout << "Thread " << threadId << " failed to withdraw " << amount << ". Insufficient balance." << endl;
    }
    mtx.unlock(); // Unlock the mutex after accessing shared data
    cout << "Thread " << threadId << " has released the mutex." << endl;
}

int main() {
    cout << "Initial balance: " << balance << endl;
    cout << "Starting threads..." << endl;

    thread t1(deposit, 500, 1); // Deposit 500 (Thread 1)
    thread t2(withdraw, 200, 2); // Withdraw 200 (Thread 2)
    thread t3(withdraw, 1500, 3); // Attempt to withdraw 1500 (Thread 3)

    t1.join();
    t2.join();
    t3.join();

    cout << "Final balance: " << balance << endl;
    return 0;
}
```

Traffic Light System

Scenario:

A traffic light system controls multiple intersections. Each intersection is managed by a separate thread, simulating concurrent traffic light operations.

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

void trafficLight(string intersection, int greenTime, int redTime) {
    while (true) {
        cout << intersection << ": Green light for " << greenTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(greenTime)); // Simulate green light
        cout << intersection << ": Red light for " << redTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(redTime)); // Simulate red light
    }
}

int main() {
    thread t1(trafficLight, "Intersection 1", 5, 3); // Intersection 1: Green for 5s, Red for 3
s
    thread t2(trafficLight, "Intersection 2", 4, 2); // Intersection 2: Green for 4s, Red for 2
s

    t1.join();
    t2.join();

    return 0;
}
```

V2

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

void trafficLight(string intersection, int greenTime, int redTime) {
    while (true) {
        // Green light phase
        cout << intersection << ": Green light ON for " << greenTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(greenTime)); // Simulate green light duration

        // Transition to red light
        cout << intersection << ": Yellow light ON for 2 seconds (transition)." << endl;
        this_thread::sleep_for(chrono::seconds(2)); // Simulate yellow light (transition)

        // Red light phase
        cout << intersection << ": Red light ON for " << redTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(redTime)); // Simulate red light duration

        // Transition back to green light
        cout << intersection << ": Yellow light ON for 2 seconds (transition)." << endl;
        this_thread::sleep_for(chrono::seconds(2)); // Simulate yellow light (transition)
    }
}

int main() {
    cout << "Starting traffic light simulation..." << endl;

    // Create threads for two intersections
    thread t1(trafficLight, "Intersection 1", 5, 3); // Intersection 1: Green for 5s, Red for 3
s
    thread t2(trafficLight, "Intersection 2", 4, 2); // Intersection 2: Green for 4s, Red for 2
s

    // Wait for threads to finish (they won't, since the simulation runs indefinitely)
    t1.join();
    t2.join();

    return 0;
}
```

V3 With Timestamp

```cpp
#include <chrono>
#include <iomanip>
using namespace std;

// Function to get the current time as a string
string getCurrentTime() {
    auto now = chrono::system_clock::now();
    auto now_time_t = chrono::system_clock::to_time_t(now);
    stringstream ss;
    ss << put_time(localtime(&now_time_t), "%H:%M:%S");
    return ss.str();
}

void trafficLight(string intersection, int greenTime, int redTime) {
    while (true) {
        // Green light phase
        cout << "[" << getCurrentTime() << "] " << intersection << ": Green light ON for " << greenTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(greenTime)); // Simulate green light duration

        // Transition to red light
        cout << "[" << getCurrentTime() << "] " << intersection << ": Yellow light ON for 2 seconds (transition)." << endl;
        this_thread::sleep_for(chrono::seconds(2)); // Simulate yellow light (transition)

        // Red light phase
        cout << "[" << getCurrentTime() << "] " << intersection << ": Red light ON for " << redTime << " seconds." << endl;
        this_thread::sleep_for(chrono::seconds(redTime)); // Simulate red light duration

        // Transition back to green light
        cout << "[" << getCurrentTime() << "] " << intersection << ": Yellow light ON for 2 seconds (transition)." << endl;
        this_thread::sleep_for(chrono::seconds(2)); // Simulate yellow light (transition)
    }
}

int main() {
    cout << "Starting traffic light simulation..." << endl;

    // Create threads for two intersections
    thread t1(trafficLight, "Intersection 1", 5, 3); // Intersection 1: Green for 5s, Red for 3s
    thread t2(trafficLight, "Intersection 2", 4, 2); // Intersection 2: Green for 4s, Red for 2s

    // Wait for threads to finish (they won't, since the simulation runs indefinitely)
    t1.join();
    t2.join();

    return 0;
}
```