Threads ::

#include <iostream>

#include <thread>

void sayHello() {

    std::cout << "Hello from a thread!" << std::endl;

}

int main() {

    std::thread t(sayHello);  // Create a thread that runs the sayHello function

    t.join();                 // Wait for the thread to finish

    return 0;

}

Thread Joining ::

#include <iostream>

#include <thread>

void task()

{

    std::cout << "Task is running." << std::endl;

}

int main()

{

    std::thread t(task);

    std::cout << "Waiting for thread to finish it's task." << std::endl;

    t.join(); // wait for thread to complete

    std::cout << "Thread has finished execution." << std::endl;

    return 0;

}

Detach Thread ::

#include <iostream>

#include <thread>

void task()

{

    std::cout << "Task is going on.\n";

}

int main()

{

    std::thread t(task);

    t.detach(); // Detach the thread

    std::cout << "Thread is detached.\n";

    return 0;

}

/\*

- Program may exit before the detached thread finishes.

$ g++ 3\_detach\_thread.cpp

$ ./a.exe

Thread is detached.

$ g++ 3\_detach\_thread.cpp

$ ./a.exe

Thread is detached.

$ g++ 3\_detach\_thread.cpp

$ ./a.exe

Thread is detached.

Task is going on.

- Detached threads should be used with caution because if the main program exits while detached threads are

still going on, the detached threads may not be able to finish work.

- Joined threads ensures that a thread has completed it's task before the program proceeds, provides a way

to manage thread lifetimes explicitly.

\*/

Arguments on thread ::

#include <iostream>

#include <thread>

/\*

- Pass by value ::

\*/

void printNumbers(int x)

{

    std::cout << "Number : " << x << std::endl;

}

/\*

- Pass by reference ::

\*/

void increment(int& x)

{

    x++;

}

int main()

{

    std::thread t(printNumbers, 10); // Pass 10 to the thread

    t.join(); // wait for thread to finish

    int num = 10;

    std::thread t\_inc(increment, std::ref(num)); // Pass num by reference

    t\_inc.join();

    std::cout << "Increment number : " << num << "\n";

    return 0;

}

Lambda functions with threads ::

#include <iostream>

#include <thread>

int main()

{

    int value = 5;

    std::thread t(

        [&value]{

            value \*= 2;

            std::cout << "value in thread : " << value << std::endl;

        }

    );

    std::cout << "value in main thread before thread completes : " << value << std::endl;

    t.join();

    std::cout << "value in main thread after thread completes : " << value << std::endl;

    return 0;

}

Race Conditions ::

#include <iostream>

#include <thread>

/\*

- When multiple threads access shared data, there is a risk of a RACE CONDITION,

where the outcome depends on the timing of threads.

- To prevent race conditions, we can use MUTEXES to synchronize access to shared data.

\*/

int count = 0;

void incrementCount()

{

    for (int i=0; i<100000; i++){

        count++;

    }

}

int main()

{

    std::thread t1(incrementCount);

    std::thread t2(incrementCount);

    t1.join();

    t2.join();

    std::cout << "Final Count Value : " << count << "\n";

    /\*

    - In this, both threads are incrementing counter simulatneously without synchronizing.

    \*/

    return 0;

}

Fix Race Conditions ::

#include <iostream>

#include <thread>

#include <mutex>

int count = 0;

std::mutex mtx;

void incrementCount()

{

    for (int i=0; i<100000; i++){

        std::lock\_guard<std::mutex> lock(mtx);

        count++;

    }

}

int main()

{

    std::thread t1(incrementCount);

    std::thread t2(incrementCount);

    t1.join();

    t2.join();

    std::cout << "Final Count Value : " << count << "\n";

    return 0;

}

Unique Lock ::

#include <iostream>

#include <thread>

#include <mutex>

/\*

- lock\_guard provides a simple, RAII style mechanism for acquiring and releasing a mutex.

When a lock\_guard object is created, it locks the mutex, and when it goes out of scope, it automatically release the mutex.

- unique\_lock provides more flexibilitythan lock\_guard. It can be used for deferred locking, timed locking, and unlocking before

end of its scope. It gives more control over the locking mechanism.

\*/

std::mutex mtx;

void printMessage(const std::string& message)

{

    std::unique\_lock<std::mutex> lock(mtx);

    std::cout << message << std::endl;

    lock.unlock();

}

int main()

{

    std::thread t1(printMessage, "Hello from thread 1");

    std::thread t2(printMessage, "Hello from thread 2");

    t1.join();

    t2.join();

    return 0;

}

Call once and once flag ::

#include <iostream>

#include <thread>

#include <mutex>

/\*

- Sometimes, you want to ensure that a particular piece of code is only executed once, regardless of how many

threads attempt to execute it.

- Initialize function is only executed once.

\*/

std::once\_flag initFlag;

void initialize()

{

    std::cout << "Initialization code executed once." << std::endl;

}

void runTask()

{

    std::call\_once(initFlag, initialize);

    std::cout << "Thread running." << std::endl;

}

int main()

{

    std::thread t1(runTask);

    std::thread t2(runTask);

    std::thread t3(runTask);

    t1.join();

    t2.join();

    t3.join();

    return 0;

}

Condition Variable ::

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

std::mutex mtx;

std::condition\_variable cv;

bool ready = false;

void printMessage(int id)

{

    std::unique\_lock<std::mutex> lock(mtx);

    cv.wait(lock, []{return ready;}); // Wait "ready" is true

    std::cout << "Thread " << id << " is running.\n";

}

void setReady()

{

    std::this\_thread::sleep\_for(std::chrono::seconds(1)); // Simulate work

    std::lock\_guard<std::mutex> lock(mtx);

    ready = true;

    cv.notify\_all(); // Notify all waiting threads

}

int main()

{

    std::thread t1(printMessage, 1);

    std::thread t2(printMessage, 2);

    std::thread t3(printMessage, 3);

    std::thread signalThread(setReady);

    t1.join();

    t2.join();

    t3.join();

    signalThread.join();

    return 0;

}

/\*

- condition\_variable allows threads to wait until they are notified by another thread that some condition is true.

- cv.wait() waits until the ready flag is set to true. This function will automatically unlocks the mutex an dputs threads in waiting state.

- cv.notify\_all() wakes up all waiting threads once the condition is met.

\*/

Future and Promises ::

#include <iostream>

#include <thread>

#include <mutex>

#include <future>

/\*

- Future and promises provides a mechanism for transferring data or exceptions between threads,

with the future acting as a handle to retrieve the result of an asynchronous operation.

\*/

void computeSquare(std::promise<int> p, int x)

{

    int result = x\*x;

    p.set\_value(result); // set the result for the promise

}

int main()

{

    std::promise<int> p;

    std::future<int> f = p.get\_future(); // Get the future associated with the promise

    std::thread t(computeSquare, std::move(p), 5);

    int result = f.get(); // wait for the result

    std::cout << "Square of 5 is : " << result << std::endl;

    t.join();

    return 0;

}

/\*

- promise<int> A promise that will eventually hold an integer value.

- future<int> A future that can be used to retrieve the value set by the promise.

- f.get() blocks until the result is ready, then returns the value.

\*/

Async ::

#include <iostream>

#include <thread>

#include <mutex>

#include <future>

/\*

- async is a high level API for running tasks asynchronously.

- It returns std::future object that you can use to retrieve the result.

\*/

int computedSquare(int x)

{

    return x\*x;

}

int main()

{

    std::future<int> f = std::async(std::launch::async, computedSquare, 5);

    int result = f.get();

    std::cout << "Square of 5 is : " << result << std::endl;

    return 0;

}

/\*

- async launches a task asynchronously in a seperate thread.

- f.get() waits for the task to complete and returns the result.

\*/

Thread Pool ::  
#include <iostream>

#include <thread>

#include <mutex>

#include <future>

#include <vector>

#include <queue>

#include <functional>

#include <condition\_variable>

#include <atomic>

/\*

- Creating and destroying threads frequently can be expensive. Thread pools reduce this overhead by reusing reusing threads.

- You can control number of threads in the pool, preventing the system from being overwhelmed by too many concurrent threads.

- Thread pools help you manage how tasks are executed in parallel, which is essential for scalable, high-performance applications.

- Key Components :: Worker Threads, Task Queue, Synchronization mechanisms.

--- Worker threads :: These threads execute tasks, they wait for the task to be added to a task queue.

--- Task Queue :: where tasks are stored until a worker thread is available to execute them.

--- Sync mechanism :: Mutexes and condition variables are used to ensure that access to task queue is thread-safe and to notify worker

    threads when tasks are available.

\*/

class ThreadPool {

    public:

        ThreadPool(size\_t numThreads);

        ~ThreadPool();

        void enqueueTask(std::function<void()> task);

    private:

        std::vector<std::thread> workers; // Threads in the pool

        std::queue<std::function<void()>> tasks; // Task queue

        std::mutex queueMutex; // Mutex for task queue

        std::condition\_variable condition; // Condition variable to notify worker threads

        std::atomic<bool> stop; // Flag to stop the pool

        void workerThread();

};

ThreadPool::ThreadPool(size\_t numThreads) : stop(false) {

    for (size\_t i=0; i < numThreads; i++) {

        workers.emplace\_back([this] {

            this->workerThread();

        });

    }

}

ThreadPool::~ThreadPool() {

    stop = true;

    condition.notify\_all();

    for (std::thread &worker : workers) {

        worker.join();

    }

}

void ThreadPool::enqueueTask(std::function<void()> task) {

    std::lock\_guard<std::mutex> lock(queueMutex);

    tasks.push(std::move(task));

    condition.notify\_one();

}

void ThreadPool::workerThread() {

    while (!stop) {

        std::function<void()> task;

        {

            std::unique\_lock<std::mutex> lock(queueMutex);

            condition.wait(lock, [this]{

                return stop || !tasks.empty(); // makes the thread wait until there's a task available or pool is stopped.

            });

            if (stop && tasks.empty()) {

                return;

            }

            task = std::move(tasks.front());

            tasks.pop();

        }

        task();

    }

}

int main()

{

    ThreadPool pool(4);

    for (int i=0; i<8; i++) {

        pool.enqueueTask({

            [i] {

                std::cout << "Task " << i << " is running on thread " << std::this\_thread::get\_id() << std::endl;

            }

        });

    }

    std::this\_thread::sleep\_for(std::chrono::seconds(1)); // Give time for tasks to complete

    return 0;

}

Thread Local Storage ::

#include <iostream>

#include <thread>

/\*

- Thread Local Storage (TLS) allows each thread to have it's own instance of a variable. This is useful when threads need to

    maintain state independently from each other.

- Why use thread local storage ?

--- Avoid data races by giving each thread it's own copy of a variable, you avoid the need for synchronization mechanisms

    like mutexes, as each thread operates on it's own independent data.

--- Thread-specific data, it is useful for storing data that is specific to a thread, such as thread IDs, thread-specific counters,

or thread-speicific caches.

\*/

thread\_local int threadId = 0;

void setThreadId(int id) {

    threadId = id;

    std::cout << "Thread " << id << " has thread-local ID : " << threadId << std::endl;

}

int main()

{

    std::thread t1(setThreadId, 1);

    std::thread t2(setThreadId, 2);

    t1.join();

    t2.join();

    return 0;

}

Atomic Operations ::

#include <iostream>

#include <thread>

#include <atomic>

/\*

- Atomic operations on data that are completed in a single step without the possibility of being interrupred by other threads,

std::atmoic provides a way to perform these operations on shared data, ensuring thread safety without the need for locks.

- They are faster than using mutexes becz they don't require the overhead of locking and unlocking.

- They provide a simple to manage shared data in a mutli-threaded envirionment.

\*/

std::atomic<int> count(0);

void incrementCount() {

    for (int i=0; i<1000; i++) {

        count++;

    }

}

int main()

{

    std::thread t1(incrementCount);

    std::thread t2(incrementCount);

    t1.join();

    t2.join();

    std::cout << "Final count value : " << count << std::endl;

    return 0;

}

/\*

- atomic is a special type which ensures the counter variable is incremented atomically, meaning no other thread can interrupt

increment operation.

- counter++ is safe-operation as it guarantees that no threads will modify the variable at the same time.

\*/

Parallel Algorithms ::

#include <iostream>

#include <vector>

#include <algorithm>

#include <execution>

/\*

- It allows us to specify how algorithms should be executed sequentially, parallel or vectorized manner.

- Performance, They allow to leverage multiple CPU cores to perform operations faster on large data sets.

- It makes it easier to write parallel code wihtout worrying about thread management.

\*/

int main()

{

    std::vector<int> numbers(1000000, 1);

    // Execute std::for\_each in parallel

    std::for\_each(std::execution::par, numbers.begin(), numbers.end(),

    [](int &n) {

        n \*= 2;

    });

    std::cout << "First Number : " << numbers[0] << std::endl; // should be 2

    return 0;

}