Threading in C++

All Material is refered from a [video tutorials](https://www.youtube.com/playlist?list=PLk6CEY9XxSIAeK-EAh3hB4fgNvYkYmghp) from CPP NUTS.

1> Threading is supported only in C++ 11 and above versions.

We use threads cause it helps us to achieve parallel operation and faster execution refer Thread\_Use.cpp file.

2> Every program has default one thread which is main() function. Ways of creating thread in C++

1> Using function pointer

2> Using functors

3> Using lambda expression

4> Using Class method

5> Using Static Class method

class ThreadClass

{

public:

    int iVal;

    void startThread(int iVal)

    {

        this->iVal = iVal;

    }

    static void runThread(int iNum)

    {

        int iVal = iNum;

    }

};

void cFuncThread(int iNum)

{

    int iVal = iNum;

}

int main()

{

    int iVal = 0;

    auto lambdaFunc = [&](int iNum)

    {

        iVal = iNum;

    };

    ThreadClass obj;

    thread t\_CFuncPtr(cFuncThread, 5);

    thread t\_LambdaFunc(lambdaFunc, 10);

    thread t\_ClassMethod(&ThreadClass::startThread, &obj, 15);

    thread t\_ClassStaticMethod(&ThreadClass::runThread, 20)

    t\_CFuncPtr.join();

    t\_LambdaFunc.join();

    t\_ClassMethod.join();

    t\_ClassStaticMethod.join();

    return 0;

}

3> Thread Join ->

// TOPIC: Use Of join(), detach() and joinable() In Thread In C++ (C++11)

// JOIN NOTES

// 0 Once a thread is started we wait for this thread to finish by calling join() function on thread object..

// 1 **Double join will result into program termination**.

// 2. If needed we should check thread is joinable before joining. ( using joinable() function)

// DETACH NOTES

// 0. This is used to detach newly created thread from the parent thread.

// 1. **Always check before detaching a thread that it is joinable** otherwise we may end up double detaching and

// double detach() will result into program termination.

// 2. If we have detached thread and main function is returning then the detached thread execution is suspended.

// NOTES: Joinable return true if thread is joinable else returns false.

// Either join() or detach() should be called on thread object, otherwise during thread object's destructor it will

// terminate the program. Because inside destructor it checks if thread is still joinable? yes then it terminates the program.

4> Mutex ->

Refer Thread\_Mutex.cpp file for practical example for Mutex in C++11 & above

1. On Mutex we can use different methods such as lock(), unlock(), try\_lock().
2. Try\_lock() tries to lock the mutex. If it obtains the lock then it returns true and locks the section.
3. If lock is not obtained or currently that mutex is already locked by other thread then it returns false. And skips thread execution it returns basically it does not wait to obtain lock() this is non-blocking execution.

5> Try\_Lock ->

1. Std::try\_lock(m1, m2, m3….mn); this function tries to lock all the lockable objects at once.
2. This is non-blocking call on success returns -1 otherwise zero based index of mutex which it could not lock.
3. If unable to lock any one of given mutex then it will simply release all locks and will go ahead.
4. If it gets lock of all mutexes then unlock should be manually called for all mutexs m1, m2, m3, … mn.

6> Timed Mutex ->

Refer Timed\_Mutex.cpp for more details. Of 2 important methods of try\_lock\_for(); and try\_lock\_until();

// TOPIC: Timed Mutex In C++ Threading (std::timed\_mutex)

// We have learned Mutex, Race Condition, Critical Section in previous video.

// NOTES:

// 0. std::timed\_mutex is blocked till timeout\_time or the lock is aquired and returns true if success

// otherwise false.

// 1. Member Function:

// a. lock()

// b. try lock()

// c. try lock for() ---\ These two functions makes it different from mutex.

// d. try lock until()

// e. unlock.

// EXAMPLE: try lock\_for();

// Waits until specified timeout duration has elapsed or the lock is acquired, whichever comes first.

// On successful lock acquisition returns true, otherwise returns false.

// mutex.try\_lock\_for(); Takes parameter in seconds and thread waits there for specified seconds.

// try\_lock\_until(); Takes parameter from current time how much time thread should wait.

Example code for try\_lock\_until();

auto now = std::chrono::steady\_clock::now();

if(m.try\_lock\_until (now + std::chrono::seconds (1))){

++my Amount;

std::this thread::sleep\_for (std::chrono: :seconds (2));

cout << "Thread " << i <<"Entered" << endl;

m.unlock();

}else{

cout << "Thread " << i << " Couldn't Enter" << endl;

}

7> Recursive Mutex ->

std::recursive\_mutex recMutex;

// TOPIC: Recursive Mutex In C++ (std:: recursive\_mutex)

// NOTES:

// 0. It is same as mutex but, Same thread can lock one mutex multiple times using recursive\_mutex.

// 1. If thread T1 first call lock/try lock on recursive mutex m1, then m1 is locked by T1, now

// as T1 is running in recursion T1 can call lock/try\_lock any number of times there is no issue.

// 2. But if T1 have aquired 10 times lock/try lock on mutex m1 then thread T1 will have to unlock

// it 10 times otherwise no other thread will be able to lock mutex m1.

// It means recursive\_mutex keeps count how many times it was locked so that many times it should be unlocked.

// 3. How many time we can lock recursive\_mutex is not defined but when that number reaches and if we were calling

// lock() it will return std::system\_error OR if we were calling try lock() then it will return false.

// BOTTOM LINE:

// 0. It is similar to mutex but have extra facitility that it can be locked multiple time by same thread.

// 1. If we can avoid recursive\_mutex then we should becuase it brings overhead to the system.

// 2. It can be used in loops also.

8> Lock\_Guard->

Refer Lock\_Guard.cpp in this folder.

// 0. It is very light weight wrapper for owning mutex on scoped basis.

// 1. It aquires mutex lock the moment you create the object of lock guard..

// 2. It automatically removes the lock while goes out of scope,

// 3. You can not explicitly unlock the lock\_guard.

// 4. You can not copy lock\_guard.

9> unique\_lock<mutex> uLock(m1);

1. Unique\_lock<> is same as lock\_guard difference is it can use methods like try\_lock\_for, and try\_lock\_until().

Also it can be used as recursive lock().

1. Also it can have different locking strategies defined. Like defer\_lock, adopt\_lock etc.
2. Major advantage of unique lock is it can be used along with condition\_variable.

10> Condition Variable ->

Refer Condition\_Variable.cpp for more details.

Condition Variable

/IMPORTANT POINT: CV are used for two purpose

// A. Notify other threads

// В. Waiting for some conditions

1. Condition Variable allows running threads to wait on some conditions and once those conditions are met the waiting thread

is notified using :

a. notify\_one();

b. notify\_all();

2. You need mutex to use condition variable.

3. If some thread want to wait on some condition then it has to do these things:

a. Acquire the mutex lock using std: : unique lock<std: :mutex> lock(m);.

b. Execute wait, wait for, or wait until. The wait operations

11> std::future<> / std::promise<> ->

Refer FuturePromise.cpp in Threads.

12> Instead of using future and promise we can use std::async():

1. Std::async (std::launch::async, funcName, parameter1, parameter2); This can be used instead of future and promise.
2. This call automatically creates a thread if launching method is async if it is deferred it will simply call the function whenever fut.get() method is called.

