CPP Notes from Lectures:

```
//#############################################################################
/*
 *  Concurrent C++
 *
 */
 /*

 THREADS CAN BE CREATED BY "std:thread(CALLABLE_OBJECT,VARIABLE_#_OF_PARAMETERS)
 std::thread(a,3,4,5);  --> where 'a' is a function and '3,4,5' are parameters.

 -----> It can be created by
 1. std::thread t1(a,6)
 2. std::async(std::launch::async,a,6);
 "
 */
// First example:
void thread1() {
	std::cout << "Helldo, Worlds" << std::endl;
}
int main() {
	std::thread t1(thread1);
	t1.join();   // main thread wait for t1 to finish
	//t1.detach();  // main thread let t1 to run on its own: t1 is a daemon process.
	   // C++ runtime library is responsible returning t1's resources
	   // and main thread may finish before t2 prints "Hello"
	return 0;
}
```

```
// If neither detach nor join is called, terminate() will be called for the t1.
// A thread can only be joined once or detached once. After it is joined on detached
// it becomes unjoinable ( t.joinable() returns false )



// Second Example: Racing condition
class Fctor {
	ofstream& m_str;
public:
	Fctor(ofstream& s):m_str(s) {}   // Reference member can only be initialized
	void operator()() {
		for (int i=0; i>-100; i--)
			m_str << "from t1: " << i << endl;
	}
};

int main() {
	cout << "Hollo Bo" << endl;
	ofstream f;
	f.open("log.txt");

	Fctor fctor(f);
	std::thread t1(fctor);
	for (int i=0; i<100; i++)
		f << "from main: " << i << endl;
	t1.join();
	f.close();
	return 0;
}
```

```
// 1. expeirment with t1.join()
// 2. expeirment with t1.detach()
// 3. expeirment with t1 closing file
// 4. expeirment with  cout instead of f and m_str
// 4. experiment with t1.get_id();
// 4. experiment with thread::hardware_concurrentcy();

// A common solution: do not share, make a copy.

// If exception happens in the main()'s for loop, t1.join() will not be called.
int main() {
	cout << "Hollo Bo" << endl;
	ofstream f;
	f.open("log.txt");
	Fctor fctor(f);
	std::thread t1(fctor);
	try {
		for (int i=0; i<100; i++)
			cout << "from main: " << i << endl;  // Exception may happen here
	} catch (...) {
		t1.join();
		throw;
	}

	t1.join();
	f.close();
	return 0;
}
```

```
// Alternative way: RAII
class ThreadJoiner {
	thread& m_th;
public:
	explicit ThreadJoiner(thread& t):m_th(t) {}
	~ThreadJoiner() {
		if(m_th.joinable()) {
			m_th.join();
		}
	}
};

int main() {
	cout << "Hollo Bo" << endl;
	ofstream f;
	f.open("log.txt");

	Fctor fctor(f);
	std::thread t1(fctor);
	ThreadJoiner tj(t1);

	for (int i=0; i<100; i++)
		cout << "from main: " << i << endl;

	f.close();
	return 0;
}
```

```
// We can also create a ThreadDetacher class
// Prerequisit: join() or detach() can happen at the end of the function

// Passing parameters to a thread

void call_from_thread(string& msg) {
	msg = "Beauty is only skin-deep";
	cout << "t1 says: " << msg << endl;
}

int main() {

	string s = "A friend in need is a friend indeed.";
	std::thread t1(call_from_thread, std::ref(s));
	// Function templates ref and cref are helper functions that generate an object of type std::reference_wrapper

	t1.join();

	cout << "main says: " << s << endl;
	return 0;
}
// Paramters are always passed by value (copied).  why? same reason as bind(): deferred execution means the parmeter objects might not be valid at the time of execution
// To pass by reference:
// 1. use std::ref
// 2. use pointer
```

```
// To pass a class method as thread's initial function, use pointers
class A {
public:
	void call_from_thread(string* msg) {
		*msg = "Beauty is only skin-deep";
		cout << "t1 says: " << *msg << endl;
	}
};

int main() {

	string s = "A friend in need is a friend indeed.";
	A a;
	std::thread t1(&A::call_from_thread, &a, &s);
	t1.detach();

	cout << "main says: " << s << endl;
	return 0;
}



// Thread with moving parameters
void call_from_thread(string msg) {
	cout << "t1 says: " << msg << endl;
}

int main() {
	string* ps = new string("A friend in need is a friend indeed.");
```

```
        std::thread t1(call_from_thread,  std::move(*ps));
        t1.join();

        cout << "main: " << *ps << endl;
        return 0;
}

// A thread object cannot be copied
// But it can be moved.  Like fstream, unique_ptr, etc.

//Case 1:
std::thread t1(call_from_thread);
std::thread t2 = move(t1);  // t2 become the owner of the thread
t2.join();

//Case 2:
thread f() {
  // ...
  return std::thread(call_from_thread);   // move semantics
}

//Case 3:
void f(std::thread t);
int main() {
  f(std::thread(call_from_thread));  // move semantics
  //...
}

// Revisit the RAII example with move semantics
```

```
//#############################################################################
/*
 *  Concurrent C++
 *
 */
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <memory>
#include <thread>
#include <mutex>
/* Using mutex to synchronize threads */
std::mutex m_mutex;

void shared_print(string id, int value) {
	std::lock_guard<std::mutex> locker(m_mutex);
  //m_mutex.lock();
  // if (m_mutex.trylock()) {...}
	cout << "From " << id << ": " << value << endl;
  //m_mutex.unlock();
}
class Fctor {
public:
	void operator()() {
		for (int i=0; i>-100; i--)
			shared_print("t1", i);
	}
};
```

```
int main() {
	Fctor fctor;
	std::thread t1(fctor);

	for (int i=0; i<100; i++)
		shared_print("main", i);

	t1.join();

	return 0;
}



/*
 * 1. Avoid global variables
 * 2. Mutex should bundle together with the resource it is protecting.
 */
class LogFile {
	std::mutex m_mutex;
	ofstream f;
public:
	LogFile() {
		f.open("log.txt");
	} // Need destructor to close file
	void shared_print(string id, int value) {
		std::lock_guard<mutex> locker(m_mutex);
		f << "From " << id << ": " << value << endl;
	}
	// Never leak f to outside world, like this:
```

```
        void processf(void fun(ofstream&)) {
                fun(f);
        }

};

class Fctor {
        LogFile& m_log;
public:
        Fctor(LogFile& log):m_log(log) {}
        void operator()() {
                for (int i=0; i>-100; i--)
                        m_log.shared_print("t1", i);
        }
};

int main() {
        LogFile log;
        Fctor fctor(log);
        std::thread t1(fctor);

        for (int i=0; i<100; i++)
                log.shared_print("main", i);

        t1.join();

        return 0;
}
```

```
/* the ofstream will not be protected if it is leaked out */
// Example: add a LogFile method:
        ofstream* getStream() {
                return &f;
        }
// main():
        ofstream* fs = log.getStream();
        *fs << "ddummy" << endl;  // Unprotected access

// A more hidden leakage:
        void formated_print(function<ofstream (ofstream&)> usrFunc){
                std::lock_guard<mutex> locker(m_mutex);
                usrFunc(f);
        }

/* Important: Do not let your user work on protected data directly */


/* Interface is not thread safe */
class stack {
        int* _data;
        std::mutex _mu;
public:
        int& pop(); // pops off the item on top of the stack
        int& top(); // returns the item on top
        //...
};

void function_1(stack& st) {
```

```
	int v = st.pop();
	process(v);
}

/*
Avoiding Data Race:
1. Use mutex to syncrhonize data access;
2. Never leak a handle of data to outside
3. Design interface appropriately.
*/


/* Deadlock */
class LogFile {
	std::mutex _mu;
	std::mutex _mu_2;
	ofstream f;
public:
	LogFile() {
		f.open("log.txt");
	}
	void shared_print(string id, int value) {
		std::lock_guard<mutex> locker(_mu);
		std::lock_guard<mutex> locker1(_mu_2);
		f << "From " << id << ": " << value << endl;
	}
	void shared_print_2(string id, int value) {
		std::lock_guard<mutex> locker1(_mu_2);
		std::lock_guard<mutex> locker(_mu);
```

```
        f << "From " << id << ": " << value << endl;
    }
};
// Solution: lock the mutexes in a fixed order
//

/* C++ 11 std::lock */
class LogFile {
    std::mutex m_mutex;
    std::mutex m_mutex_2;
    ofstream f;
public:
    LogFile() {
        f.open("log.txt");
    }
    void shared_print(string id, int value) {
        std::lock(m_mutex, m_mutex_2);
        std::lock_guard<mutex> locker(m_mutex, std::adopt_lock);
        std::lock_guard<mutex> locker1(m_mutex_2, std::adopt_lock);
        f << "From " << id << ": " << value << endl;
    }
    void shared_print_2(string id, int value) {
        std::lock(m_mutex, m_mutex_2);
        std::lock_guard<mutex> locker1(m_mutex_2, std::adopt_lock);
        std::lock_guard<mutex> locker(m_mutex, std::adopt_lock);
        f << "From " << id << ": " << value << endl;
    }
};
```

/* Avoiding deadlock

1. Prefer locking single mutex.

2. Avoid locking a mutex and then calling a user provided function.

3. Use std::lock() to lock more than one mutex.

4. Lock the mutex in same order.

Locking Granularity:

- Fine-grained lock: protects small amount of data

- Coarse-grained lock: protects big amount of data

*/

/* Deferred Lock */

```
class LogFile {
	std::mutex m_mutex;
	ofstream f;
public:
	LogFile() {
		f.open("log.txt");
	}
	void shared_print(string id, int value) {
		//m_mutex.lock();  // lock before lock_guard is created
		//std::lock_guard<mutex> locker(m_mutex, std::adopt_lock);
		std::unique_lock<mutex> locker(m_mutex, std::defer_lock);
		locker.lock();  // Now the mutex is locked
		f << "From " << id << ": " << value << endl;
	}
};
```

```
// Objects cannot be copied but can be moved: thread, packaged_task, fstream, unique_ptr
// unique_lock
// mutex can neither be copied nor moved



/* unique_lock for transferring mutex ownership */


class LogFile {
	std::mutex m_mutex;
	ofstream f;
public:
	LogFile() {
		f.open("log.txt");
	}
	unique_lock<mutex> giveMeLock() {
		return unique_lock<mutex>(m_mutex);  // Moved
	}
	void shared_print(string id, int value) {
		std::unique_lock<mutex> locker(m_mutex);
		f << "From " << id << ": " << value << endl;
	}
};


int main() {
	LogFile log;
	unique_lock<mutex> locker = log.giveMeLock();
  // I don't want to shared_print anything, but I don't want anybody else to do that either untill I am done.
```

```
  // I can also release the lock before locker is destroyed
  locker.unlock();  // lock_guard can't unlock

  //...
  // allow other thread to use log

  locker.lock();  // lock again. -- finer grained lock alows more resource sharing

        return 0;
}



/* Lock for Initialization */
class LogFile {
        std::mutex m_mutex;
        ofstream f;
public:
        void shared_print(string id, int value) {
                if (!f.is_open()) {   // lazy initialization
                        std::unique_lock<mutex> locker(m_mutex);
                        f.open("log.txt");   // This must be synchronized
                }
                f << "From " << id << ": " << value << endl;  // I don't care this is not synchronized
        }
};

// Problem: log.txt still will be opened multiple times
```

```
class LogFile {
        std::mutex m_mutex;
        ofstream f;
public:
        void shared_print(string id, int value) {
                if (!f.is_open()) {   // lazy initialization   -- A
                        std::unique_lock<mutex> locker(m_mutex);
      if (!f.is_open()) {
                          f.open("log.txt");   // This must be synchronized  -- B
      }
                }
                f << "From " << id << ": " << value << endl;  // I don't care this is not synchronized
        }
};
// Double-checked locking
// Problem: race condition between point A and point B



// C++ 11 solution:
class LogFile {
  static int x;
        std::mutex m_mutex;
        ofstream f;
        std::once_flag m_flag;
        void init() { f.open("log.txt"); }
public:
        void shared_print(string id, int value) {
    std::call_once(_flag, &LogFile::init, this); // init() will only be called once by one thread
                //std::call_once(m_flag, [&](){ f.open("log.txt"); });  // Lambda solution
```

```
                //std::call_once(_flag, [&](){ _f.open("log.txt"); });  // file will be opened only once by one thread
                f << "From " << id << ": " << value << endl;
        }
};
int LogFile::x = 9;

//Note: once_flag and mutex cannot be copied or moved.
//      LogFile can neither be copy constructed nor copy assigned

// static member data are guaranteed to be initialized only once.



std::recursive_mutex
// A mutex can be locked multiple times (it must be released multiple times also)

////////////////////////////////////////////////



/* Synchrosizing Operations */
#include "stdafx.h"

#include <deque>
#include <functional>
#include <iostream>
#include <fstream>
#include <thread>
#include <string>
#include <mutex>
```

```
std::deque<int> q;
std::mutex mu;

void function_1() {
	int count = 10;
	while (count > 0) {
		std::unique_lock<mutex> locker(mu);
		q.push_front(count);
		locker.unlock();
		std::this_thread::sleep_for(chrono::seconds(1));
		count--;
	}
}

void function_2() {
	int data = 0;
	while ( data != 1) {
		std::unique_lock<mutex> locker(mu);
		if (!q.empty()) {
			data = q.back();
			q.pop_back();
			locker.unlock();
			cout << "t2 got a value from t1: " << data << endl;
		} else {
			locker.unlock();
		}
		std::this_thread::sleep_for(chrono::milliseconds(10));
	}
```

```
}
// It is hard to set the sleep time.
int main() {
	std::thread t1(function_1);
	std::thread t2(function_2);
	t1.join();
	t2.join();
	return 0;
}



// Using conditional variable and mutex
void function_1() {
	int count = 10;
	while (count > 0) {
		std::unique_lock<mutex> locker(mu);
		q.push_front(count);
		locker.unlock();
		cond.notify_one();  // Notify one waiting thread, if there is one.
		std::this_thread::sleep_for(chrono::seconds(1));
		count--;
	}
}

void function_2() {
	int data = 0;
	while ( data != 1) {
		std::unique_lock<mutex> locker(mu);
```

```
                cond.wait(locker, [](){ return !q.empty();} );  // Unlock mu and wait to be notified
                        // relock mu
                data = q.back();
                q.pop_back();
                locker.unlock();
                cout << "t2 got a value from t1: " << data << endl;
        }
}



/*
FUTURE, PROMISE, ASYNC

If we want the data to be returned from child thread :
1. We have to send a variable by reference to the child thread and also manage it like
        -- As it will be a shared variable we have to lock it and unlock it using mutex
        -- TO make sure that the value is read after child thread is written, we have to use condition variable

So, to avoid the above complex scenarios, we use the future,async and promise in the below manner.

*/
/* For threads to return values: future */
int factorial(int N) {
        int res = 1;
        for (int i=N; i>1; i--)
                res *= i;
        return res;
}
```

```
int main() {
	//future<int> fu = std::async(factorial, 4);  --> initial syntax which returns value to the parent thread.
	future<int> fu = std::async(std::launch::deferred | std::launch::async, factorial, 4); // parmater "async" will create a thread but the parameter deferred will defer the thread creation till ".get()" function called.
	cout << "Got from child thread #: " << fu.get() << endl;
	// fu.get();  // if two times called then it'll crash
	return 0;
}

/*

PROMISE IS USED TO PASS THE DATA FROM PARENT TO CHILD THREAD

*/

/* Asynchronously provide data with promise */
int factorial(future<int>& f) {
	// do something else

	int N = f.get();     // If promise is distroyed, exception: std::future_errc::broken_promise
	cout << "Got from parent: " << N << endl;
	int res = 1;
	for (int i=N; i>1; i--)
		res *= i;

	return res;
}
```

```
int main() {
	promise<int> p;
	future<int> f = p.get_future();

	future<int> fu = std::async(std::launch::async, factorial, std::ref(f));

	// Do something else
	std::this_thread::sleep_for(chrono::milliseconds(20));
	//p.set_value(5);
	//p.set_value(28);  // It can only be set once
	p.set_exception(std::make_exception_ptr(std::runtime_error("Flat tire")));

	cout << "Got from child thread #: " << fu.get() << endl;
	return 0;
}

/* shared_future */
int factorial(shared_future<int> f) {
	// do something else

	int N = f.get();     // If promise is distroyed, exception: std::future_errc::broken_promise
	f.get();
	cout << "Got from parent: " << N << endl;
	int res = 1;
	for (int i=N; i>1; i--)
		res *= i;

	return res;
}
```

```
int main() {
	// Both promise and future cannot be copied, they can only be moved.
	promise<int> p;
	future<int> f = p.get_future();
	shared_future<int> sf = f.share();

	future<int> fu = std::async(std::launch::async, factorial, sf);
	future<int> fu2 = std::async(std::launch::async, factorial, sf);

	// Do something else
	std::this_thread::sleep_for(chrono::milliseconds(20));
	p.set_value(5);

	cout << "Got from child thread #: " << fu.get() << endl;
	cout << "Got from child thread #: " << fu2.get() << endl;
	return 0;
}


/* async() are used in the same ways as thread(), bind() */
class A {
public:
	string note;
	void f(int x, char c) { }
	long g(double x) { note = "changed"; return 0;}
	int operator()(int N) { return 0;}
};
A a;
```

```
int main() {
	a.note = "Original";
	std::future<int> fu3 = std::async(A(), 4);    // A tmpA;  tmpA is moved to async(); create a task/thread with tmpA(4);
	std::future<int> fu4 = std::async(a, 7);
	std::future<int> fu4 = std::async(std::ref(a), 7); // a(7);  Must use reference wrapper
	std::future<int> fu5 = std::async(&a, 7); // Won't compile

	std::future<void> fu1 = std::async(&A::f, a, 56, 'z'); // A copy of a invokes f(56, 'z')
	std::future<long> fu2 = std::async(&A::g, &a, 5.6);    // a.g(5.6);  a is passed by reference
		// note: the parameter of the invocable are always passed by value, but the invokeable itself can be passed by ref.
	cout << a.note << endl;
	return 0;
}
/*
	std::thread t1(a, 6);
	std::async(a, 6);
   std::bind(a, 6);
   std::call_once(once_flag, a, 6);

	std::thread t2(a, 6);
	std::thread t3(std::ref(a), 6);
	std::thread t4(std::move(a), 6);
	std::thread t4([](int x){return x*x;}, 6);
	std::thread t5(&A::f, a, 56, 'z');  // copy_of_a.f(56, 'z')
	std::thread t6(&A::f, &a, 56, 'z');  // a.f(56, 'z')
*/
```

```
/* packaged_task */

std::mutex mu;
std::deque<std::packaged_task<int()> > task_q;

int factorial(int N) {
	int res = 1;
	for (int i=N; i>1; i--)
		res *= i;

	return res;
}

void thread_1() {
	for (int i=0; i<10000; i++) {
		std::packaged_task<int()> t;
		{
			std::lock_guard<std::mutex> locker(mu);
			if (task_q.empty())
				continue;
			t = std::move(task_q.front());
			task_q.pop_front();
		}
		t();
	}
}
```

```
int main() {
	std::thread th(thread_1);

	std::packaged_task<int()> t(bind(factorial, 6));
	std::future<int> ret = t.get_future();
	std::packaged_task<int()> t2(bind(factorial, 9));
	std::future<int> ret2 = t2.get_future();
	{
		std::lock_guard<std::mutex> locker(mu);
		task_q.push_back(std::move(t));
		task_q.push_back(std::move(t2));
	}
	cout << "I see: " << ret.get() << endl;
	cout << "I see: " << ret2.get() << endl;

	th.join();
	return 0;
}



/* Summary
 * 3 ways to get a future:
 * - promise::get_future()
 * - packaged_task::get_future()
 * - async() returns a future
 */
```

```
/* threads with time constrains */

int main() {
   /* thread */
   std::thread t1(factorial, 6);
   std::this_thread::sleep_for(chrono::milliseconds(3));
   chrono::steady_clock::time_point tp = chrono::steady_clock::now() + chrono::microseconds(4);
   std::this_thread::sleep_until(tp);

   /* Mutex */
   std::mutex mu;
   std::lock_guard<mutex> locker(mu);
   std::unique_lock<mutex> ulocker(mu);
   ulocker.try_lock();
   ulocker.try_lock_for(chrono::nanoseconds(500));
   ulocker.try_lock_until(tp);

   /* Condition Variable */
   std:condition_variable cond;
   cond.wait_for(ulocker, chrono::microseconds(2));
   cond.wait_until(ulocker, tp);

   /* Future and Promise */
   std::promise<int> p;
   std::future<int> f = p.get_future();
   f.get();
   f.wait();
   f.wait_for(chrono::milliseconds(2));
```

```
    f.wait_until(tp);

    /* async() */
    std::future<int> fu = async(factorial, 6);

    /* Packaged Task */
    std::packaged_task<int(int)> t(factorial);
    std::future<int> fu2 = t.get_future();
    t(6);

            return 0;
}




   // Together with thread library
            std::this_thread::sleep_until(steady_clock::now() + seconds(3));

            std::future<int> fu;
            fu.wait_for(seconds(3));
            fu.wait_until(steady_clock::now() + seconds(3));

            std::condition_variable c;
            std::mutex mu;
            std::unique_lock<std::mutex> locker(mu);
            c.wait_for(locker, seconds(3));
            c.wait_until(locker, steady_clock::now() + seconds(3));
```

MY NOTES:

EXCEPTION HANDLING:

```
/*
#include<iostream>
#include<thread>
#include<exception>
using namespace std;

class MyException : public exception {
	const char * what()  const throw() {
		return "Custom exception : Cannot divide by zero";
	}
};

int divide() {
	int c;
	cout << "c:";
	cin >> c;
	cout << "c:" << c;
	if (c == 0) {
		throw "airthemetic exception";
	}
	int b = 3 / c;
	return b;
}

int divide1() {
	int c;
	cout << "c:";
	cin >> c;
	cout << "c:" << c;
	if (c == 0) {
		throw new MyException();
	}
	int b = 3 / c;
	return b;
}
int main() {

	try {
		//divide();
		divide1();
		cout << "Hi from main";
	}
	catch (const char *msg) {
		cout << "exception is thrown : " << msg;
	}
	catch (exception &e) {
		cout << e.what();
	}
	catch (...) {
		printf("exception is thrown");
```

```
	}
	int a;
	cin >> a;
	return 0;
}
*/

/*
#include <iostream>
#include <exception>
using namespace std;

class MyException : public exception {
	virtual const char * what() const throw () {
		return "C++ Exception";
	}
};

int main() {
	try {
		throw new MyException();
	}
	catch (exception& e) {
		//std::cout << "MyException caught" << std::endl;
		//std::cout << e.what() << std::endl;
	}
	int a;
	cin >> a;
}

*/


// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception : public exception
{
	virtual const char* what() const throw()
	{
		return "My exception happened";
	}
};

class A {
	int a;

};

/*
int main() {
	try
	{
		A a = new A;
		myexception e = new myexception;
		cout << "Typeof:"<<typeid(e).name();
```

```
		throw  myexception() ;
	}
	catch (exception& e)
	{
		cout << e.what() << '\n';
	}
	int a;
	cin >> a;
	return 0;
}

*/
```

THREAD PROGRAMS:

```
#include<iostream>
#include<thread>
#include<mutex>
#include<string>
#include <fstream>

using namespace std;

class LogFile {
	std::mutex m_mutex;
	std::ofstream f;
public:
	LogFile() {
		f.open("log.txt");
	} // Need destructor to close file
	void shared_print(string id, int value) {
		//cout << "waiting for lock";
		std::lock_guard<mutex> locker(m_mutex);
		//cout << "lock acquired";
		f << "From " << id << ": " << value << endl;
	}
	// Never leak f to outside world, like this:
	void processf(void fun(ofstream&)) {
		fun(f);
	}

};

class Fctor {
	LogFile& m_log;
public:
	Fctor(LogFile& log) :m_log(log) {}
	void operator()() {
		for (int i = 0; i > -100; i--)
			m_log.shared_print("t1", i);
	}
};
```

```
/*
int main() {


	LogFile log;
	Fctor fctor(log);



	std::thread t1(fctor);


	std::ofstream f1;
	f1.open("log.txt");
	f1 << "I got the lock much before" << endl;
	for (int i = 0; i < 100; i++)
		log.shared_print("main", i);



	t1.join();



	cout << "at the end";

	int a;
	cin >> a;
	cout << a;
	return 0;
}
*/
```

RACE CONDITION AND MUTEX:

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

std::mutex mu;


void shared_print(string msg, int id) {
	//mu.lock();

	std::lock_guard<std::mutex> lk_guard(mu);
	std::cout<< msg << id << endl;
	//mu.unlock();
}


/*
1. Moving the shared resource into a common function called critical section
```

```
void shared_print(string msg, int id) {
	std::cout<< msg << id << endl;
}
*/


/*
2. Protecting the critical section code using a mutex variable

void shared_print(string msg, int id) {
	mu.lock();
	std::cout<< msg << id << endl;
	mu.unlock();
}

Using mutex will create a problem like, If any exception occurs in the critical section
code and we are unable
to reach the unlock statement then the shared resource gets locked forever.

So, we will be using "lock_guard", which uses RAII method(Whenever object goes out of
scope it gets unlocked in the destructor)
*/
void function_1() {
	for (int i = 0;i < 100;i++) {
		shared_print("From child: ", i);
	}
}


/*
3. Lock Guard is used in the following manner.

void shared_print(string msg, int id) {
	//mu.lock();

	std::lock_guard<std::mutex> lk_guard(mu);  //RAii initialisation
	std::cout<< msg << id << endl;
	//mu.unlock();
}
*/

/*
4. Still there is a problem in the above manner because "cout" is not completely
protected(Anyone can use "cout" in any
of the other functions).
So, we have to bind the mutex to the resource, that can be done by using a class and in
that class
variables as mutex and resource.

class LogFile{
	std::mutex m_mutex;
	ofstream f;
public:
	LogFile(){
		f.open("log.txt");
	} //Need destructor to close the file
```

```
	void shared_print(string id, int value){
		std::lock_guard<mutex> locker(m_mutex);
		f<< "From" <<id <<":"<<value <<endl;
	}

	// Never return f to the outside world
	ofstream& getStream(){
		return f;
	}

	//Never pass f as an argument to user provided function
	void processf(void fun(ofstream&)){
		fun(f);
	}
}

// In main function we'll create LogFile class and call the functions and we'll
pass it as pass by reference to the function.

int main(){
	LogFile log;
	std::thread t1(function_1, std::ref(log));
}
*/

/*
int main() {

	std::thread t1(function_1);

	for (int i = 0;i < 100;i++) {
		shared_print("From main: ", i );
	}

	t1.join();

	int a;
	cin >> a;
	return 0;

}
*/
```

UNIQUE LOCK AND LAZY INITIALIZATION:

```
#include <iostream>
#include <fstream>
#include <string>
#include <memory>
#include <thread>
#include <mutex>
```

```
using namespace std;
/*
 * 1. Avoid global variables
 * 2. Mutex should bundle together with the resource it is protecting.
 */

/*
1. Unique lock provides more flexibility then the "lock_guard"
2. Unlike "lock_guard()", in unique lock you can lock and unlock the mutex at any point
of time, So provides
more fine grained control on locks.
3. In unique lock you can get the ownership of mutex at one time and then lock it later.
              std::unique_lock<mutex> locker(_mu,std::defer_lock);
              // do something else
              locker.lock();
              //...
              locker.unlock();
              //..
              locker.lock();

3. Wrapper class of a mutex either it be "unique_lock" or "lock_guard", cannot be copied,
bit in "unique_lock", it can just be movied
from one object to other, which is like the transfering of ownership of mutex from one
unique lock to the other.

       std::unique_lock<mutex> locker2 = std::move(locker);

4. Performance(lock_guard) > Performance(Unique_lock) // flexibility comes with cost.


*/


class LogFile {
       std::mutex m_mutex;
       ofstream f;
public:
       LogFile() {
              f.open("log.txt");
       } // Need destructor to close file
       void shared_print(string id, int value) {
              std::lock_guard<mutex> locker(m_mutex);
              f << "From " << id << ": " << value << endl;
       }
};


// CALL ONCE:

/*
For the functions or code which should be executed only once we can use the
"std::once_flag" combined with lamba functions
to exceute the code.


class LogFile{
       std::once_flag _flag;
```

```
}
std::call_once(_flag,[&](){_f.open("log.txt");});
*/
```

CONDITION VARIABLE:

```
#include <iostream>
#include <string>
#include <memory>
#include <thread>
#include <mutex>
#include<deque>

/*
In the producer consumer problem, One thread produces the data and the other thread
consumes it, but one thread keeps on

listening to the data status(Queue elements status) which is called "Busy Waiting".

To avoid above situation we use the condition variables.
*/

/*

In the condition variable, we will notify the other thread once task of the current
thread is done.

We use, "std::condition_variable cond" to do that.

We use, "cond.notify_one()" to notify thread which is waiting on that particular
condition.

on the listerner side "cond.wait(<unique-locker>)" is used.


// Using conditional variable and mutex
void function_1() {
	int count = 10;
	while (count > 0) {
		std::unique_lock<mutex> locker(mu);
		q.push_front(count);
		locker.unlock();
		cond.notify_one();				// Notify one waiting
thread, if there is one.
		std::this_thread::sleep_for(chrono::seconds(1));
		count--;
	}
}

void function_2() {
	int data = 0;
	while ( data != 1) {
		std::unique_lock<mutex> locker(mu);		// we use locker parameter
in wait() because when thread is going fo sleep

	// we don't need to hold lock and we will release it.
```

```
	// In order to avoid spurious wake( waking up of thread by iteself instead of

	by notify method) we use a condition in lambda function.
		cond.wait(locker, [](){ return !q.empty();} );  // Unlock mu and wait to be
notified
			// relock mu
		data = q.back();
		q.pop_back();
		locker.unlock();
		cout << "t2 got a value from t1: " << data << endl;
	}
}

---> If you want to notify all the threads waiting on a single condition variable, we
call
"cond.notify_all()" function
*/
using namespace std;

std::deque<int> q;
std::mutex mu;

void function_1() {
	int count = 10;
	while (count > 0) {
		std::unique_lock<mutex> locker(mu);
		q.push_front(count);
		locker.unlock();
		std::this_thread::sleep_for(chrono::seconds(1));
		count--;
	}
}

void function_2() {
	int data = 0;
	while (data != 1) {
		std::unique_lock<mutex> locker(mu);
		if (!q.empty()) {
			data = q.back();
			q.pop_back();
			locker.unlock();
			cout << "t2 got a value from t1: " << data << endl;
		}
		else {
			locker.unlock();
		}
		std::this_thread::sleep_for(chrono::milliseconds(10));
	}
}
// It is hard to set the sleep time.
int main() {
	std::thread t1(function_1);
	std::thread t2(function_2);
	t1.join();
	t2.join();
	return 0;
}
```