Concurrency

Goals:

1. What is concurrency
2. Approaches to concurrency
3. Previous occurrences of multithreading
4. How does multi-threading work

What is Concurrency?

- Concurrency means multiple computations happening at the same time
- Concurrency is great because it allows us to deal with many things at once
- Concurrency is 2 or more activities happening at the same time AND is INDEPENDENT OF EACH OTHER

Where is concurrency?

- Concurrency is everywhere in modern programming
- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer
- Concurrency is essential in modern programming
    - Websites must handle multiple simultaneous users
    - Mobile apps do processing on servers/cloud while being used
    - GUI's code is being compiled, while you edit it

*** To get a computation to run faster, you have to split up a computation into concurrent pieces

CONCURRENT = happening at the same time

- Because we have hardware concurrency, we have computers with multiple cores, that have multiple processors that enable us to run more than 1 task in parallel

Why Concurrency?

- Separate different areas of functionality, when they are happening at the same time
- Parallelizing is getting more things done at once

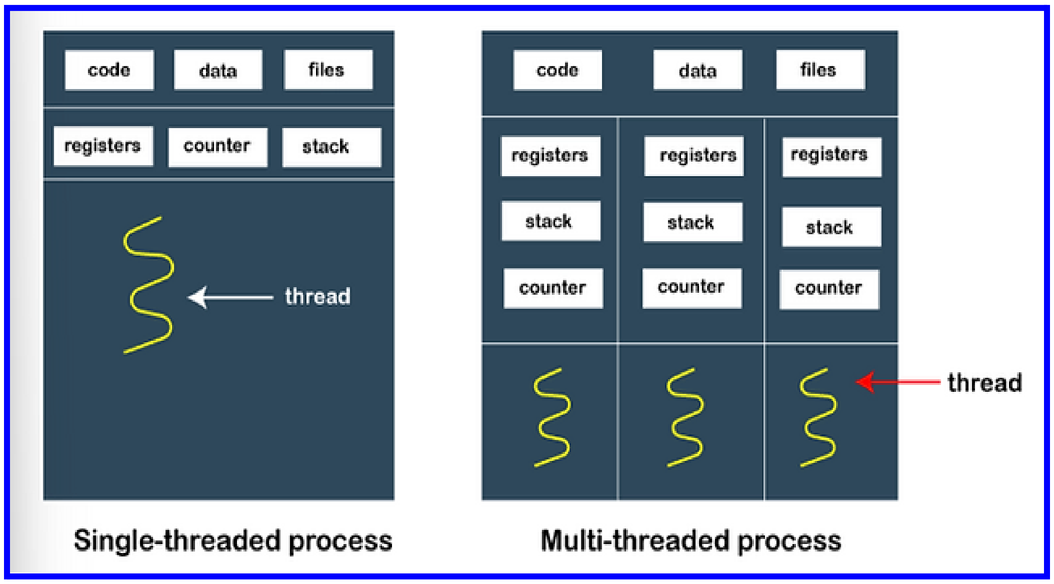
- → Parraleling = several threads perform the same task on different parts of the data at the same time [in parallel]

Why Not Concurrency?

- Tough to code and debug
- Threads are a limited resource

Approaches to Concurrency

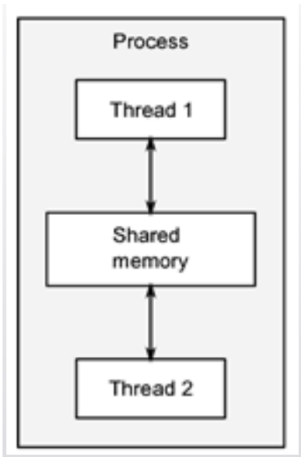
→ multiple single-threaded processes
→ multiple multi-threaded processes

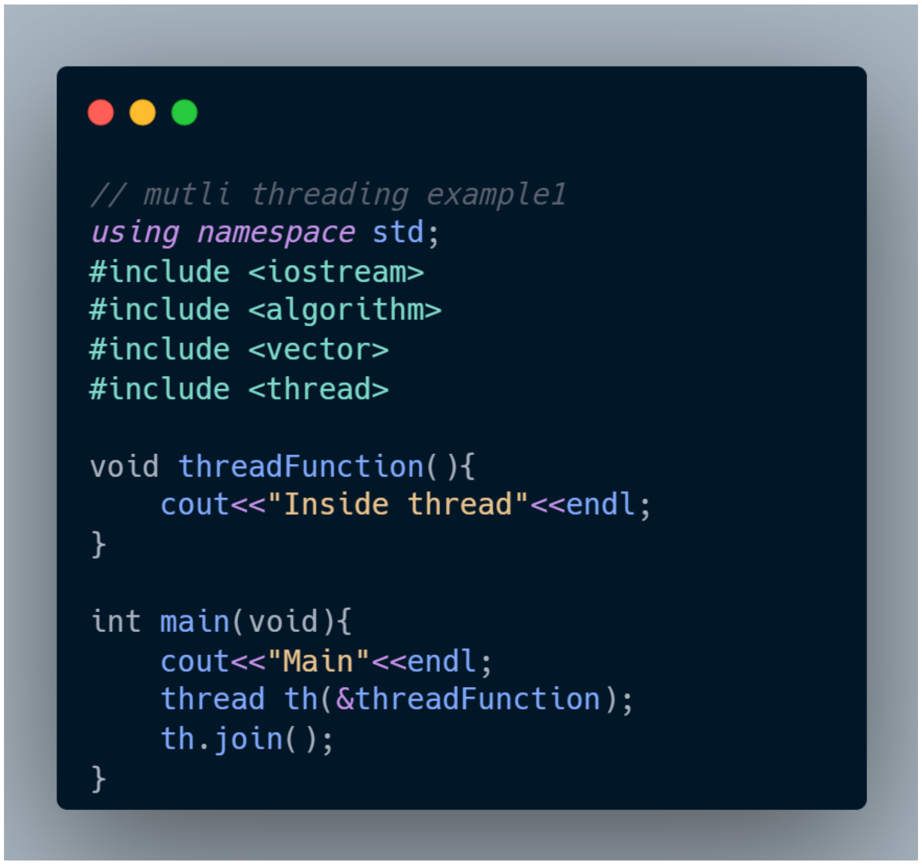


multiple single-threaded processes	multiple multi-threaded processes
• Advantage: Easier to write safe concurrent code • Disadvantage: communication between processes is slow	• Like having multiple developers in an office • Advantage: less overhead/too much work because we have multiple threads working simultaneously + faster • Distadvantge: must make sure the data seen by each thread is consistent

What is a thread?

- Thread = Lightweight process
- Threads are independent of each other but share the same address space
    - Each developer is independent, but their ultimate goal is the same ⇒ complete the task
    - A thread is like a developer
    - A process is like the office a developer is in

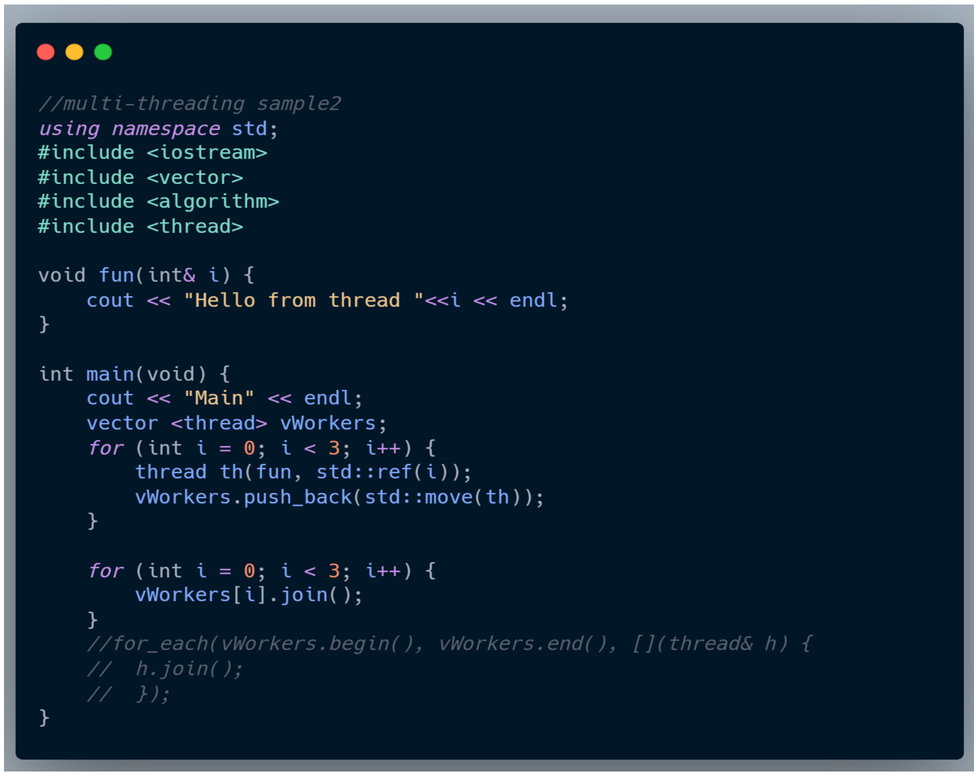




```cpp
// mutli threading example1
using namespace std;
#include <iostream>
#include <algorithm>
#include <vector>
#include <thread>

void threadFunction(){
    cout<<"Inside thread"<<endl;
}

int main(void){
    cout<<"Main"<<endl;
    thread th(&threadFunction);
    th.join();
}
```



```cpp
//multi-threading sample2
using namespace std;
#include <iostream>
#include <vector>
#include <algorithm>
#include <thread>

void fun(int& i) {
    cout << "Hello from thread "<<i << endl;
}

int main(void) {
    cout << "Main" << endl;
    vector <thread> vWorkers;
    for (int i = 0; i < 3; i++) {
        thread th(fun, std::ref(i));
        vWorkers.push_back(std::move(th));
    }

    for (int i = 0; i < 3; i++) {
        vWorkers[i].join();
    }
    //for_each(vWorkers.begin(), vWorkers.end(), [](thread& h) {
    //  h.join();
    //  });
}
```

Problem with Multithreading:

- We need to make sure that the data seen by each thread is the same
- If we don't make sure that the data is the same for both threads, a race condition occurs

Race Condition

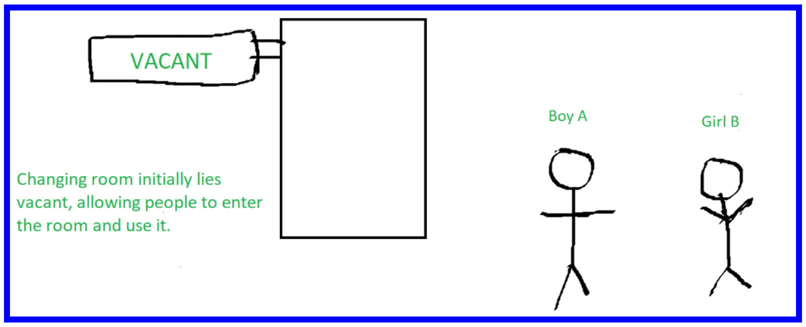
- A race condition occurs when two threads access a shared variable at the same time

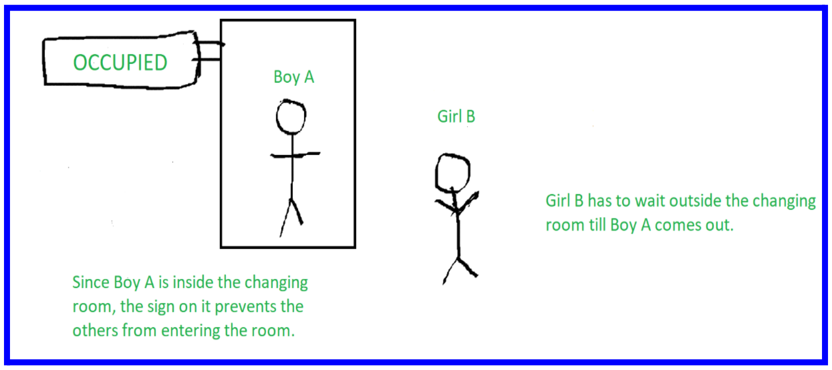
  Steps:

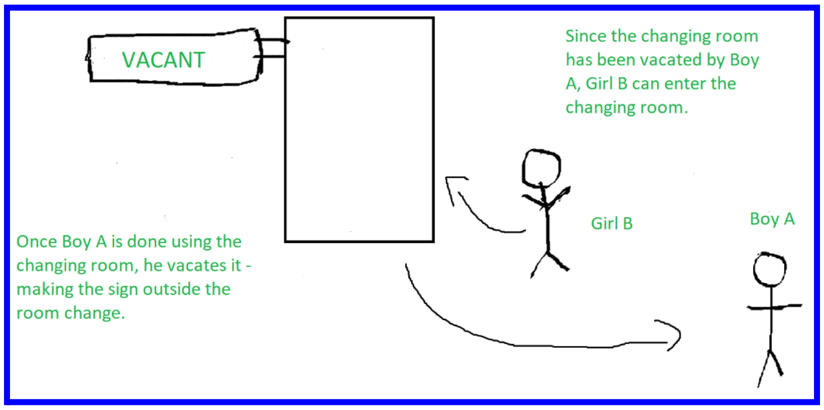
  1. First thread reads the variable, and the 2nd thread also reads the same value from the variable
  2. 1st thread and 2nd thread race to see which can write the value last to the shared variable
  3. The last one wins because the last one is the one that gets overwritten

How can we solve a Race Condition? → Mutual Exclusion / Bathroom sharing!

- Mutual Exclusion says that threads should access the data one thread at a time – properly.
- Steps for Mutual Exclusion:
  1. Lock the data
  2. Access the data
  3. Unlock the data


VACANT
Boy A
Girl B
Changing room initially lies vacant, allowing people to enter the room and use it.


OCCUPIED
Boy A
Girl B
Girl B has to wait outside the changing room till Boy A comes out.
Since Boy A is inside the changing room, the sign on it prevents the others from entering the room.


VACANT
Since the changing room has been vacated by Boy A, Girl B can enter the changing room.
Girl B
Boy A
Once Boy A is done using the changing room, he vacates it - making the sign outside the room change.

Mutual Exclusion - One at a time, please! Requires …

- Library: #include <mutex>
- Object: mutex m;
- Lock function: lock()
- Unlock function: unlock()

By Solving the Race Condition using Mutual Exclusion, we've created another issue called Dead lock! XP

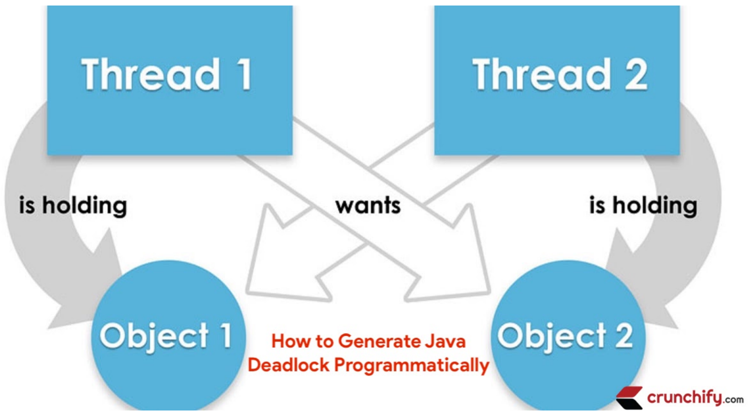
- **Though mutual exclusion ensures that that there's no Race Condition, we need to make sure no thread is keeping data from another thread**

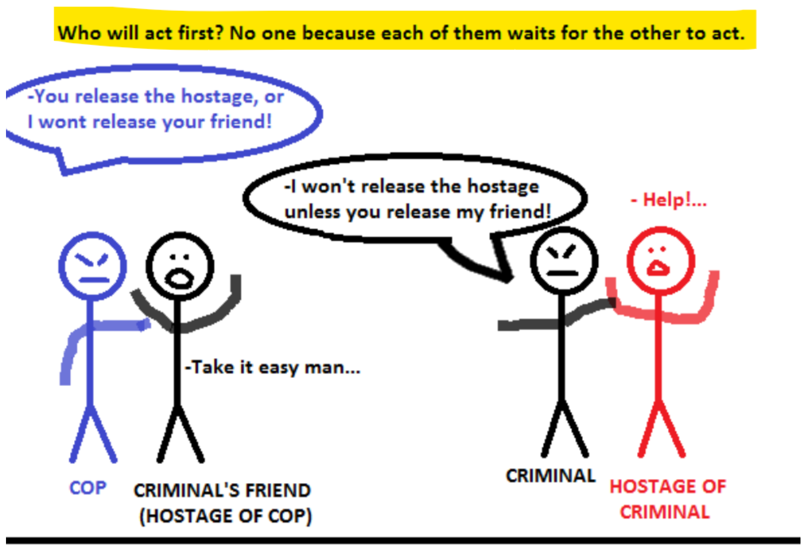
Deadlock

- Deadlock occurs when 2 threads each lock a different variable at the same time, and then try to lock the variable that the other thread has!
  ⇒ Deadlock is like girls getting into a shoe fight! XD

- Because both threads are holding on the variables that the other thread wants, the threads stay deadlocked and nothing happens

- Each thread stops executing and waits for the other thread to release the variable




Who will act first? No one because each of them waits for the other to act.
-You release the hostage, or I wont release your friend!
-I won't release the hostage unless you release my friend!
- Help!...
-Take it easy man...
COP
CRIMINAL'S FRIEND (HOSTAGE OF COP)
CRIMINAL
HOSTAGE OF CRIMINAL

COP: Thread #1 demands Resource #2 but Criminal owns the LOCK
CRIMINAL: Thread #2 demands Resource #1 but Cop owns the LOCK

CRIMINALS FRIEND: Resource #2, the owner of the LOCK is Cop
HOSTAGE OF CRIMINAL: Resource #1, the owner of the LOCK is CRIMINAL

How do we break out of a deadlock - a couple of ways!

1. Lock Sequencing [synchronization]
    - You get to choose the order in which things are locked and locked
    - “You go 1st, you go 2nd, and you go 3rd”
2. Lock hierarchy
    - Prioritize the threads, which one should go 1st and which one goes last
3. Unique_lock()
    - Use unique_lock() to break out of a deadlock because a unique_lock() doesn't always own the mutex its associated with
    - adopt_lock() manages the lock on the mutex
    - defer_lock() tells the computer that the mutex should remain unlocked
    - unique_lock() is like the parent that tells 2 children they have to share, and tells one to wait their turn
    - We use a unique_lock() to have a deferred lock [open bathroom door]
    - Deferred lock ensures the mutex remains open upon construction

How else can we Protect Data from threads?

- Locks are useful to protect data, but the locks need to be set up strategically to ensure the program is 100% thread safe
- Atomic Types strategically set up locks to ensure the program is thread-safe

```
/*
    Problems with Counter Class
    - this code is not safe for a multi-threaded program
    because the value can be changed incorrectly
    - we make this code safe using ATOMIC TYPES
*/
```

```
class Counter{
   private:
        int value;
   public:
        void increment(){value++; }
        void decrement(){value--; }
        int get(){return value;}
};
```

What is an Atomic Type?

- Atomic type is a template class that you can use on any data type to make it thread-safe
- No need for individually implementing locks!
- #include <atomic>

```
// this program shows an atomic class example
/*      Atomic Calss Functions
    load() -> reads contained value
    store() -> stores contained value
    exchange() ->swaps values by setting a new contained value
    and returns previousl contained value
*/
```

```
#include <atomic>

class AtomicCounter{
   private:
        std::atomic<int> value;
   public:
        void increment(){value++;}
        void decrement(){value--;}
        int get(){return value.load();}
};
```

Optimizing threads

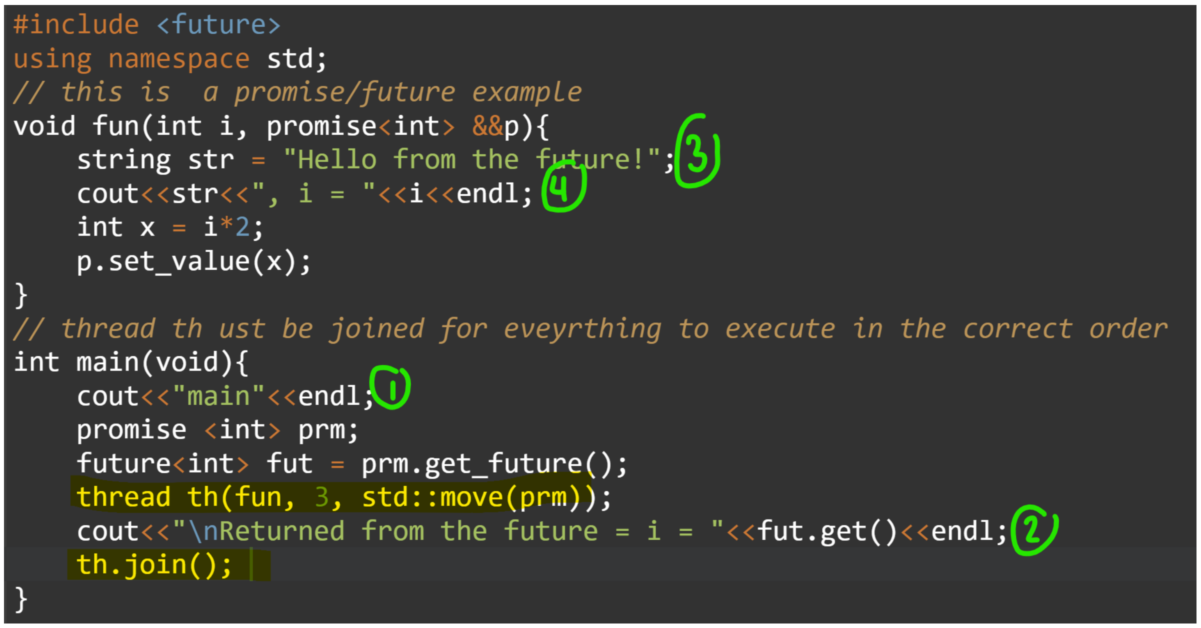
- When one thread is waiting for another thread to complete its taks, it has the following options…
    1) Wait for the other thread to set the flag to green; keep checking the flag to see when its turn comes

        → problem is time is wasted checking the flag, also since the flag is not being able to be accessed by any other thread, that flag is only between those 2 threads :(

    2) Have the waiting thread fall asleep until the mutex is released
        → problem is getting the amount of time to sleep is not fixed – the amount of time varies, which is also a guessing game

Instead of all this waiting and ambiguity – we could be promised the future!

- PROMISE: is when you input a value
- FUTURE: returns the future value – magic!!

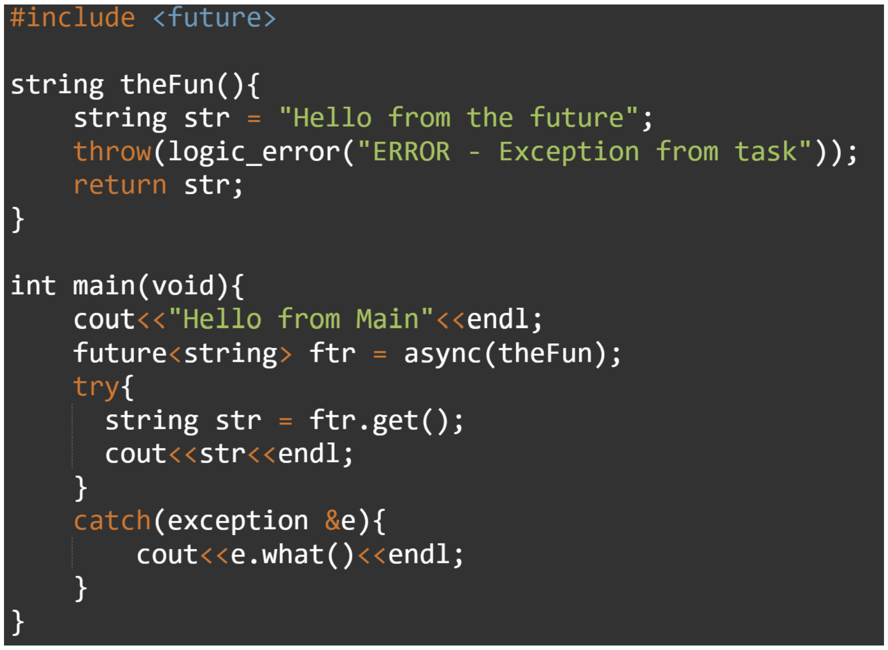


```
#include <future>
using namespace std;
// this is  a promise/future example
void fun(int i, promise<int> &&p){
    string str = "Hello from the future!";③
    cout<<str<<", i = "<<i<<endl;④
    int x = i*2;
    p.set_value(x);
}
// thread th ust be joined for eveyrthing to execute in the correct order
int main(void){
    cout<<"main"<<endl;①
    promise <int> prm;
    future<int> fut = prm.get_future();
    thread th(fun, 3, std::move(prm));
    cout<<"\nReturned from the future = i = "<<fut.get()<<endl;②
    th.join();
}
```

ASYNCHRONOUS TASKS

- The idea of promise and future is implemented in the asynchronous tasks
- Asynchronous task is used when the result is not needed right away

- The point of asynchronous tasks is to delegate long-running operations off of the main thread and onto the background thread(s)
- Let's say you type in “google.com” into the browser, and you're screen is frozen with no message – you as the user only think its frozen but in the back end, you're running an operation that taking time and not allowing the UI to continue to be executed.
- Once that time block is over, execute will continue as normal.
- But the question is why was that time block there in the first place?
- This was because there was a major running operation on the main UI, which should have been delegated to the background threads, ultimately allowing the UI to run smoothly



```cpp
#include <future>

string theFun(){
    string str = "Hello from the future";
    throw(logic_error("ERROR - Exception from task"));
    return str;
}

int main(void){
    cout<<"Hello from Main"<<endl;
    future<string> ftr = async(theFun);
    try{
      string str = ftr.get();
      cout<<str<<endl;
    }
    catch(exception &e){
        cout<<e.what()<<endl;
    }
}
```