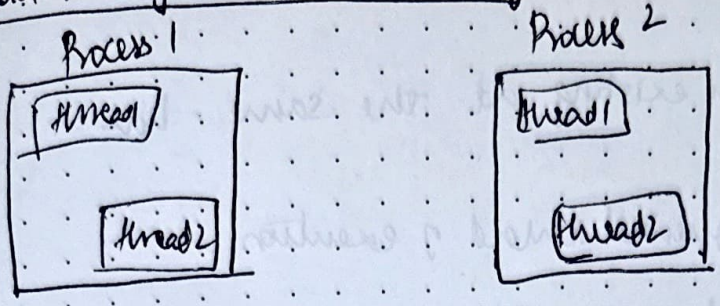
Multithreading and Concurrency



Process: Instance of a program that is getting executed.

It has its own resources like memory, thread etc. OS allocates resources to process when its created.

(javac Test.java)
compilation → generates bytecode that can be executed by JVM.

↳ execution → (java Test) → at this point, JVM starts the new process, here the class which has "public static void main(String[] args)" method

Each process has its own main-memory.

Thread: Lightweight process

or

Smallest sequence of instructions that are executed by CPU independently.

→ 1 process can have multiple threads.

→ When a process is created, it starts with 1 thread and that initial 'main thread' and from that we can create multiple threads to perform task concurrently.

Concurrency

→ 2 (or) more events happening or existing at the same time.

Typically, we have one main sequential thread of execution.

One CPU core executes serially.

But what if we could have a second path in our code executing.

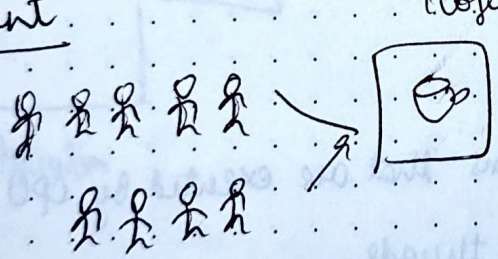
"Performance is the currency of computing"

Parallelism vs Concurrency

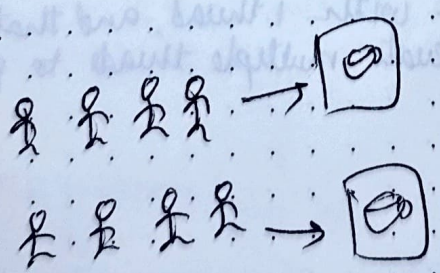
→ multiple things can happen at once, the order matters, and sometimes tasks have to wait on shared resources.

→ Parallelism definition: Everything happens at once, instantaneously.

Concurrent



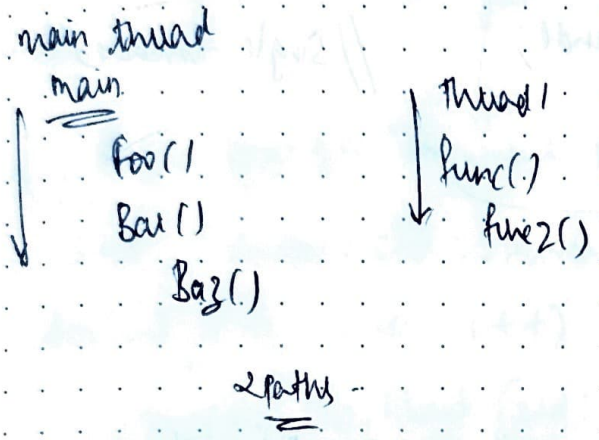
Parallel



Necessity of concurrency

→ Concurrency (like parallelism) is essential because it is necessary for a system to function

Thread based concurrency using C++



2 paths

→ 1 process has multiple threads

- Each threads shares the same code, data & kernel context
- Thread has its own TID
- Thread has its own logical control flow
- Thread has its own stack for local variables

When to use threads

→ When we have heavy computation

→ Using threads on your GPU for graphics

→ Use threads to separate work
  - Gives performance
  - Also simplifies the logic of your problem

→ Standard C++ thread library

(or) pthread by Intel

```
#include <iostream>
#include <thread>

int main(){
    std::cout << "hello" << std::endl;
    return 0;
}
```

// Single threaded.

By Adding one more fn

```
void test(int x){
    std::cout << "Hello from thread!" << std::endl;
    std::cout << "Argument:" << x << std::endl;
}

int main(){
    std::thread myThread(test, 100);
    myThread.join();
}
```

⇒ We need to make the main thread wait so that it executes in order

join() → blocks the current thread until the thread identified by *this finishes its execution.

→ Creating threads using lambda function

Launching a multiple threads from a vector

```
int main(){
  auto lambda = [](int x){
    std::cout<<"Hello from thread"<<std::this_thread::get_id()<<
                                          std::endl;
    std::cout<<"Argument passed in:"<<x<<endl;
  };
  std::vector<std::thread> threads;
  for(int i=0; i<10; i++)
  {
    threads.push_back(std::thread(lambda,100));
    threads[i].join();
  }
  std::cout<<"Hello from my thread"<<std::endl;
  return 0;
}
```

But getting same thread id, why?

So it is nothing but a sequential code..

→ so break & put this into another for loop..

Now it is correct..

but the output is jumpled up

Interleaving of threads is happening, so how to handle this synchronization

Introducing the jthread

jthread for support with auto-joining and cancellation.

C++ 20 RAII → Resource acquisition is initialization.

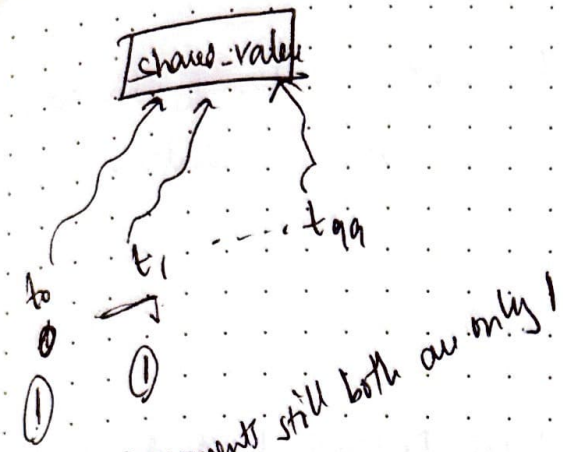
using jthread, we do not have to do the join().

Joins at the end of the scope.

How to have threads working on a shared problem?

How to share data & improvize our performance.

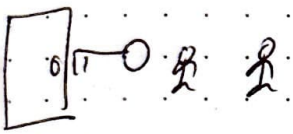
```
static int shared_value = 0;
void shared_v_increment() {
    shared_value = shared_value + 1;
}

int main() {
    std::vector<std::thread> threads;
    for(int i=0; i<10; i++) {
        threads.push_back(std::thread(shared_value++));
    }
    for (int i=0; i<10; i++){
        threads[i].join();
    }
    std::cout<<"shared value" << shared_value << std::endl;
}
```



This is called data race condition.

So, how to fix this ?

mutex → mutual exclusion



thread access shared state then read & write only after its done. thread1 does it.

#include <mutex>

std:: mutex glock;

→ now inside this

```
void shared_value_increment(){
    glock.lock();
    shared_value += 1
    glock.unlock();
}
```

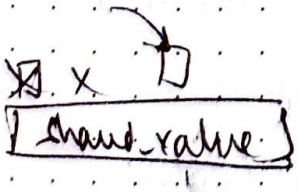
lock() → locks the mutex, blocks if the mutex is not available

unlock() → unlocks the mutex

Now, it works.

t_0 t_1 t_2 t_3 ... t_{99}

Say t2 acquires the lock first, then all others are blocked.



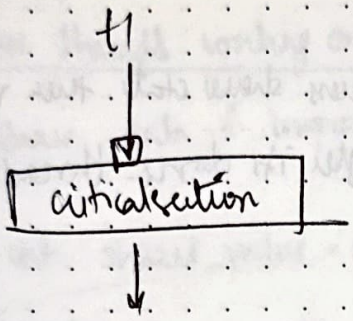
** Won't this slow down the performance ?

So, critical sections should be protected

→ Thread ~~Scheduler~~ creation is non-deterministic, and execution of Threads is not sequential. It is determined by the OS thread scheduler.

Preventing deadlock:

say we forgot to unlock the code and just one program holds the lock and so the program gets freezed.



Deadlock → blocked (0.)

say

```
lock()
try {
    critical section
}
except
{
}
unlock()
```

but some exception is thrown;
↓
so it doesn't reach the unlock().
So, we need to fix this.
by putting unlock() in except & also outside.

→ lock_guard() → is a mutex wrapper.

The class lock_guard is a mutex wrapper that provides a RAII-style mechanism for own duration of scoped block.

When a lock_guard is created, it attempts to take ownership of the mutex it is given.

```
std::lock_guard<std::mutex> lockguard(glock);
void shared_value_increment(){
    std::lock_guard<std::mutex> lockguard(glock);
    shared_value = shared_value + 1;
}
```

scoped_lock(C++17)

lock_guard(C++11) → Instead of manually locking & unlocking,

↓

Even in presence of try & catch, this works.

Another way to update shared_value without lock

std::atomic

```
#include<atomic>
```

can get rid of our locks

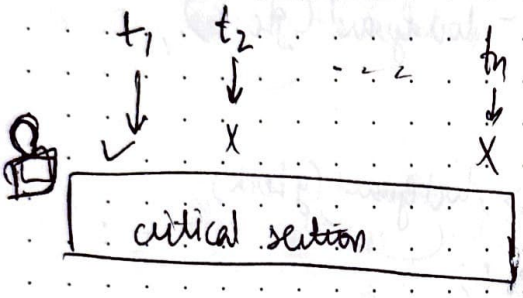
```
static std::atomic<int> shared_value = 0;
void shared_value_increment(){
    shared_value++;
}
```

Any data type are able to do this

operator ++ operator += operator &=
operator -- operator ->=

use only these ⇒

Synchronization of threads with locks



→ What does other threads do when t1 is accessing the critical section?

They are waiting, happens when there is spin lock, as this results in wastage of cpu cycles as they are waiting, because then other threads are constantly asking, do i have the lock x or ?

There must be more efficient way.

std :: condition_variable

(or) is synchronous primitive that can be used to block a thread, multiple threads at the same time, until another thread modifies a shared variable, and notifies the condition variable.

→ execute notify, notify_all on the std :: condition_variable.

1. boolean
2. lock_variable
3. condition_variable
4. 2 threads, worker / repeater
   (1) (2)

```
std::mutex gLock;
std::condition_variable gConditionVariable;
int main(){
    result = 0
    bool notified = false;
    // Repeating thread, // Must wait on work done by the working thread.
        std::thread repeater([&]){


        });
    // working thread
    std::thread worker([&]){
        std::unique_lock<std::mutex> lock(gLock);
    }).    result = 42 + 1 + 7;
                //Our work is done
                notified = true;
    return 0;
                std::this_thread::sleep_for(std::chrono
                                ::seconds(1);
```

unique_lock → more power than the lock_guard

—— × Incomplete ——×

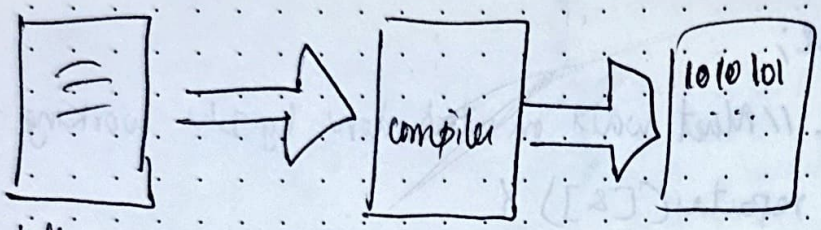
std :: async

↳ without synchronization

Launch a thread & wait be blocked on that thread.

Just like node.js async keyword

Compiler

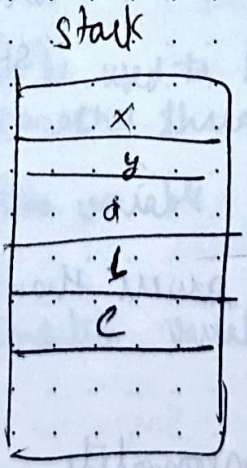


Segmentation fault → stack overflow

& "address of operator"

where actually in memory these variables live.

```
#include <iostream>
int main(){
    int x=42;
    float y=72;
    char a='a';
    signed char b='b';
    unsigned char c='c';
    return 0;
}
```



in DRAM

```
std::cout << "&x: " << &x << std::endl;
```

Address is returned in x & y in hexadecimal.

but

sizeof(x) → 4 bytes (to store an integer)

y → next location after x, after 4 bytes.

a: abc^a
b: bc^a
c: c^a
} strange right?

(void*) &a
(void*) &b
(void*) &c
} → now gives proper address

They are offset only by 1byte, as char takes only 1byte.

```
void foo(){
}
&foo
```

o/p: 1, strange.

So we do (void*) &foo → address of the function

You can think of & as a function itself.

Pass By Value

"arguments"

```
int main(){
  int x = 42
  std:: cout << "x value: " << x << "x address" << &x << std:: endl;
  setValue (x)
  std:: cout <<
  return 0
}
```

```
void setValue (int arg){
  arg = 999;   → std:: cout << arg << endl
  y            → std:: cout << arge << endl;
```

Output

x value 42, address 0x7ff...b7h

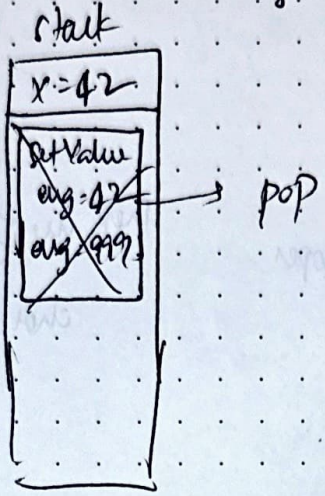
setValue : arg is initially 42

setValue : arg at end is 999.

x value 42, address same

"Make copies of our arguments when passed in function"

But arg address is diff than x address.



References

2nd use of "&" symbol

ALIAS

```
int& ref = x;
// "int&" is a full type for a reference type
cout << ref
cout << &ref
```

both x & ref give same value & same references.

→ Another variable name that can refer to x.

↔ Same named location memory as x.

Common use → pass by reference

```
#include <typeinfo>
typeid(x).name()
typeid(ref).name()
```

x type → i

ref type → i

Pass By Reference

1) Mutate Data
2) Efficient

```
void PassByReference(int& arg)
    arg = 999;
}

x = 42
cout << x
cout << &x
PassByReference(x);
cout << x
cout << &x
```

x address remain same

x value changes to 9999.

We avoided making a copy & mutating the same variable.

```
#include <vector>
#include <algorithm>
```

Pointers → A data type that stores an address.

```
int x = 7
int* px = &x;
cout << px;
```

What are we going to do with this value?

Dereferencing the value

Retrieve the value/type of what we point to.

"*px Dereferenced" << *px << std::endl;

↳ (7)

```
int main (){
    int x = 7;
    int* *px = &x;
    *px = 42;
    cout << *px;
```

o/p is 42.

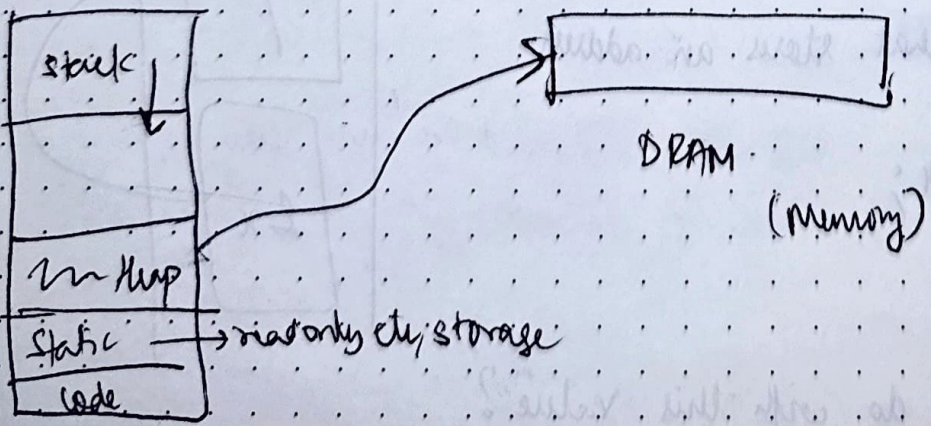
HEAP → new / delete

DYNAMIC MEMORY ALLOCATION

```
{
    int x = 7;
}
```

Block scope.

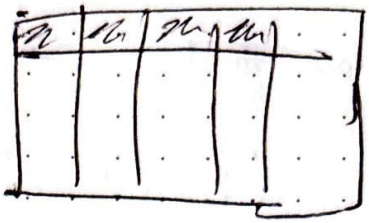
// x is not available here → as scope ends.



If we want to allocate memory

int* x = new int
↳ allocating memory

↳ int* → ptr

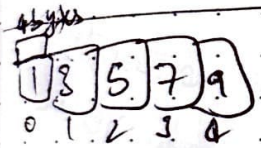


delete x;

```
int* AllocateMemory(){
    return new int;
}
int main(){
    int* x = AllocateMemory();
    delete x;
    return 0;
}
```

* Allows us to allocate resources at runtime.
* That resource is returned to us in a pointer.

delete[] studentids;
↳ deletes the array



int array[5]

int array[] = {1,3,5,7,9};

std::cout << array[0] << std::endl;

int* px = array; → points to first index of array

cout << px;
↳ gives memory of array.

dereferencing * px gives 1 (first element of the array)

px++ → goes up by 4 bytes

* px → gives 3.

// Show the array offset.

*(px+0) *(px+2) *(px+4) } → gives the full array.
*(px+1) *(px+3)

→ no need a pointer for the above as

*(array+0) *(array+2) *(array+4)
*(array+1) *(array+3)

array is by default points to the beginning.

array[0] } you are de-referencing the pointer
array[1] } It is the same as these

Passing arrays into functions

It is a good practise to pass arrays/vectors by reference to avoid copies.

```
void PrintArray(int arr[]){
    for(int i=0; i<

}

int main(){
    int array[] = {1,3,5,7};
    PrintArray(array);
    return 0;
}
```

```
sizeof array
    = sizeof(array)
      ---------------
      sizeof(int)
```

Pitfalls of pointers

(1)
```
int main(){
    int** Px = nullptr;
    *Px = 42;
    return 0;
}
```
Segmentation fault, dereferencing something which we do not have access to

(2) We forget to de-allocate memory
```
while (true){
    int* array = new int[1000];
}
return 0;
}
```
→ eventually there will be memory leak & we will run out of memory.

(3) Dangling pointer

That refers to a memory location that has already been freed or deleted but the pointer still holds the address of that location so dereferencing of that pointer can lead to undefined behaviour.

Eg -
```
int* ptr = new int(5) // Allocate memory
delete ptr
ptr = 20;
```
undefined

We are not resetting to NULL or nullptr so the pointer still holds the old address

```
int* getPointer(){
    int localVar = 10;
    return &localVar
}
```
Local vars in a fn. exist in stack memory & are destroyed once the fn returns.

④ double free

Deleting memory twice causes delete free.

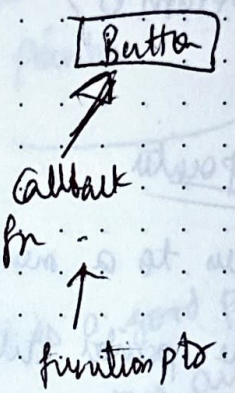
Freeing the same pointer twice

```
int* ptr = int*(malloc(sizeof(int)));
free(ptr)
free(ptr)
```

```
new int;
delete ptr
delete ptr
```

→ Function Pointers

```
int add(int x, int y){
    return x+y;
}

int multiply(int x, int y){
    return x*y;
}

int main(){
    int (*op)(int, int);
        ↳ name of fn pointer

    op = add;
    cout << op(2,2);
```



std::move in move semantics

efficient transfer ~~from~~ of resources from 1 object to another object.

std::string s1 = "long string..." → rvalue → no named storage or location

lvalue → location where we are going to store.

$2 = $1 ;

→ lvalue → locator value

↳ object that persists beyond a single expression. It has a memory address.

→ rvalue → temporary value that does not persist beyond the expression in which it is used. It is not associated with a memory address.

→ std::unique_ptr

SMART POINTERS

→ Raw pointers allows sharing → Allow room for errors (meaning forgetting to de-allocate)
→ Ownership

```
int x = 42;
int* ptr = &x;
int* ptr2 = ptr;

int* x = new int[100];
for (int i=0; i<100; i++){
    x[i] = i;
}
```

- *ptr = ~~auto~~ x → later say we delete x → error → & delete[] x;
- cout << *ptr → gives 0
- *ptr2 = ptr

Smart pointers address these issues.

(RAII - scoped)

```
#include <memory>   //unique_ptr

class UDT {
    public:
        UDT() { std::cout << "Const created"
        ~UDT() { std::cout << "destructo" << std::endl; }
};

int main() {
    std::unique_ptr<UDT> mike = std::unique_ptr<UDT>();
    return 0;
}
```

We dont have to think about deleting the memory created.

```
//creats an "array", that is pointed to by one unique pointer
std::unique_ptr<UDT[]> mike_array = std::unique_ptr<UDT[]>
                                    (new UDT[10]);
```

The equivalent to the line above is

```
std::unique_ptr<UDT[]> mike_array = std::make_unique<UDT[]>
                                    (10);
```

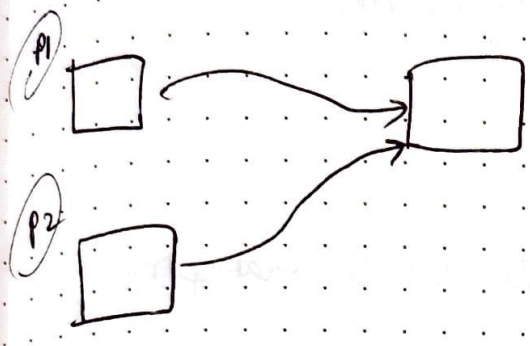
Factory Pattern

→ 2 unique pointers cannot point to the same box of memory

Unique pointer

1. Cannot Share
2. No copies
3. No worries about deleting (Prevents memory leak)
4. Move is allowed
5. Simplifies memory management.

→ shared_ptr



make sure updates.

→ unique_ptr was a scoped pointer

→ shared_ptr is a smart pointer that retains shared ownership of an object through pointers.

The object will be destroyed only when the last shared_ptr that points to it is destroyed (or) reset.

(make_shared)

use_count()

To count no: of pointers are pointing

```
shared_ptr<int> ptr1 =
    std::make_shared<int>(42);
{
std::shared_ptr<int> ptr2 = ptr1;
std::cout << *ptr2  // output 42
    ptr1.use_count() → 2
}
```

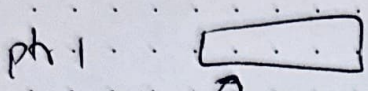
→ as block scope ended.

ptr1.use_count() → gives 1

Weak_ptr

A smart pointer (a non-owning)

(1) ~~Sort of acting~~ smart pointer but holds a non-owning pointer, does not increase the reference count.



(dangling ptr)

free ptr1

ptr2[0] → gives undefined.

So, dereferencing this is still allowed when we use weak_ptr.

~~std::weak_ptr<UDT> ptr2 = ptr1;~~

```
std::shared_ptr<UDT> ptr1 = std::make_shared<UDT>();
    {
std::weak_ptr<UDT> ptr2 = ptr1
    ptr2.use_count() → still gives 1.
    }
ptr1.use_count() → gives 1.
```

Interface (.hpp or .h) and Implementation (.cpp, .cxx, .c)

#include "mikemath.hpp" → our local path.

#include <iostream> → present in some system filepath

~~class Student~~

{