Non type template params

```
std::array<int,10> a1;
std::array<float,8> a2;

std::get<2>(t1)

std::shared_ptr<A> pa;
pa = std::shared_ptr<A>(new B(11,12));

std::shared_ptr<B> pb;
pb=pa;                          //downcasting
pb = dynamic_pointer_cast<B>(pa);
```

Namespaces
File Handling ==> ifstream, ofstream, ios (seekg)

```
class Compute {
  public:
  operator() () {
  }
}
```

Concurrency & IPC:-

Parallel Computation:-
- std::thread
- std::async

Threads
Semaphores
Mutex
Prod-Cons problem
Deadlock

Threads:-
* Resource Sharing (except stack)
* Concurrent execution

Task Driven Parallelism
Data Driven Parallelism, e.g. parallel sum of large array

std::thread constructor
==> Normal functions
==> Lambda expressions
==> binded functions
==> Function Objects
==> Member functions, hint:- std::bind

TODO:- parallel sum of large array

IPC:-

Key terms:-
* Race conditions
* Critical Section, Entry Section, Exit Section
* Mutual Exclusion

Solutions:-
* Semaphores (no C++ support)
* Mutex
* Spinlocks (no C++ support)
* Atomic variables (C++ objects)

Signalling/Synchronization:-
* Semaphores (no C++ support)
* Condition variables

Solution-1:- (example7.cpp)
std::mutex m1;
m1.lock();
m1.unlock();

Solution-2:- (example8.cpp)
std::atomic<int> val(100);
---
lock_and_set, XCHG, SWP

reg=0 //init

T2

while(XCHG(reg,1)); //enry
while(XCHG(reg,1)); //busy loop
//spinning

//critical section
//critical section

reg=0 //exit
reg=0

busy loop based solutions like spinlocks are meaningful for SMP only (multicore)

```
std::mutex m1;

std::unique_lock<std::mutex> ulck(m1);
//(or)
std::lock_guard<std::mutex> ulck(m1);
```

example6 -- race cond demo (val++, val--)
7 - avoid race cond is mutex
8 - atomic var
9 - prevent loop overlap using mutex
10a/10b - unique_lock / lock_guard

Dead lock scenario:-

```
std::mutex m1;
std::mutex m2;

T1                      T2

m1.lock();              m2.lock();
//delay                 m1.lock();
m2.lock();
```

solution:-

```
std::unique_lock<std::mutex> u1
                        (m1, std::defer_lock);
std::unique_lock<std::mutex> u2
                        (m2, std::defer_lock);
std::lock(u1,u2);
```

Further topics:-
std::condition_variables
std::async
std::future
std::promise

Activity:-
* Post read of covered topics (thread, mutex, locks etc)
* Pre-read of next topics
* Coding Tasks
* File Handling, Namespaces (if pending)
* Exception Handling, give a try