* ***Purpose :*** **Classwork.**

* ***Date :*** **08/01/2026**

* ***Author :*** **Vikas Srivastava**

* ***ID :*** **55984**

* ***Batch ID :*** **25SUB4505**

---

1. Code : To demonstrate a data race condition in multithreading and explain how multiple threads accessing shared data without synchronization can lead to inconsistent results.

```
#include <iostream>
#include <thread>
using namespace std;
int globVar = 0;
void Update(){
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
}
int main(){
	thread t1(Update);
	thread t2(Update);
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	t1.join();
	t2.join();
	cout<<"global var: "<<globVar<<endl;
}
```

2. Code : To demonstrate the use of lambda expressions in C++ for creating inline functions executed by threads.

```
#include <iostream>
using namespace std;
int main(){
	auto myLambda = [] {cout<<"Hello Welcome to Lambdas in C++"<<endl; };
	myLambda();
}
```

3. Code : To demonstrate capturing variables inside lambda expressions and using them in multithreaded execution.

```
#include <iostream>
using namespace std;
int main(){
	[] {cout<<"Hello Welcome to Lambdas in C++"<<endl; }();
}
```

4. Code : To demonstrate advanced lambda usage with threads, including parameter passing and scoped variable access.

```
#include <iostream>
using namespace std;
int main(){
	int var = 100; //local variable
	auto myLambda = [&var] {
		cout<<"Hello Welcome to Lambdas in C++: "<<var<<endl;
		var+=10;
	};
	myLambda();
	cout<<"after updation in Main(): "<<var<<endl;
}
```

5. Code : To demonstrate the use of a mutex to protect shared resources and prevent race conditions in a multithreaded program.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
int globVar = 0;
mutex mVar;
void Update(){
	mVar.lock(); //resource is locked
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	mVar.unlock();//resource is unlocked
}
int main(){
	thread t1(Update);
```

```
	thread t2(Update);
	mVar.lock();
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	mVar.unlock();
	t1.join();
	t2.join();
	cout<<"global var: "<<globVar<<endl;
}
```

6. Code : To demonstrate mutual exclusion using std::mutex when multiple threads modify shared data.

```
/*
 *	If lock() is established, and if not unlocked()
 *		then, it results in deadlock --> your program will not
 *		continue further
 */
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
int globVar = 0;
mutex mVar; // this is an important resource
	//It has to be wrapped around in a class
	//RAII
void Update(){
	mVar.lock(); //resource is locked
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	mVar.unlock(); //resource is locked
}
int main(){
	thread t1(Update);
	thread t2(Update);
	mVar.lock();
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	mVar.unlock();
	t1.join();
	t2.join();
	cout<<"global var: "<<globVar<<endl;
}
```

7. Code : To demonstrate safe locking and unlocking of mutexes, ensuring proper synchronization between threads.

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
int globVar = 0;
mutex mVar;
void Update(){
	lock_guard<mutex> lock(mVar);
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
}
int main(){
	thread t1(Update);
	thread t2(Update);
	thread t3(Update);
	t1.join();
	t2.join();
	t3.join();
	cout<<"global var: "<<globVar<<endl;
}
```

8. Code : To demonstrate a multithreading problem scenario, focusing on correct synchronization and controlled thread execution.

```
/*
 *	Creating 5 thread and joining them
 */
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
void funTask(){
	cout<<"Thread created..."<<endl;
}
int main(){
	vector<thread> threadObjs;
	for(int cnt=0; cnt < 5; cnt++)
		threadObjs.emplace_back(thread(funTask));
	for (auto& tId: threadObjs)
```

```
        tId.join();
}
```

9. Code : To demonstrate thread execution order and behavior when multiple threads are running simultaneously.

```
#include <iostream>
#include <thread>
int main(){
        std::thread t1([] {
                std::cout<<"Inside lambda here"<<std::endl;
        }); //thread object t1 is created by passing lambda expression.
        std::cout<<"In main()"<<std::endl;
        t1.join();
}
```

10. Code : To demonstrate concurrent thread execution and observe how thread scheduling affects program output.

```
#include <iostream>
#include <thread>
class Test{
public:
        void operator()(){
                std::cout<<"Thread from Test"<<std::endl;
        }
};
int main(){
        Test obj = Test();
        std::thread t1(obj); //thread object t1 is created.
        std::cout<<"Thread from main()"<<std::endl;
        t1.join();
}
```

11. Code : To introduce **basic thread creation in C++** using the std::thread class and demonstrate concurrent execution of a function.

```
/*
 *      Sequential execution
 *
 */
#include <iostream>
```

```
#include <thread>
#include <chrono>
using namespace std;
void fun() {
   cout << "fun() called" << endl;
   std::this_thread::sleep_for(std::chrono::seconds(1));
}
int main() {
   fun();
   fun();
   fun();
   fun();
   fun();
}
```

12. Code : To demonstrate **multiple thread creation** and show how different threads run independently in parallel.

```
/*
 *          Concurrent execution
 *
 */
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
void fun(){
          cout<<"fun() called"<<endl;
          this_thread::sleep_for(chrono::seconds(1));
}
int main(){
          thread t1(fun);
          thread t2(fun);
          thread t3(fun);
          thread t4(fun);
          thread t5(fun);
          t1.join();
          t2.join();
          t3.join();
          t4.join();
          t5.join();
}
```

13. Code : To demonstrate **thread management using join()**, ensuring the main thread waits for the child thread to complete execution.

```
#include <iostream>
#include <thread>
void funOne(){
	for (int cnt=0;cnt<100;cnt++){
		std::cout<<"0";
	}
}
int main(){
	std::thread t1(funOne); //thread object t1 is created.
	for (int cnt=0;cnt<100;cnt++){
		std::cout<<"X";
	}
	t1.join(); //waiting for thread t1 to complete
}
```

14. Code : To demonstrate **thread management using detach()**, allowing a thread to run independently of the main thread.

```
#include <iostream>
#include <thread>
void funOne(){
	for (int cnt=0;cnt<100;cnt++){
		std::cout<<"0";
	}
}
int main(){
	std::thread t1(funOne); //thread object t1 is created.
	t1.detach(); //waiting for thread t1 to complete
	for (int cnt=0;cnt<100;cnt++){
		std::cout<<"X";
	}
}
```

15. Code : To demonstrate **basic multithreading** by executing a single function concurrently using a separate thread.

```
#include <iostream>
#include <thread>
void funOne(){
```

```
        while (1){
                std::cout<<"0";
        }
}
int main(){
        std::thread t1(funOne); //thread object t1 is created.
        while(1){
                std::cout<<"X";
        }
}
```

16. Code : To demonstrate **passing arguments to a thread function** in C++.

```
#include <iostream>
#include <thread>
void funOne(){
        for (int cnt=0;cnt<100;cnt++){
                std::cout<<"0";
        }
}
int main(){
        std::thread t1(funOne); //thread object t1 is created.
        for (int cnt=0;cnt<100;cnt++){
                std::cout<<"X";
        }
        t1.join(); //waiting for thread t1 to complete
}
```

17. Code : To demonstrate **using member functions or lambda expressions** with threads for concurrent execution.

```
#include <iostream>
#include <thread>
void funOne(){
        std::cout<<"Thread here "<<std::endl;
}
int main(){
        std::thread t1(funOne); //thread object t1 is created.
        std::cout<<"In main()"<<std::endl;
        t1.join();
}
```

18. Code : To demonstrate **multiple threads executing simultaneously** and highlight the effects of concurrent output execution.

```
#include <iostream>
#include <thread>
using namespace std;
void funOne(int arg){
        std::cout<<"New Thread: "<<arg<<endl;
}
int main(){
        std::thread t1(funOne, 100);
        std::cout<<"Main Thread: "<<endl;
        t1.join();
}
```