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

* ***Date :*** **09/01/2026**

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

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

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

---

1. *Objective:*

To demonstrate how a **condition variable** is used in C++ to synchronize threads, allowing one thread to wait until another thread signals that a condition has been met.

Code:

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

mutex mVar;
condition_variable cVar;
bool ready = false;

void funOne(){
        unique_lock<mutex> lock(mVar);

        cVar.wait(lock, [] {return ready; });

        cout<<"Recieved Notification.. Proceeding now "<<ready<<endl;
}

int main(){
        thread t1(funOne);
        this_thread::sleep_for(chrono::seconds(2));
```

```
        {
                lock_guard<mutex> lock(mVar);
                ready = true;
        }

        cVar.notify_one();

        t1.join();
}
```

2. *Objective:*

To demonstrate the use of **std::future** in C++ for retrieving results from a function executed asynchronously in a separate thread.

Code:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <future>
using namespace std;

void calculateSum(promise<int> prom){
        int sum = 0;
        for (int i = 1; i <= 10; i++){
                this_thread::sleep_for(chrono::seconds(1));
                sum+=i;
        }
        prom.set_value(sum);//result is produced here
}

int main(){
        promise<int> pObj;
        future<int> fObj = pObj.get_future();

        thread t1(calculateSum, move(pObj));

        int result = fObj.get(); //wait until result is ready
        cout<<"Result :"<<result<<endl;
```

```
        t1.join();
}
```

3. *Objective:*

To demonstrate thread execution using **local variables**, highlighting how data behaves when shared or accessed across multiple threads.

Code:

```
/*
 *        Filling an array of 100 elements with 5 threads
 *
 *                Using global array variable
 *
 */
#include <iostream>
#include <thread>
#include <vector>

// Global array of 100 elements
int arr[100];

// Thread function
void fillArray(int startIndex, int count)
{
   for (int i = startIndex; i < startIndex + count; i++) {
      arr[i] = 100 + i;   // Fill with thread ID for visibility
   }
}

int main()
{
   const int numThreads = 5;
   const int elementsPerThread = 20;

   std::vector<std::thread> threads;
```

```
    // Create 5 threads
    for (int i = 0; i < numThreads; i++) {
        threads.emplace_back(
            fillArray,
            i * elementsPerThread,   // start index
            elementsPerThread
        );
    }

    // Wait for all threads to complete
    for (auto& t : threads) {
        t.join();
    }

            std::cout<<"Array: ";
    // Verify the result
    for (int i = 0; i < 100; i++) {
        std::cout << arr[i] << " ";
    }
            std::cout<<std::endl;
    return 0;
}
```

4. Objective:

To demonstrate **heap memory allocation** in multithreading and show how dynamically allocated memory can be shared safely between threads.

Code:

```
/*
 * using heap based array (Low level)
 *
 *
 */

#include <iostream>
#include <thread>
#include <vector>

// Thread function
```

```
void fillArray(int* arr, int startIndex, int count)
{
   for (int i = startIndex; i < startIndex + count; i++) {
      arr[i] = 100 + i;
   }
}

int main()
{
   //Heap array (shared across threads)
   int *arr = new  int[100];

   const int numThreads = 5;
   const int elementsPerThread = 20;

   std::vector<std::thread> threads;

   // Create threads
   for (int i = 0; i < numThreads; i++) {
      threads.emplace_back(
         fillArray,
         arr,                    // pointer to heap array
         i * elementsPerThread,    // start index
         elementsPerThread       // number of elements
      );
   }

   // Join threads
   for (auto& t : threads) {
      t.join();
   }

   // Print array contents
           std::cout<<"Array: ";
   for (int i = 0; i < 100; i++) {
      std::cout << arr[i] << " ";
   }
           std::cout<<std::endl;

           delete []arr;

   return 0;
```

```
}
```

5. Objective:

To demonstrate an alternative approach to **heap memory usage in multithreading**, emphasizing proper synchronization and memory handling.

Code:

```
/*
 * using heap based array (Low level)
 *
 *
 */

#include <iostream>
#include <thread>
#include <vector>
using namespace std;

// Thread function
void fillArray(int* arr, int startIndex, int count)
{
   for (int i = startIndex; i < startIndex + count; i++) {
      arr[i] = 100 + i;
   }
}

int main()
{

   auto arr = make_unique<int[]>(100); //unique_ptr

   const int numThreads = 5;
   const int elementsPerThread = 20;

   vector<thread> threads;
```

```
    // Create threads
    for (int i = 0; i < numThreads; i++) {
        threads.emplace_back(
            fillArray,
            arr.get(),  //using get() pointer to heap array
            i * elementsPerThread,    // start index
            elementsPerThread       // number of elements
        );
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    // Print array contents
            cout<<"Array: ";
    for (int i = 0; i < 100; i++) {
        cout << arr[i] << " ";
    }
            cout<<endl;

    return 0;
}
```

6. ***Objective:***

   To demonstrate the behavior of **local variables in multithreaded programs** and how each thread maintains its own copy of local data.

Code:

```
/*
 * using local static array
 *
 *
 */

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

```
// Thread function
void fillArray(int* arr, int startIndex, int count)
{
   for (int i = startIndex; i < startIndex + count; i++) {
      arr[i] = 100 + i;
   }
}

int main()
{
   // Static array (shared across threads)
   static int arr[100];

   const int numThreads = 5;
   const int elementsPerThread = 20;

   std::vector<std::thread> threads;

   // Create threads
   for (int i = 0; i < numThreads; i++) {
      threads.emplace_back(
         fillArray,
         arr,                  // pointer to static array
         i * elementsPerThread,    // start index
         elementsPerThread       // number of elements
      );
   }

   // Join threads
   for (auto& t : threads) {
      t.join();
   }

   // Print array contents
           std::cout<<"Array: ";
   for (int i = 0; i < 100; i++) {
      std::cout << arr[i] << " ";
   }
           std::cout<<std::endl;

   return 0;
}
```

7. Objective:

To demonstrate the use of **STL vectors in a multithreaded environment** and highlight the need for synchronization when multiple threads access shared containers.

Code:

```
/*
 * using vector<int> from STL
 *
 */

#include <iostream>
#include <thread>
#include <vector>
using namespace std;

// Thread function
void fillArray(vector<int> &arr, int startIndex, int count)
{
   for (int i = startIndex; i < startIndex + count; i++) {
      arr[i] = 100 + i;
   }
}

int main()
{
   // vector (shared across threads)
   vector<int> vArr(100);

   const int numThreads = 5;
   const int elementsPerThread = 20;
```

```
    vector<thread> threads;

    // Create threads
    for (int i = 0; i < numThreads; i++) {
        threads.emplace_back(
            fillArray,
            ref(vArr),                  // pointer to heap array
            i * elementsPerThread,    // start index
            elementsPerThread       // number of elements
        );
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    // Print array contents
            cout<<"Array: ";
    for (int i : vArr) {
        cout << i << " ";
    }
            cout<<endl;

    return 0;
}
```

8. Objective:

To demonstrate the use of **std::shared_future**, which allows **multiple threads** to access the result of a single asynchronous operation.

Code:

```
#include <iostream>
#include <thread>
#include <future>
using namespace std;

void calculateSum(promise<int> prom)
{
    int sum = 0;
```

```
    for (int i = 1; i <= 10; i++) {
        sum += i;
    }
    cout << "Sum calculated\n";
    prom.set_value(sum);
}

void readResult(shared_future<int> sf, int id)
{
    cout << "Reader " << id
         << " got sum = " << sf.get() << endl;
}

int main()
{
    promise<int> p;
    shared_future<int> sf = p.get_future().share();
    thread producer(calculateSum, move(p));

    thread t1(readResult, sf, 1);
    thread t2(readResult, sf, 2);
    thread t3(readResult, sf, 3);

    producer.join();
    t1.join();
    t2.join();
    t3.join();
}
```

9. Objective:

To demonstrate the implementation of a **basic thread pool**, where multiple worker threads execute tasks from a shared task queue to improve performance and resource utilization.

Code:

```
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
```

```
#include <functional>
#include <chrono>
using namespace std;


class SimpleThreadPool{
	vector<thread> workers; //stores thread objects
	queue<function<void()>> tasks;

	mutex mLock;
	condition_variable cLock;
	bool stop = false;

public:
	SimpleThreadPool(unsigned int num){
		for (unsigned int i = 0; i < num; ++i){
			workers.emplace_back( [this] {
				while (true){
					function<void ()> task;
					{
						unique_lock<mutex> lock(mLock);
						cLock.wait( lock, [this] {
								return stop | !tasks.empty();
							});

						if (stop && tasks.empty())
							return;

						task = move(tasks.front());
						tasks.pop();
					}

					task();
				}

			});
		}

	}

	void submit(function<void ()> task){
		{
			lock_guard<mutex> lock(mLock);
```

```
                tasks.push(task);
            }
            cLock.notify_one();
        }

        ~SimpleThreadPool(){
            {
                lock_guard<mutex> lock(mLock);
                stop = true;
            }
            cLock.notify_all();
            for(auto &t: workers)
                t.join();
        }
};

int main(){
        SimpleThreadPool pool(3);

        for (int i = 0; i< 6; ++i){
            pool.submit([i] {
                cout<<"Task "<<i+1
                <<" executed by thread "
                << this_thread::get_id()<<"\n";
                this_thread::sleep_for(chrono::seconds(1));
            });
        }
}
```

10. Objective:

To demonstrate the use of **std::unique_lock** for flexible mutex locking and unlocking, especially useful with condition variables and complex locking scenarios.

Code:

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

int globVar = 0;
mutex mVar;

void Update(){
	unique_lock<mutex> lock(mVar);
	for (int cnt=0;cnt < 100000; cnt++)
		globVar++;
	lock.unlock();
}

int main(){
	thread t1(Update);
	thread t2(Update);
	thread t3(Update);

	t1.join();
	t2.join();
	t3.join();

	cout<<"global var: "<<globVar<<endl;
}
```