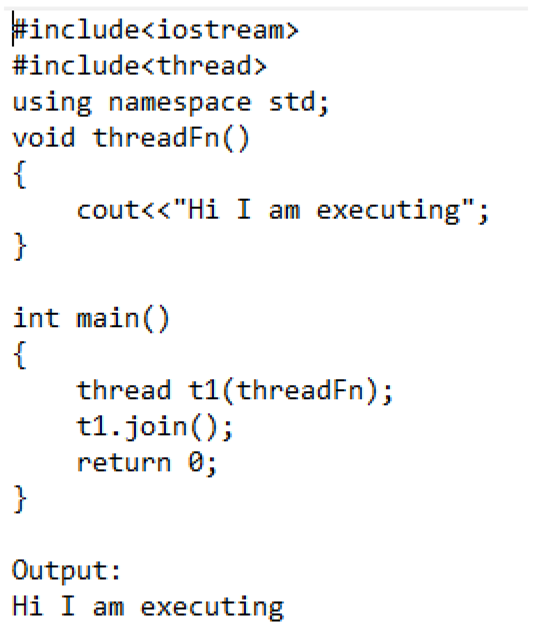
Threads

Every process starts with a main thread. We can create multiple threads along with it.

Syntax:

#include<thread>	-- Library to be linked
thread <thread_name>(func_name)	-- declare a thread with a call to func_name()
thread <thread_name>{func_name}	-- same as above
thread <thread_name>{func_name, arg1, arg2}	-- threading by passing arguments
thread <thread_name>{func_name, ref(arg1)}	-- Passing arguments by reference
<thread_name>.join()	-- main() func waits for the thread to complete and join.
this_thread::get_id()	-- returns the thread id.
this_thread::sleep_for(time)	-- delay the execution for the sleep time

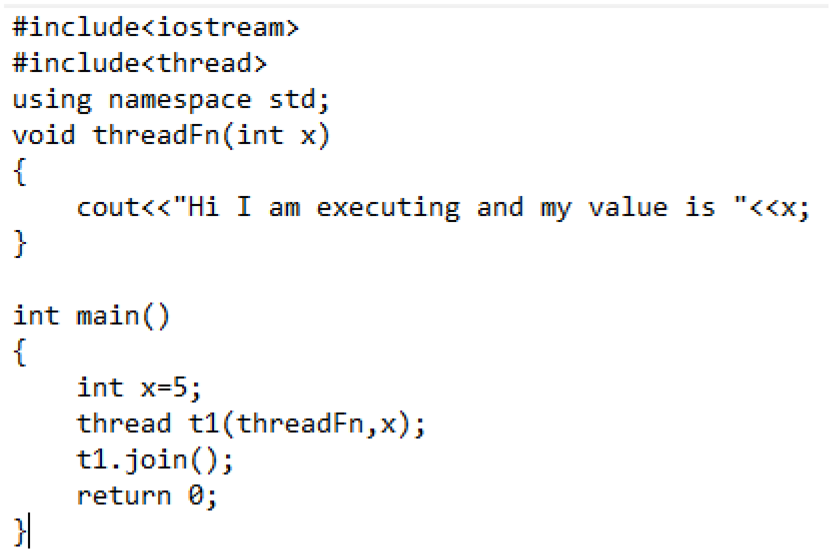
Simple Demonstration



```
#include<iostream>
#include<thread>
using namespace std;
void threadFn()
{
    cout<<"Hi I am executing";
}

int main()
{
    thread t1(threadFn);
    t1.join();
    return 0;
}
```

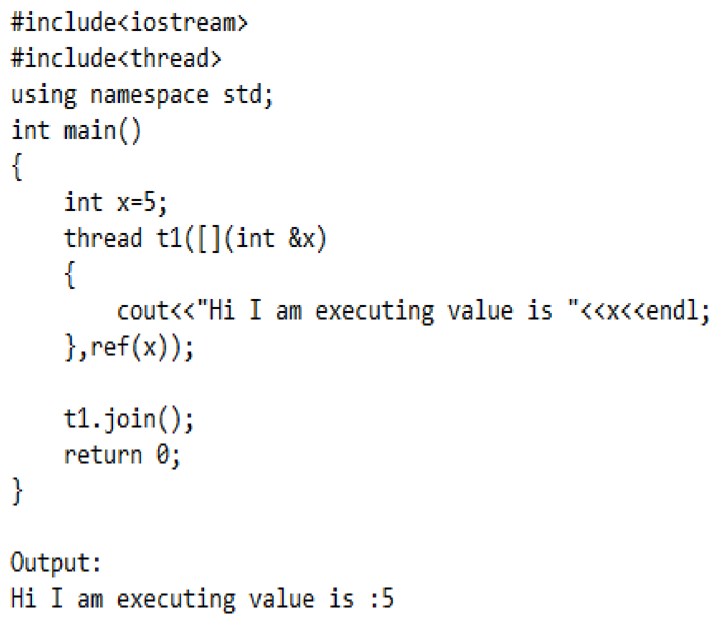
```
Output:
Hi I am executing
```

Calling Thread Function by passing parameters



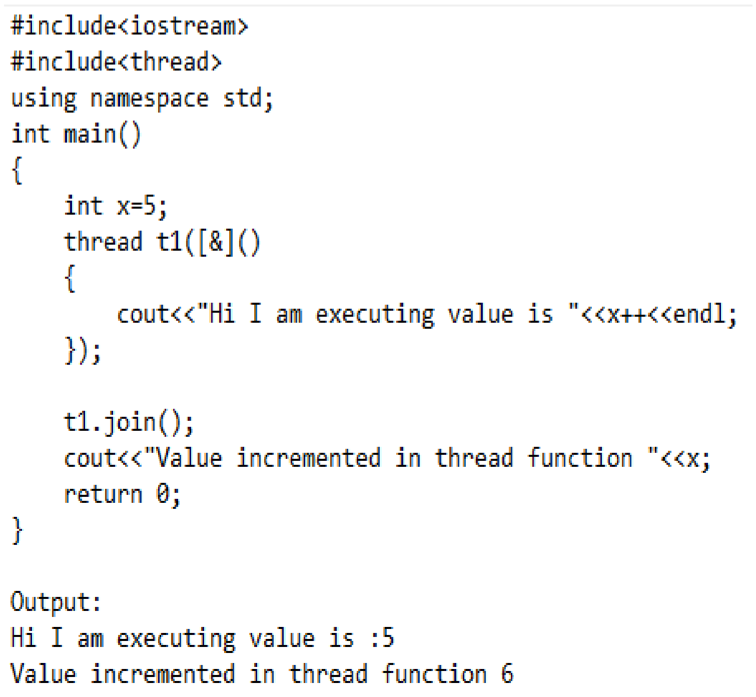
```
#include<iostream>
#include<thread>
using namespace std;
void threadFn(int x)
{
    cout<<"Hi I am executing and my value is "<<x;
}

int main()
{
    int x=5;
    thread t1(threadFn,x);
    t1.join();
    return 0;
}
```

Writing thread function in declaration



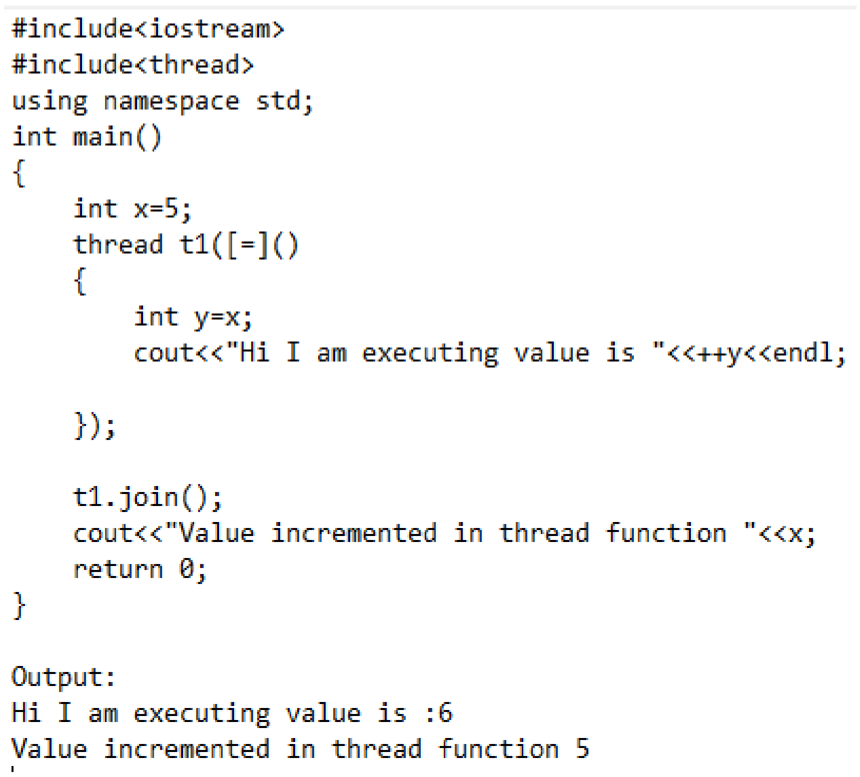
```
#include<iostream>
#include<thread>
using namespace std;
int main()
{
    int x=5;
    thread t1([](int &x)
    {
        cout<<"Hi I am executing value is "<<x<<endl;
    },ref(x));

    t1.join();
    return 0;
}

Output:
Hi I am executing value is :5
```

Passing all local variables to the thread function as reference



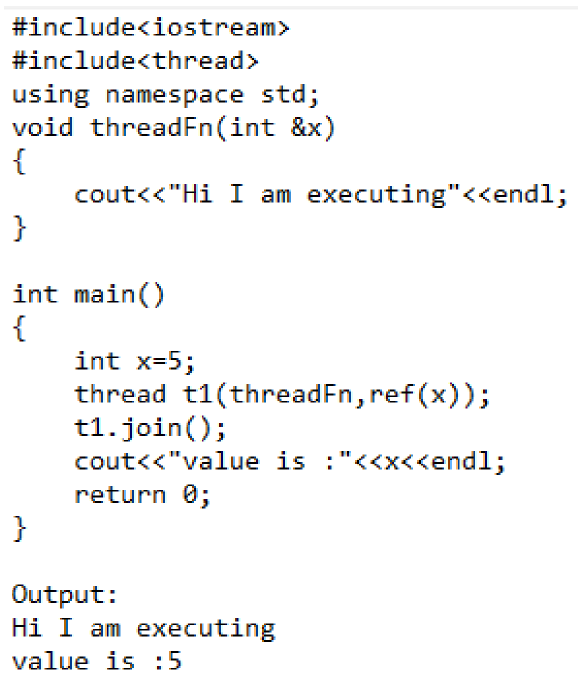
```
#include<iostream>
#include<thread>
using namespace std;
int main()
{
    int x=5;
    thread t1([&]()
    {
        cout<<"Hi I am executing value is "<<x++<<endl;
    });

    t1.join();
    cout<<"Value incremented in thread function "<<x;
    return 0;
}

Output:
Hi I am executing value is :5
Value incremented in thread function 6
```

Passing local variables but not as reference – cannot alter main function values



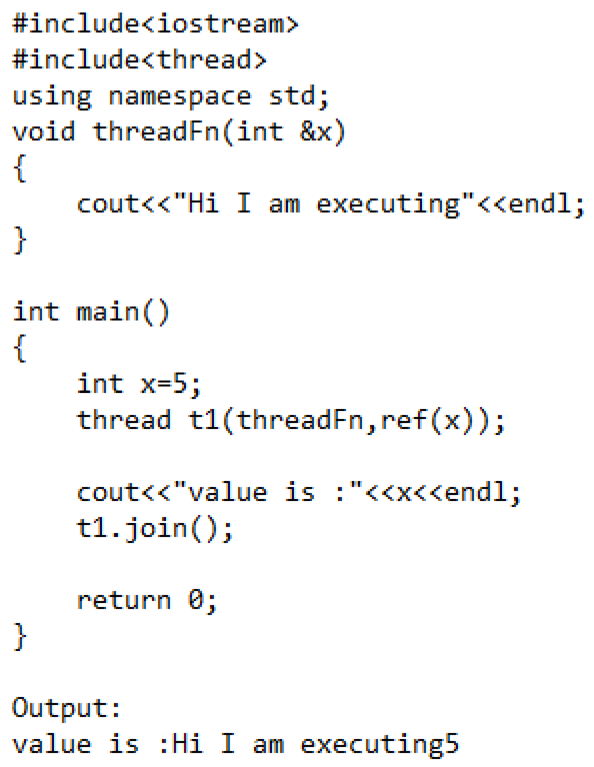
```
#include<iostream>
#include<thread>
using namespace std;
int main()
{
    int x=5;
    thread t1([=]()
    {
        int y=x;
        cout<<"Hi I am executing value is "<<++y<<endl;

    });

    t1.join();
    cout<<"Value incremented in thread function "<<x;
    return 0;
}

Output:
Hi I am executing value is :6
Value incremented in thread function 5
```

Using join – waits the thread to complete execution



```
#include<iostream>
#include<thread>
using namespace std;
void threadFn(int &x)
{
    cout<<"Hi I am executing"<<endl;
}

int main()
{
    int x=5;
    thread t1(threadFn,ref(x));
    t1.join();
    cout<<"value is :"<<x<<endl;
    return 0;
}

Output:
Hi I am executing
value is :5
```

Using join at incorrect places



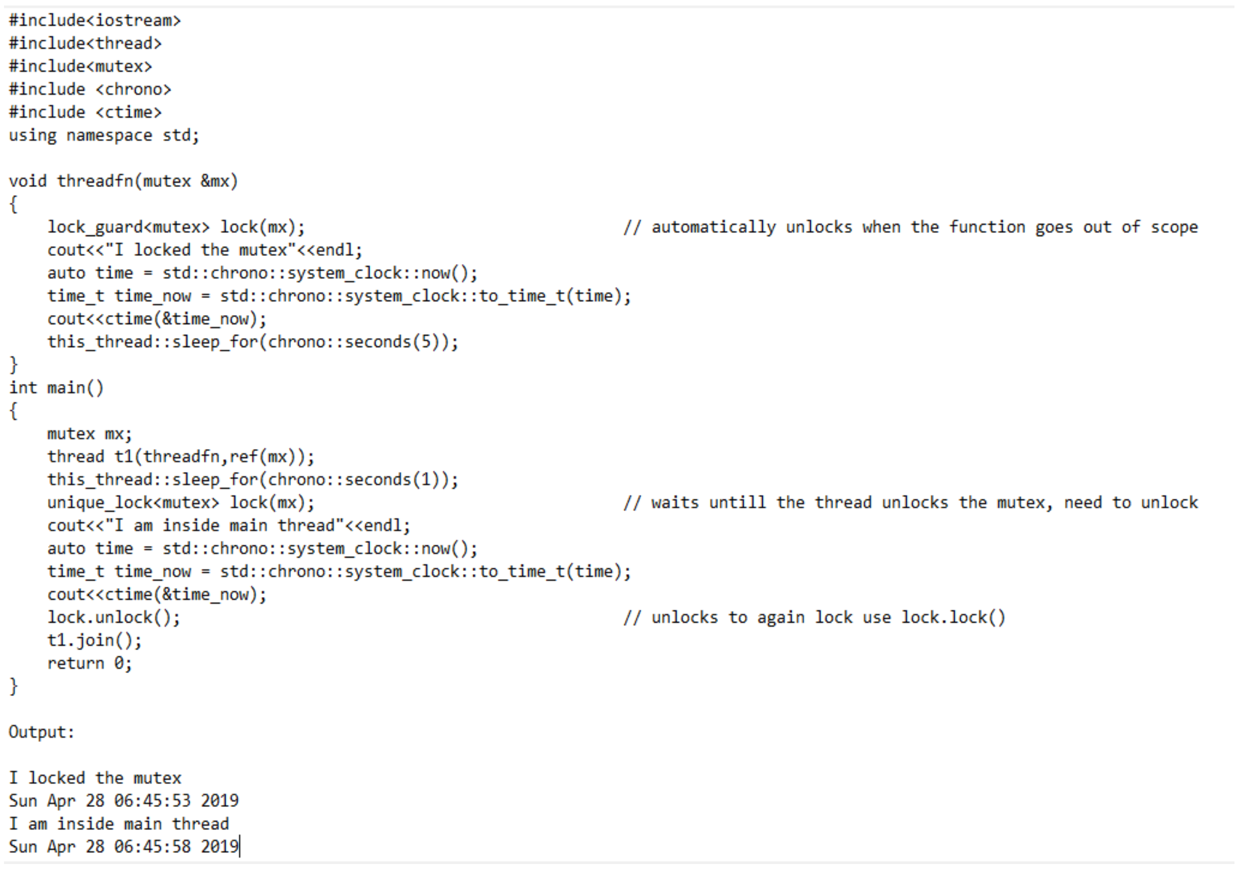
```
#include<iostream>
#include<thread>
using namespace std;
void threadFn(int &x)
{
    cout<<"Hi I am executing"<<endl;
}

int main()
{
    int x=5;
    thread t1(threadFn,ref(x));

    cout<<"value is :"<<x<<endl;
    t1.join();

    return 0;
}

Output:
value is :Hi I am executing5
```

Packages supporting multithreading

Future

Promise

Mutex lock



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

void threadfn(mutex &mx)
{
    lock_guard<mutex> lock(mx);                         // automatically unlocks when the function goes out of scope
    cout<<"I locked the mutex"<<endl;
    auto time = std::chrono::system_clock::now();
    time_t time_now = std::chrono::system_clock::to_time_t(time);
    cout<<ctime(&time_now);
    this_thread::sleep_for(chrono::seconds(5));
}
int main()
{
    mutex mx;
    thread t1(threadfn,ref(mx));
    this_thread::sleep_for(chrono::seconds(1));
    unique_lock<mutex> lock(mx);                        // waits untill the thread unlocks the mutex, need to unlock
    cout<<"I am inside main thread"<<endl;
    auto time = std::chrono::system_clock::now();
    time_t time_now = std::chrono::system_clock::to_time_t(time);
    cout<<ctime(&time_now);
    lock.unlock();                                      // unlocks to again lock use lock.lock()
    t1.join();
    return 0;
}

Output:

I locked the mutex
Sun Apr 28 06:45:53 2019
I am inside main thread
Sun Apr 28 06:45:58 2019
```

Difference between lock_guard and unique_lock?

lock_guard → automatically unlocks once the function goes out of scope. Cannot be used with condition variables.

unique_lock → needs to be unlocked and can be locked and unlocked. Used with condition variables.

Mutex with condition variables

https://www.youtube.com/watch?v=kdXGTGveme8