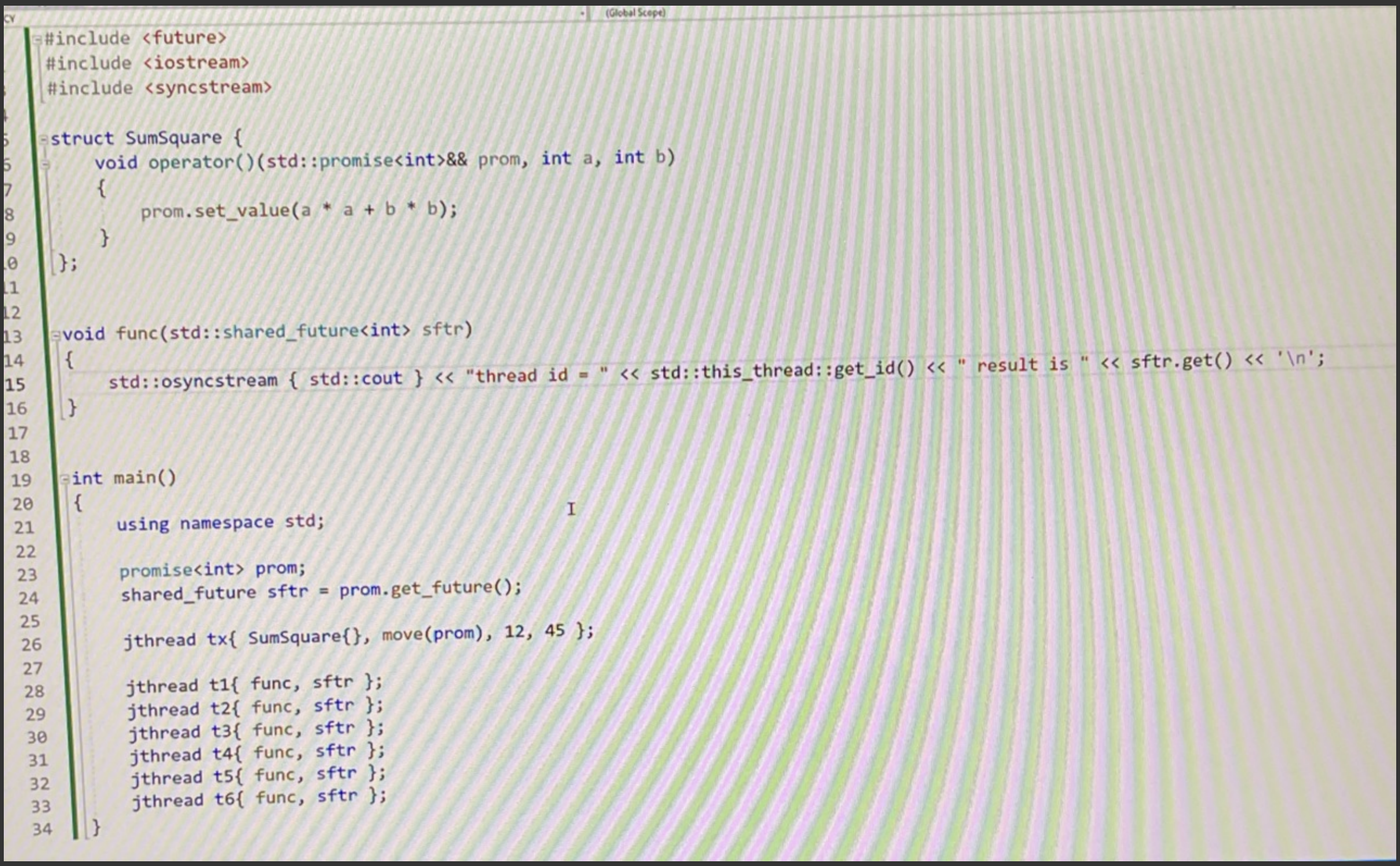
#Shared Future:

- Kopyalanabilir
- Birden fazla get edilebilir

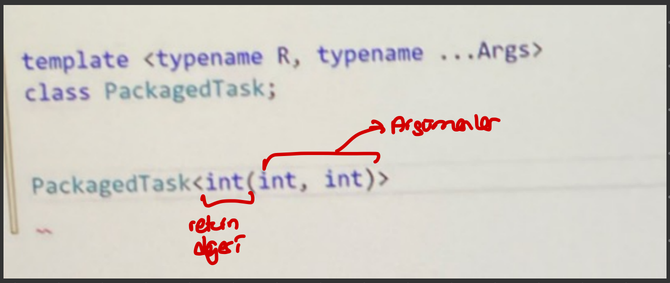


```cpp
#include <future>
#include <iostream>
#include <syncstream>

struct SumSquare {
    void operator()(std::promise<int>&& prom, int a, int b)
    {
        prom.set_value(a * a + b * b);
    }
};

void func(std::shared_future<int> sftr)
{
    std::osyncstream { std::cout } << "thread id = " << std::this_thread::get_id() << " result is " << sftr.get() << '\n';
}

int main()
{
    using namespace std;

    promise<int> prom;
    shared_future sftr = prom.get_future();

    jthread tx{ SumSquare{}, move(prom), 12, 45 };

    jthread t1{ func, sftr };
    jthread t2{ func, sftr };
    jthread t3{ func, sftr };
    jthread t4{ func, sftr };
    jthread t5{ func, sftr };
    jthread t6{ func, sftr };
}
```

#Packaged Task:

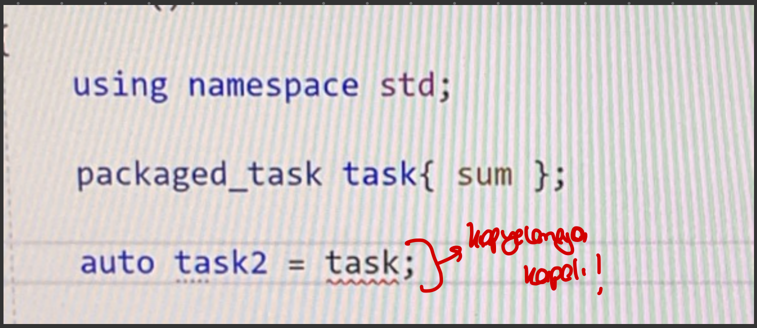
- Bir task'i daha sonra çağırılabilecek ve farklı threadlere aktarabildiğimiz, bir callable sarmalayıcısı
- return değerimiz future nesnesi
- bu bir sınıf ve operator()'a sahip → sarmaladığı callable'ı çağırır!

* #include <future>



```cpp
template <typename R, typename ...Args>
class PackagedTask;

PackagedTask<int(int, int)>
```

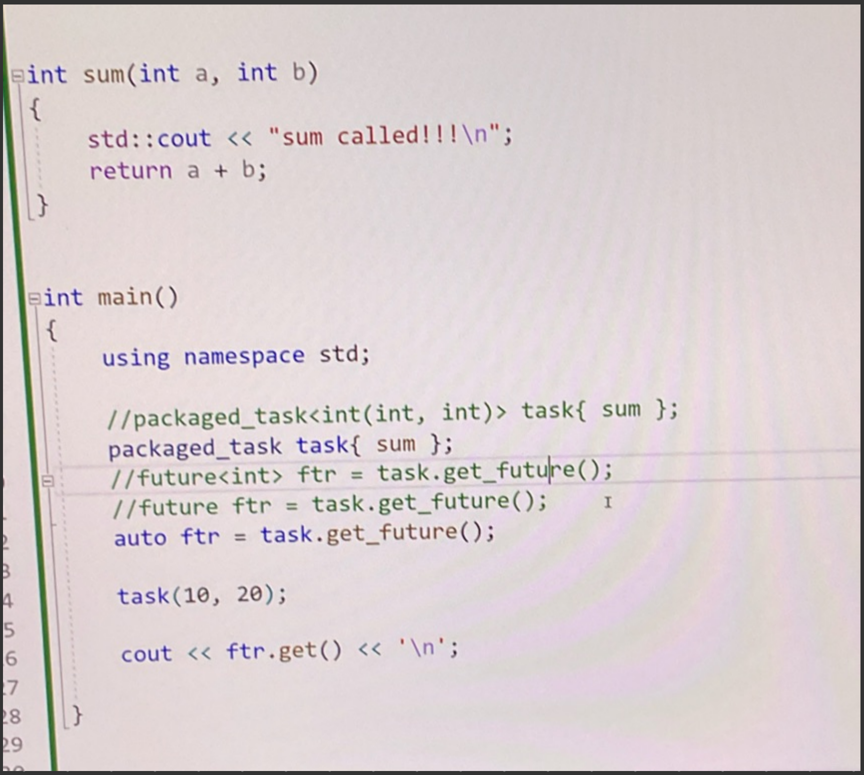
$\underbrace{\text{int}}_{\text{return değeri}}(\overbrace{\text{int, int}}^{\text{Argümanlar}})$



```cpp
using namespace std;

packaged_task task{ sum };

auto task2 = task;
```

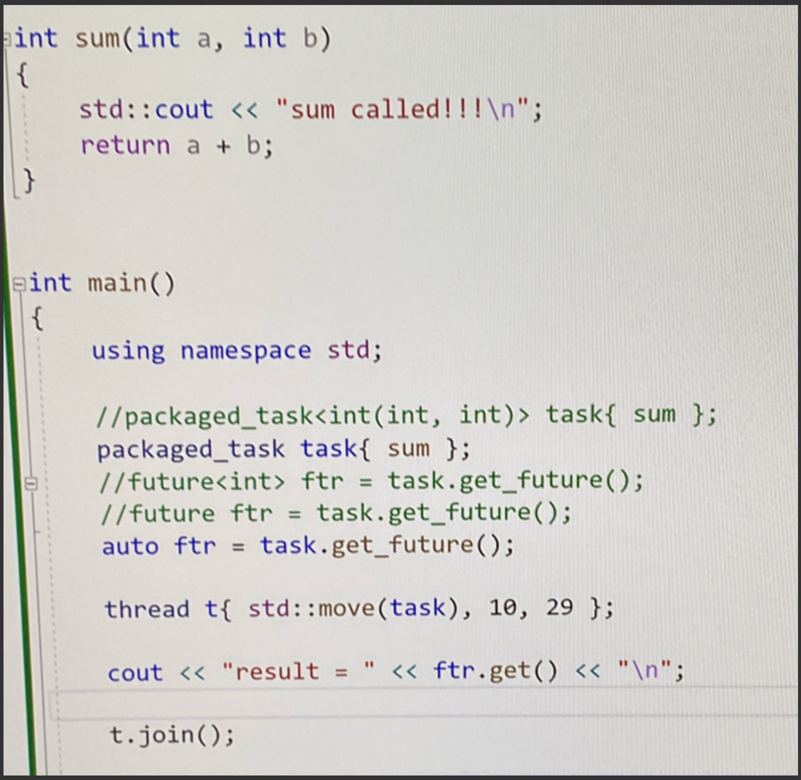
→ kopyalamaya kapalı!

=> Fakat move edilebilir.



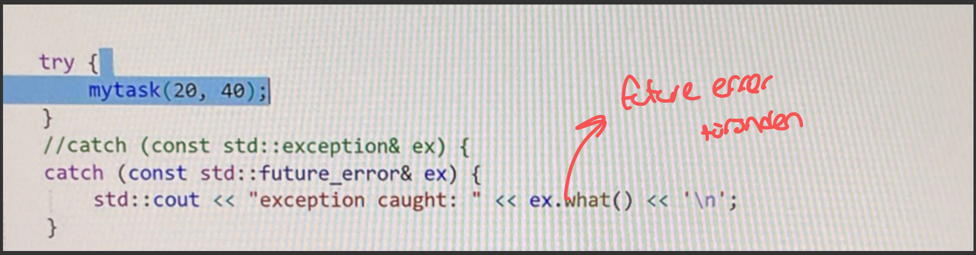
```cpp
int sum(int a, int b)
{
    std::cout << "sum called!!!\n";
    return a + b;
}


int main()
{
    using namespace std;

    //packaged_task<int(int, int)> task{ sum };
    packaged_task task{ sum };
    //future<int> ftr = task.get_future();
    //future ftr = task.get_future();
    auto ftr = task.get_future();

    task(10, 20);

    cout << ftr.get() << '\n';
}
```

→ Ayrı threadde çalışma örneği:



```cpp
int sum(int a, int b)
{
    std::cout << "sum called!!!\n";
    return a + b;
}


int main()
{
    using namespace std;

    //packaged_task<int(int, int)> task{ sum };
    packaged_task task{ sum };
    //future<int> ftr = task.get_future();
    //future ftr = task.get_future();
    auto ftr = task.get_future();

    thread t{ std::move(task), 10, 29 };

    cout << "result = " << ftr.get() << "\n";

    t.join();
```

⊛ Eğer bir workload vermeden operator() çağırılırsa → exception gönderilir.



```cpp
try {
    mytask(20, 40);
}
//catch (const std::exception& ex) {
catch (const std::future_error& ex) {
    std::cout << "exception caught: " << ex.what() << '\n';
}
```

future error türünden

* Tüm callable alternatifler argüman olarak gönderilebilir.

* Initialization Assignment Split: int x;
x = expr yapma!

- Çünkü const olma imkanı gitti
- ya ternary, ya lambda kullan!

* Packaged task, containerda tutulabilir.

* Conditioned Variable: - Bir thread iş yapıyor. Diğer thread ona bağlı bir iş yapıyor.
- Bir thread işi bitirdiğinde bir sinyal çıkar, o sinyal'e göre diğer thread işe başlar!

↓

bu sinyali alan
ve diğer thread'i
harekete geçiren
(yönetici)

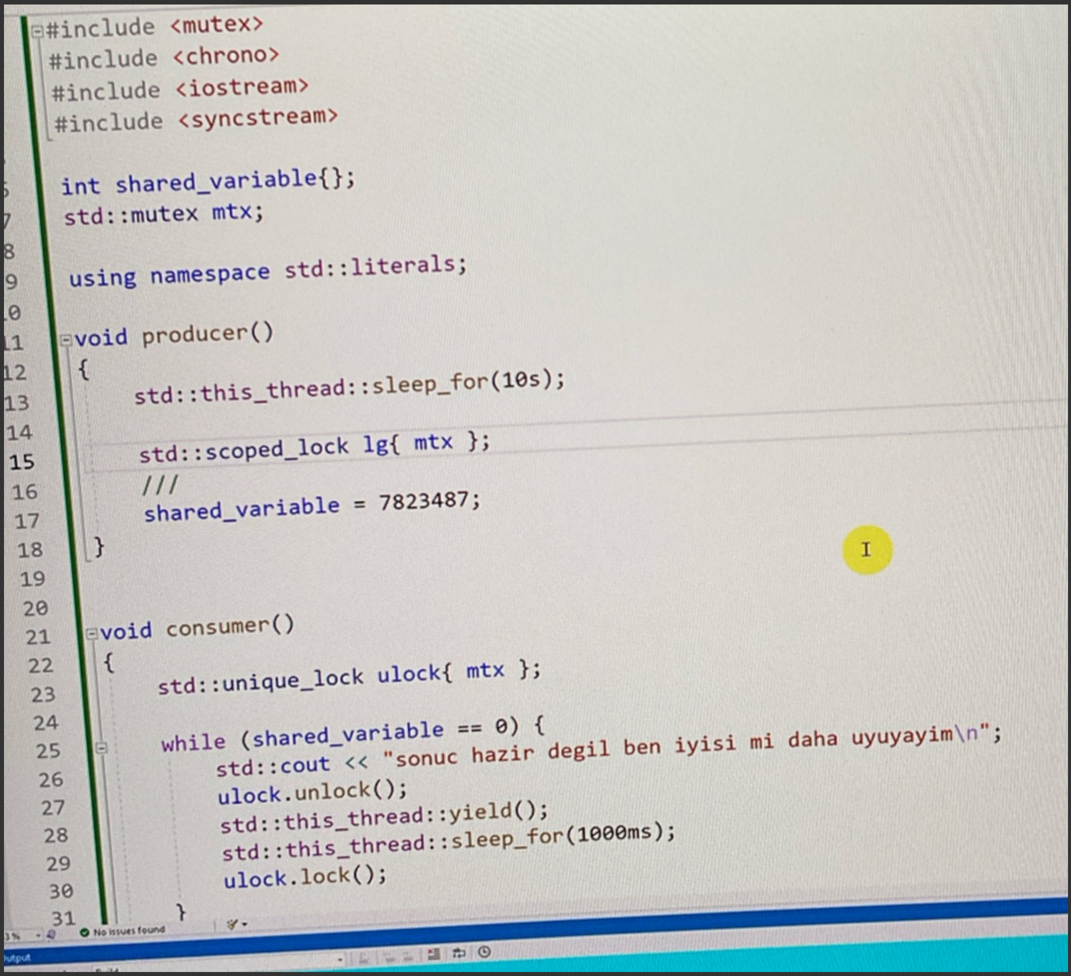
= conditioned variable ✓

- Boş yere CPU zamanı almıyor oluruz.

→ Biz ikinci thread'i ya tam vaktinde uyandırırız,
ya da vaktinden önce uyanır → spurious wake up

→ Bu yüzden conditioned variable kullanılan durumlarda, wake up türü kontrol edilmeli!

→ Sürekli "veri hazır mı" diye poll etmenin önüne geçer!

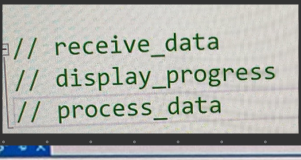


```
#include <mutex>
#include <chrono>
#include <iostream>
#include <syncstream>

int shared_variable{};
std::mutex mtx;

using namespace std::literals;

void producer()
{
    std::this_thread::sleep_for(10s);

    std::scoped_lock lg{ mtx };
    ///
    shared_variable = 7823487;
}


void consumer()
{
    std::unique_lock ulock{ mtx };

    while (shared_variable == 0) {
        std::cout << "sonuc hazir degil ben iyisi mi daha uyuyayim\n";
        ulock.unlock();
        std::this_thread::yield();
        std::this_thread::sleep_for(1000ms);
        ulock.lock();
    }
```

⇒ Polling modeli

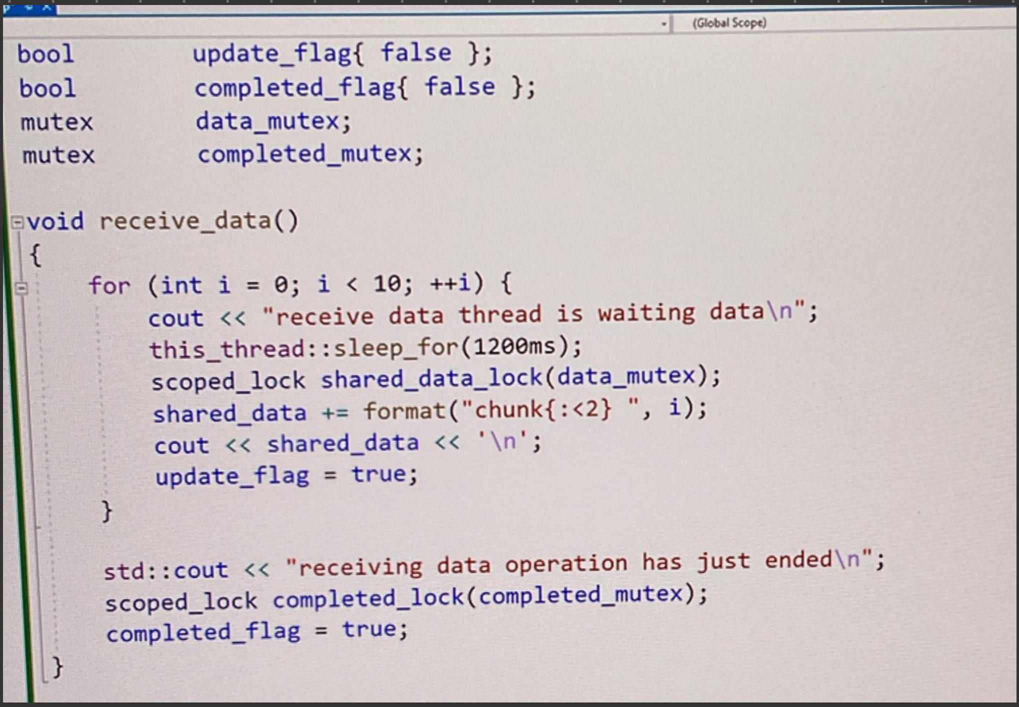
*Örnek: → Polling ile

-Hedef:

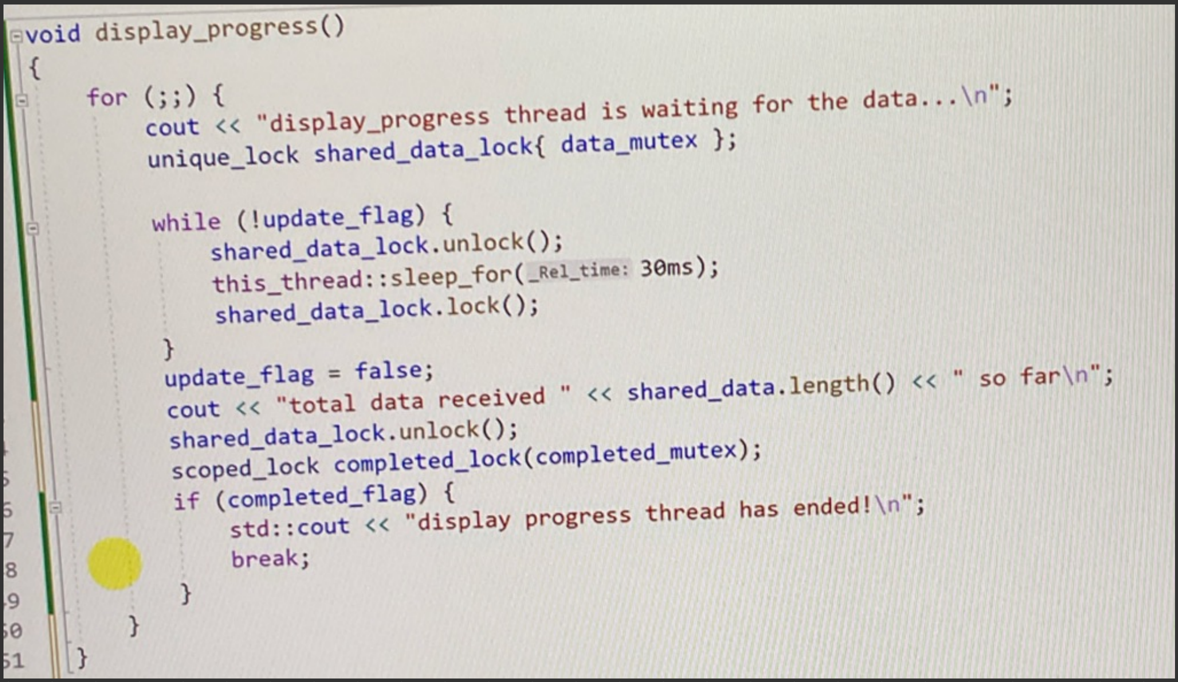


```cpp
// receive_data
// display_progress
// process_data
```

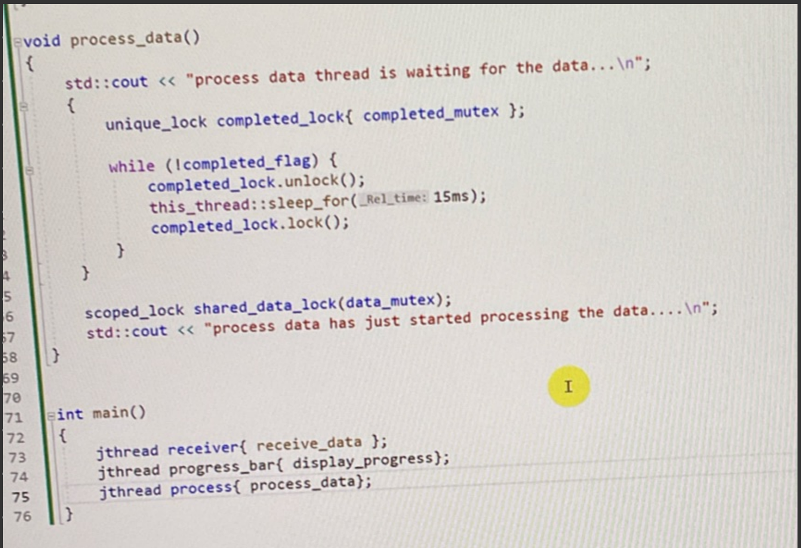
=> 3 farklı thread



```cpp
bool        update_flag{ false };
bool        completed_flag{ false };
mutex       data_mutex;
mutex       completed_mutex;

void receive_data()
{
    for (int i = 0; i < 10; ++i) {
        cout << "receive data thread is waiting data\n";
        this_thread::sleep_for(1200ms);
        scoped_lock shared_data_lock(data_mutex);
        shared_data += format("chunk{:<2} ", i);
        cout << shared_data << '\n';
        update_flag = true;
    }

    std::cout << "receiving data operation has just ended\n";
    scoped_lock completed_lock(completed_mutex);
    completed_flag = true;
}
```



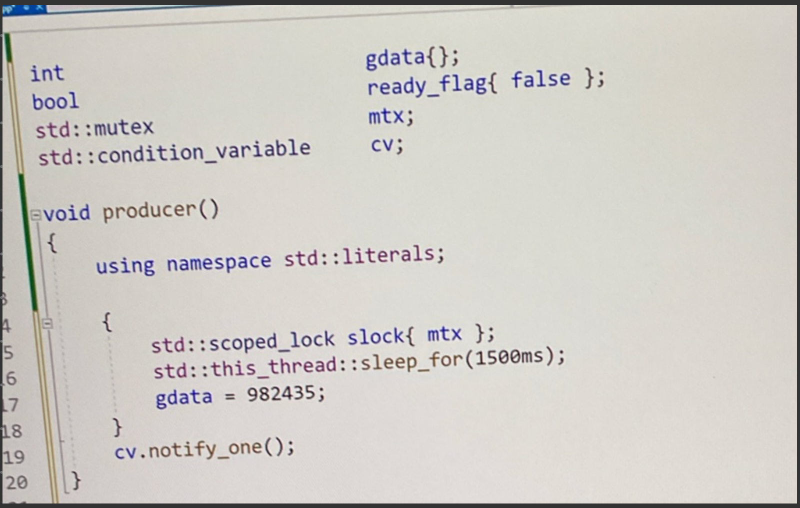
```cpp
void display_progress()
{
    for (;;) {
        cout << "display_progress thread is waiting for the data...\n";
        unique_lock shared_data_lock{ data_mutex };

        while (!update_flag) {
            shared_data_lock.unlock();
            this_thread::sleep_for(_Rel_time: 30ms);
            shared_data_lock.lock();
        }
        update_flag = false;
        cout << "total data received " << shared_data.length() << " so far\n";
        shared_data_lock.unlock();
        scoped_lock completed_lock(completed_mutex);
        if (completed_flag) {
            std::cout << "display progress thread has ended!\n";
            break;
        }
    }
}
```



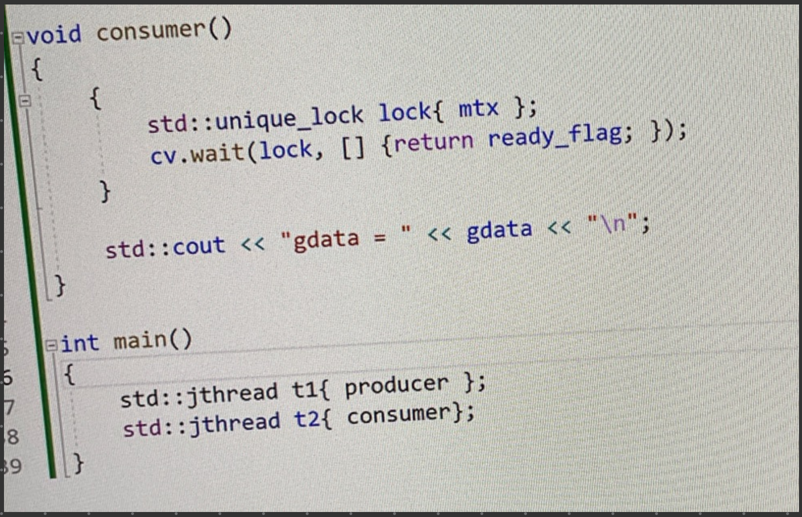
```cpp
void process_data()
{
    std::cout << "process data thread is waiting for the data...\n";
    {
        unique_lock completed_lock{ completed_mutex };

        while (!completed_flag) {
            completed_lock.unlock();
            this_thread::sleep_for(_Rel_time: 15ms);
            completed_lock.lock();
        }
    }

    scoped_lock shared_data_lock(data_mutex);
    std::cout << "process data has just started processing the data....\n";
}


int main()
{
    jthread receiver{ receive_data };
    jthread progress_bar{ display_progress};
    jthread process{ process_data};
}
```

*Örnek: → Condition variable:

→ producer consumer.

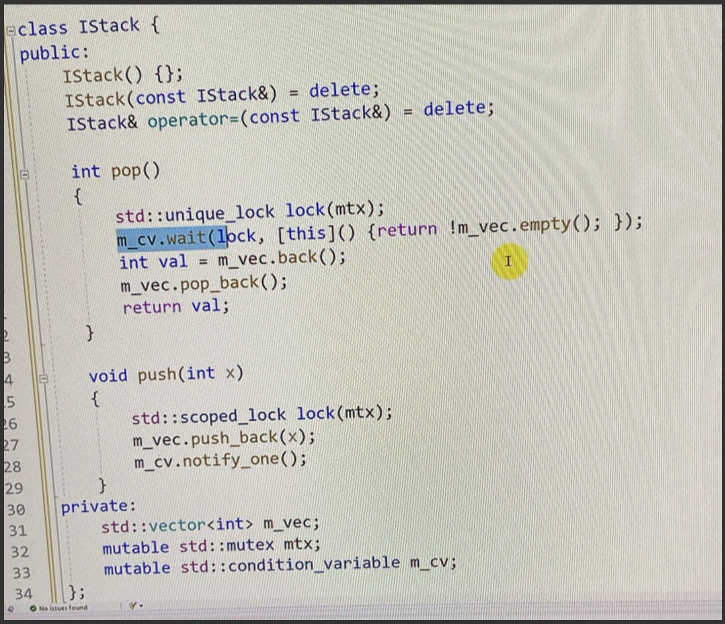


```cpp
int                            gdata{};
bool                           ready_flag{ false };
std::mutex                     mtx;
std::condition_variable        cv;

void producer()
{
    using namespace std::literals;

    {
        std::scoped_lock slock{ mtx };
        std::this_thread::sleep_for(1500ms);
        gdata = 982435;
    }
    cv.notify_one();
}
```

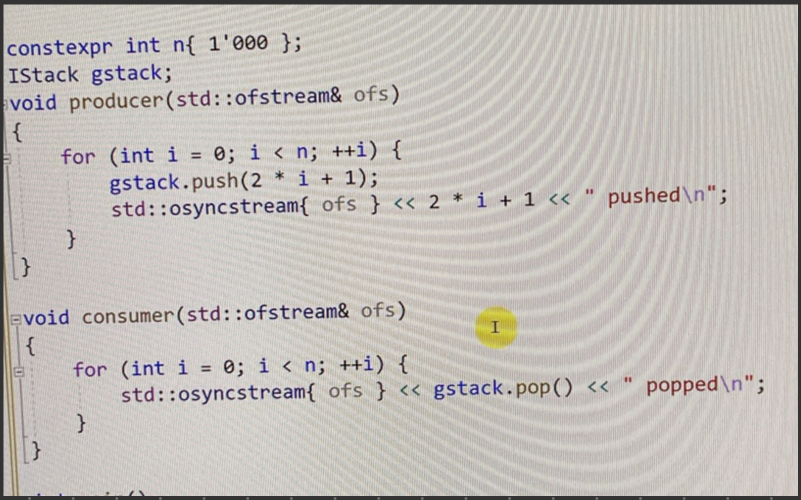


```cpp
void consumer()
{
    {
        std::unique_lock lock{ mtx };
        cv.wait(lock, [] {return ready_flag; });
    }

    std::cout << "gdata = " << gdata << "\n";
}

int main()
{
    std::jthread t1{ producer };
    std::jthread t2{ consumer};
}
```

* Stack örneği: Single Producer / Single Consumer



```cpp
class IStack {
public:
    IStack() {};
    IStack(const IStack&) = delete;
    IStack& operator=(const IStack&) = delete;

    int pop()
    {
        std::unique_lock lock(mtx);
        m_cv.wait(lock, [this]() {return !m_vec.empty(); });
        int val = m_vec.back();
        m_vec.pop_back();
        return val;
    }

    void push(int x)
    {
        std::scoped_lock lock(mtx);
        m_vec.push_back(x);
        m_cv.notify_one();
    }
private:
    std::vector<int> m_vec;
    mutable std::mutex mtx;
    mutable std::condition_variable m_cv;
};
```



```cpp
constexpr int n{ 1'000 };
IStack gstack;
void producer(std::ofstream& ofs)
{
    for (int i = 0; i < n; ++i) {
        gstack.push(2 * i + 1);
        std::osyncstream{ ofs } << 2 * i + 1 << " pushed\n";
    }
}

void consumer(std::ofstream& ofs)
{
    for (int i = 0; i < n; ++i) {
        std::osyncstream{ ofs } << gstack.pop() << " popped\n";
    }
}
```

Devamı diğer deste!