ერთდროული მოქმედებების სინქრონიზება

განხილული საკითხები:

- ხდომილების ან პირობის მოლოდინი
- დინებებისგან დაცული რიგის აგება პირობიანი ცვლადის საფუძველზე >>>
- ერთჯერადი ხდომილებებისთვის დალოდება >>>
- მნიშვნელობების დაბრუნება ზურგის ამოცანებიდან >>>
- ამოცანის დაკავშირება დამდეგთან >>>
- დაპირებების (std::promise) გაკეთება >>>
- გამონაკლისის შენახვა დამდეგისთვის >>>
- shared_future >>>
- ლოდინი დროსთან დაკავშირებულ შეზღუდვებში >>>
- საათები >>>
- ხანგრძლივობები (durations) >>>
- დროის მომენტები >>>
- ფუნქციები, რომლებიც აღიარებენ შესვენებებს >>>

სავარჯიშოები >>>

ერთდროულ პროცესებში ერთ-ერთი მთავარი საკითხი არის სხვადასხვა დინებების მიერ განხორციელებული მოქმედებების სინქრონიზება. შესაძლოა ერთ დინებას მოუწიოს რომ დაელოდოს მეორე დინების მიერ დაწყებული ამოცანის დასრულებას ვიდრე თვითონ დაამთავრებს საკუთარს. ზოგადად, გავრცელებული შემთხვევებია როდესაც დინება იცდის ვიდრე განსაზღვრული ხდომილება მოხდება ან რაიმე პირობა გახდება ჭეშმარიტი. C++-ის სტანდარტული ბიბლიოთეკა გვთავაზობს (პირობის ცვლადები) condition variables და futures-ებს ასეთი სცენარებისთვის. უფრო მეტი, ერთდროული პროცესების ტექნიკური დახასიათება (Technical Specification, TS) futures-ებისთვის დამატებით გვთავაზობს latches (ურდულების) და barriers (ბარიერეების) სახით.

ხდომილების ან პირობის მოლოდინი

სხვა დინების მიერ გამოსროლილი ხდომილებისთვის დალოდების ძირითადი მექანიზმი არის პირობიანი ცვლადი - condition variable. იდეურად, პირობიანი ცვლადი დაკავშირებულია ხდომილებასთან ან პირობასთან, ხოლო ორი ან მეტი დინება შესაძლოა იცდიდეს ვიდრე ეს პირობა შესრულდება. როდესაც დინება დაადგენს რომ პირობა სრულდება, იგი აცნობებს (notify) ერთ ან მეტ დინებას, რომლებიც იცდიან, რათა მათ გაიღვიძონ და გააგრძელონ მუშაობა.

C++ -ის სტანდარტული ბიბლიოთეკა გვთავაზობს პირობიანი ცვლადის ორ იმპლემენტაციას: `std::condition_variable` და `std::condition_variable_any`. ორივე აღწერილია `<condition_variable>`-ში. ორივე შემთხვევაში აუცილებელი მუტექსი საჭირო სინქრონიზების მისაღწევად. პირველი მათგანი მხოლოდ მუტექსთან მუშაობს, ხოლო მეორეს შეუძლია მუტექსსინაირი მინიმალური კრიტერიუმების მქონე მექანიზმთან მუშაობა, ანუ უფრო ზოგადია და შესაბამისად უფრო ნელიც. ამიტომ როდესაც არის არჩევანი, აუნდა ვისარგებლოთ `std::condition_variable`-ით.

პირველი მაგალითი გავარჩიოთ https://en.cppreference.com/w/cpp/thread/condition_variable-იდან.

`condition_variable` კლასი არის სინქრონიზაციის პრიმიტივი, რომელიც გამოიყენება ერთი ან მრავალი დინების დაბლოკვისთვის (შეჩერებისთვის), ვიდრე სხვა დინება გააკეთებს ორ საქმეს: შეცვლის საზიარო ცვლადს (პირობას) და აცნობებს `condition_variable` -ს.

თუ დინება აპირებს ცვლადის შეცვლას, მას მოუწევს:

1. დაისაკუთროს `std::mutex` (ჩვეულებრივ std::lock_guard-ით)
2. განახორციელოს ცვლილება ვიდრე ჩაკეტვა მოქმედებს

3. `std::condition_variable` -ზე შეასრულოს notify_one ან notify_all (ამისთვის აღარაა საჭირო ჩაკეტვის დაჭერა)

იმ შემთხვევაშიც კი, თუ ცვლადი ატომურია, იგი უნდა შეიცვალოს მუტექსის ქვეშ, რათა ცვლილება სწორად ეცნობოს მომლოდინე დინებას.

ნებისმიერ დინებას, რომელიც აპირებს რომ დაელოდოს `std::condition_variable` -ს, მოუწევს:

1. დაადოს `std::unique_lock<std::mutex>`, იგივე მუტექსზე რაც გამოიყენება საზიარო ცვლადის დასაცავად,
2. შეასრულოს wait, wait_for, ან wait_until. მოცდის ოპერაციები თავისით ათავისუფლებენ მუტექსს და აჩერებენ დინების მოქმედებას,
3. როდესაც პირობიანი ცვლადი უკვე გაფრთხილებულია, შესვენება დასრულდა, ან ცრუ გაღვიძება (spurious wakeup) მოხდა, დინება იღვიძებს, ხოლო მუტექსი ატომურად აღდგება. შემდეგ დინებამ უნდა გადაამოწმოს პირობა და განაახლოს მოლოდინი, თუ გაღვიძება ცრუ აღმოჩნდა.

უნდა გავითვალისწინოთ, რომ `std::condition_variable` მუშაობს მხოლოდ `std::unique_lock<std::mutex>` -თან. ეს შეზღუდვა განაპირობებს მაქსიმალურ წარმადობას გარკვეულ პლატფორმებზე.

შემდეგ მაგალითში, `condition_variable` გამოიყენება წყვილში `std::mutex` -თან დინებებს შორის ურთიერთობის აწყობისთვის.

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
	// Wait until main() sends data
	std::unique_lock<std::mutex> lk(m);
	cv.wait(lk, [] {return ready; });

	// after the wait, we own the lock.
	std::cout << "Worker thread is processing data\n";
	data += " after processing";

	// Send data back to main()
	processed = true;
	std::cout << "Worker thread signals data processing completed\n";

	// Manual unlocking is done before notifying, to avoid waking up
	// the waiting thread only to block again (see notify_one for details)
	lk.unlock();
	cv.notify_one();
}

int main()
{
	std::thread worker(worker_thread);
```

```
        data = "Example data";
        // send data to the worker thread
        {
                std::lock_guard<std::mutex> lk(m);
                ready = true;
                std::cout << "main() signals data ready for processing\n";
        }
        cv.notify_one();

        // wait for the worker
        {
                std::unique_lock<std::mutex> lk(m);
                cv.wait(lk, [] {return processed; });
        }
        std::cout << "Back in main(), data = " << data << '\n';

        worker.join();
}
```

შედეგი:

```
main() signals data ready for processing
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing
```

ახლა განვიხილოთ მაგალითი, რომელშიც ლოდინის პირობები მიმსგავსებულია რეალურთან. განვიხილოთ ლისტიგი:

Listing 3.1 Waiting for data to process with std::condition_variable

```
std::mutex mut;
std::queue<data_chunk> data_queue;    ❶
std::condition_variable data_cond;

void data_preparation_thread()
{
        while (more_data_to_prepare())
        {
                data_chunk const data = prepare_data();
                {
                        std::lock_guard<std::mutex> lk(mut);
                        data_queue.push(data);      ❷
                }
                data_cond.notify_one();    ❸
        }
}
void data_processing_thread()
{
        while (true)
        {
                std::unique_lock<std::mutex> lk(mut);   ❹
                data_cond.wait(
                        lk, [] {return !data_queue.empty(); });  ❺
                data_chunk data = data_queue.front();
                data_queue.pop();
                lk.unlock();        ❻
                process(data);
                if (is_last_chunk(data))
```

```
                    break;
            }
    }
```

ორ დინებას შორის მონაცემების გაცვლისთვის გამოიყენება რიგი ❶. როდესაც მონაცემი მზადაა, ის დინება რომელიც ამზადებს მონაცემებს კეტავს მუტექსს, იცავს რიგს `std::lock_guard` -ით და ჩასვამს მონაცემებს რიგში ❷. შემდეგ ეს დინება `data_cond` ნიმუშით გამოიძახებს კლას `std::condition_variable`-ის წევრ ფუნქციას (მეთოდს) `notify_one()`-ს, რათა გააფრთხილოს მომლოდინე დინება ❸. შევნიშნოთ, რომ მონაცემების რიგში ჩასასმელი კოდი პატარა დაცულ მარყუჟში არის მოთავსებული, ამიტომ მომლოდინე დინების გაფრთხილება ხდება მუტექსის გახსნის შემდეგ - ეს ასე იმიტომაა, რომ თუ მომლოდინე დინება მყისიერად გაიღვიძებს, იგი არ აღმოჩნდება დაბლოკილი (შეჩერებული), ანუ კვლავ იმის მოლოდინში თუ როდის გახსნით მუტექსს.

მეორე მხრივ, გვაქვს დამმუშავებელი დინება. იგი ჯერ ჩაკეტავს მუტექსს, მაგრამ ამჯერად `std::unique_loc`-ით ❹, რის მიზეზსაც სულ მალე ავხსნით. დინება ამის შემდეგ იძახებს `wait()` მეთოდს `std::condition_variable` პირობიანი ცვლადით, ხოლო მეთოდს არგუმენტებად გადააწვდის ჩამკეტავ ობიექტს და ლამბდას, რომელიც წარმოადგენს იმ პირობას რომლისთვისაც იცდიან ❺. ჩვენს შემთხვევაში, ლამბდა ფუნქცია `[]{return !data_queue.empty(); }` ამოწმებს არის თუ არა მონაცემების რიგი ცარიელი.

შემდეგ `wait()`-ის განხორციელება ამოწმებს პირობას (გადაწოდებული ლამბდა ფუნქციის საშუალებით) და ბრუნდება (returns) თუ პირობა შესრულდა (რიგი არაა ცარიელი და ლამბდა აბრუნებს `true`-ს). თუ პირობა არ სრულდება (ანუ ლამბდა აბრუნებს `false`-ს), `wait()` გახსნის მუტექსს და დინებას ჩააყენებს დაბლოკილ ანუ მომლოდინე დგომარეობაში. როდესაც პირობიან ცვლადს ეცნობება `notify_one()`-ით მონაცემების მომზადებელი დინებისგან, მაშინ დინება გაიღვიძებს, ხელახლა მოითხოვს ჩაკეტვას მუტექსზე მაინც გადაამოწმებს პირობას, რაც დააბრუნა `wait()`-მა ჯერ კიდევ ჩაკეტილი მუტექსით როდესაც პირობა შესრულდა. თუ პირობა არის იყო შესრულებული, დინება გახსნის მუტექსს და გააგრძელებს ლოდინს.

(კ.გ.: თუმცა, ეს ახსნა ცოტა საეჭვოა - ეს ასეა თუ დინებამ ტყუილად გაიღვიძა როდესაც სრულდელებოდა `data_chunk const data = prepare_data();` სტრიქონი, რასაც ალბათ მეტი დრო უნდა ვიდრე მის მომდევნო მარყუჟს და ალბათობაც დიდია. მაგრამ თუ ტყუილად გაიღვიძა უშუალოდ მარყუჟის დროს, მაშინ ვერ ჩაკეტავს მუტექსს და ამ მიზეზითაც კვლავ დაიძინებს. ნებისმიერ შემთხვევაში, პირობას გადამოწმება უნდა).

ესაა რის გამოც გამოიყენება `std::unique_loc`. მომლოდინე დინებამ უნდა გახსნას მუტექსი ვიდრე იცდის და შემდეგ კვლავ ჩაკეტოს იგი. ასეთი მოქნილობა არ გააჩნია `std::lock_guard` -ს. `std::unique_loc` -ის მოქნილობა საჭირო ხდება როდესაც მივიღეთ დასამუშავებელი მონაცემები მაგრამ ჯერ არ დაგვიწყია დამუშავება ❻. მონაცემების დამუშავება ხშირად ხარჯიანია დროში, ხოლო საჭიროზე მეტი დროით მუტექსის დაკეტვა ცუდი აზრია.

რიგის გამოყენება დინებებს შორის მონაცემების გაცვლისთვის გავრცელებული სცენარია. თუ კარგად გავაკეთებთ, სინქრონიზება შეიძლება შემოიფარგლოს თვითონ რიგით. ამიტომ საჭიროა განვავითაროთ დინებებისგან დაცული რიგის თემა.

<<< დინებებისგან დაცული რიგის აგება პირობიანი ცვლადის საფუძველზე

როგორც სტეკის შემთხვევაში ვნახეთ, საჭიროა კარგად შევისწავლოთ რიგის ინტერფეისი, რომ ჩიხისკენ მიდრეკილებები გავრიცხოთ. სტანდარტული ბიბლიოთეკიდან:

ლისტინგი 3.2 std::queue -ს ინტერფეისი

```
template <class T, class Container = std::deque<T> >
class queue {
public:
       explicit queue(const Container&);
```

```
        explicit queue(Container&& = Container());
        template <class Alloc> explicit queue(const Alloc&);
        template <class Alloc> queue(const Container&, const Alloc&);
        template <class Alloc> queue(Container&&, const Alloc&);
        template <class Alloc> queue(queue&&, const Alloc&);
        void swap(queue& q);
        bool empty() const;
        size_type size() const;
        T& front();
        const T& front() const;
        T& back();
        const T& back() const;
        void push(const T& x);
        void push(T&& x);
        void pop();
        template <class... Args> void emplace(Args&&... args);
};
```

აგების, მინიჭების და გადაცვლის ოპერაციების გარდა, გვრჩება სამი ჯგუფის ოპერაციები: რომლებიც ინტერესდებიან რიგის მონაცემებით მთლიანობაში - (`empty()`, `size()`), რომელიც მოითხოვენ რიგის ელემენტებს - (`front()`, `back()`), და რომლებიც ცვლიან რიგს - (`push()`, `pop()`, `emplace()`). სტეკისთვისაც იგივე იყო, ამიტომ ინტერფეისში გვაქვს მონაცემების რბოლასთან დაკავშირებული იგივე საკითხები. შესაბამისად, აქაც უნდა შევაერთოთ `top()` და `pop()`. ლისტინგ 3.1-ის კოდი გვკარნახობს ერთ აზრს: როდესაც იცვლება მონაცემები დინებებს შორის, მიმღებ (დამმუშავებელ) დინებას ხშირად უწევს მონაცემებისთვის ლოდინი. ამიტომ ვაკეთებთ `pop()`-ის ორ ვარიანტს: `try_pop()`, რომელიც ცდილობს მნიშვნელობის ამოღებას რიგიდან, მაგრამ ყოველთვის დაუყოვნებლივ ბრუნდება (ჩავარდნის ან წარმატების ჩვენებით დასაბრუნებელ მნიშვნელობაში), მაშინაც კი როდესაც ამოსაღები არაფერი არაა; `wait_and_pop()`, რომელიც დაელოდება ვიდრე გამოჩნდება ამოსაღები მნიშვნელობა. ინტერფეისს ექნება სახე:

ლისტინგი 3.3 დინებებისგან დაცული `threadsafe_queue`-ს ინტერფეისი

```
#include <memory>
template<typename T>
class threadsafe_queue
{
public:
        threadsafe_queue();
        threadsafe_queue(const threadsafe_queue&);
        threadsafe_queue& operator=(const threadsafe_queue&) = delete;
        void push(T new_value);
        bool try_pop(T& value);          ❶
        std::shared_ptr<T> try_pop();    ❷
        void wait_and_pop(T& value);
        std::shared_ptr<T> wait_and_pop();
        bool empty() const;
};
```

ისევე როგორც სტეკის შემთხვევაში, შევამცირეთ კონსტრუქტორები, გამოვრიცხეთ მინიჭება. ორ-ორი ვერსია გვაქვს `try_pop()`-ისა და `wait_for_pop()`-ისთვის. `try_pop()`-ის პირველი გადატვირთვა ❶ ინახავს ამოღებულ მნიშვნელობას გადაწოდებულ ცვლადში, ხოლო დასაბრუნებელ მნიშვნელობას იყენებს სტატუსისთვის: აბრუნებს ჭეშმარიტს თუ ამოიღო მნიშვნელობა და მცდარს წინააღმდეგ შემთხვევაში. მეორე გადატვირთვას ❷ არ შეუძლია ამის გაკეთება, რადგან უშუალოდ აბრუნებს ამოღებულ მნიშვნელობას. მაგრამ დაბრუნებული პოინტერი შესაძლოა გავუტოლოთ `nullptr`-ს როდესაც არაფერია ამოსაღები.

ახლა ვნახოთ თუ როგორ შეიძლება რიგის გამოყენება დინებებს შორის მონაცემების გაცვლისთვის.

ლისტინგი 3.4 წინა ლისტინგის push( )-ის და wait_and_pop( )-ის ჩაშლა და გამოყენება

```
#include <queue>
#include <mutex>
#include <condition_variable>
template<typename T>
class threadsafe_queue
{
private:
	std::mutex mut;
	std::queue<T> data_queue;
	std::condition_variable data_cond;
public:
	void push(T new_value)
	{
		std::lock_guard<std::mutex> lk(mut);
		data_queue.push(new_value);
		data_cond.notify_one();
	}
	void wait_and_pop(T& value)
	{
		std::unique_lock<std::mutex> lk(mut);
		data_cond.wait(lk, [this] {return !data_queue.empty(); });
		value = data_queue.front();
		data_queue.pop();
	}
};
threadsafe_queue<data_chunk> data_queue;        ❶
void data_preparation_thread()
{
while (more_data_to_prepare())
{
	data_chunk const data = prepare_data();
	data_queue.push(data);          ❷
}
}
void data_processing_thread()
{
	while (true)
	{
		data_chunk data;
		data_queue.wait_and_pop(data);      ❸
		process(data);
		if (is_last_chunk(data))
			break;
	}
}
```

მუტექსი და პირობიანი ცვლადი ამჯერად წარმოადგენს `threadsafe_queue` -ს ნაწილს, ასე რომ დამატებითი ცვლადები საჭირო აღარაა ❶, შესაბამისად არავითარი გარე სინქრონიზაცია არაა საჭირო `push()`-ის გამოძახებისთვის. მეორე მხრივ, `wait_and_pop( )` ზრუნავს პირობიანი ცვლადის მოცდაზე ❸.

ახლა, დინებებისგან დაცული რიგის სრულ იმპლემენტაციას აქვს სახე:

ლისტინგი 3.5 დინებებისგან დაცული `threadsafe_queue`-ს იმპლემენტაცია

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>
```

```
template<typename T>
class threadsafe_queue
{
private:
	mutable std::mutex mut;		❶ - მუტექსის ტიპის მახასიათებელი
	std::queue<T> data_queue;
	std::condition_variable data_cond;
public:
	threadsafe_queue()
	{}
	threadsafe_queue(threadsafe_queue const& other)
	{
		std::lock_guard<std::mutex> lk(other.mut);
		data_queue = other.data_queue;
	}
	void push(T new_value)
	{
		std::lock_guard<std::mutex> lk(mut);
		data_queue.push(new_value);
	data_cond.notify_one();
	}
	void wait_and_pop(T& value)
	{
		std::unique_lock<std::mutex> lk(mut);
		data_cond.wait(lk, [this] {return !data_queue.empty(); });
		value = data_queue.front();
		data_queue.pop();
	}
	std::shared_ptr<T> wait_and_pop()
	{
		std::unique_lock<std::mutex> lk(mut);
		data_cond.wait(lk, [this] {return !data_queue.empty(); });
		std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
		data_queue.pop();
		return res;
	}
	bool try_pop(T& value)
	{
		std::lock_guard<std::mutex> lk(mut);
		if (data_queue.empty())
			return false;
		value = data_queue.front();
		data_queue.pop();
		return true;
	}
	std::shared_ptr<T> try_pop()
	{
		std::lock_guard<std::mutex> lk(mut);
		if (data_queue.empty())
			return std::shared_ptr<T>();
		std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
		data_queue.pop();
		return res;
	}
	bool empty() const
	{
		std::lock_guard<std::mutex> lk(mut);
		return data_queue.empty();
	}
};
```

თუმცა `empty()` არის მუდმივი მეთოდი და კონტრუქტორის `other` პარამეტრიც არის მუდმივი, მუტექსის ჩაკეტვა მაინც საჭიროა რადგან სხვა დინებებს შესაძლოა ჰქონდეთ არა-მუდმივი რეფერენსები ობიექტებზე და იძახებნენ სახეცვლად ოპერაციებს. მუტექსის ჩაკეტვა არის სახეცვლადი ოპერაცია, ამიტომ მუტექსი უნდა გამოცხადდდეს როგორც `mutable` ❶ დაიკეტოს `empty()` -სა და ასლის კონსტრუქტორში.

პირობიანი ცვლადები მაშინაც გამოიყენება, როდესაც ერთზე მეტი დინება ელოდება ერთი და იმავე ხდომილებას. თუ დინებები გამოიყენება საერთო საქმის დასანაწილებლად, და ამიტომ მხოლოდ ერთმა დინებამ უნდა უპასუხოს ხდომილების შესრულების ცნობას (გაფრთხილებას), მაშინ ზუსტად იგივე სტრუქტურა გამოიყენება რაც ლისტინგ 3.1-ში. როდესაც მონაცემები მზადაა, `notify_one()`-ის გამოძახება აამუშავებს ერთ-ერთ მომლოდინე დინებას რომ შეამოწმოს პირობა და დაბრუნდეს `wait()`-ისგან. არავითარი გარანტია არაა თუ თუ რომელი დინება იქნება გაფრთხილებული, ან თუ არსებობს დინება რომელიც იცდის; ყველა დინება შესაძლოა მონაცემებს ამუშავებდეს.

სხვა სცენარში, რამდენიმე დინება ელოდება ხდომილებას, მაგრამ საჭიროა რომ ყველამ უპასუხოს. ეს ხდება საზიარო მონაცემების ინიციალიზაციისას, ყველა დინებას სჭირდება მონაცემები, მაგრამ უნდა დაელოდონ მას (ამ მაგალითისთვის უკეთესი ვარიანტი არის `std::call_once`). ამ დროს იყენებენ `notify_all()`-ს.

თუ მომლოდინე დინებისგან მოცდა მხოლოდ ერთხელ არის საჭირო, პირობიანი ცვლადის გამოყენება არაა საუკეთესო ვარიანტი. ეს განსაკუთრებით მაშინაა მნიშვნელოვნაი, როდესაც დაცდის პირობა არის მონაცემების კერძო სახის მისაწვდომობა. ასეთ სცენარებში, *future* უფრო შესაფერისი ჩანს.

<<< ერთჯერადი ხდომილებებისთვის ლოდინი

Future თავსართი საშუალებას იძლევა, რომ მივიღოთ ასინქრონული წვდომა მნიშვნელობებზე, რომლებიც დაყენებულია სხვადასხვა მომწოდებლების მიერ, შესაძლოა სხვადასხვა დინებებში.

თითოეული ეს მომწოდებელი (ან `async`, ან `packaged task`-ის ობიექტი, ან `promise`) აზიარებს წვდომას საზიარო მონაცემზე `future` ობიექტის საშუალებით.

C++ -ის სტანდარტულ ბიბლიოთეკაში არის ორი სახის Future, განხორციელებული როგორც კლასის თარგი <Future> სათაურიან ბიბლიოთეკაში. ჰქვიათ *ეული დამდეგი* (*unique future*) და *საზიარო დამდეგი* (*shared future*). ისინი დამოდელებულია ეული პოინტერის და საზიარო პოინტერის საშუალებით, შესაბამისად, `std::future` -ის ობიექტი არის ერთადერთი რომელიც მიუთითებს მასთან დაკავშირებულ ხდომილებას, ხოლო `std::shared_future` -ის ობიექტები შესაძლოა მიუთითებდნენ ერთი და იმავე ხდომილებას. ამ ბოლო შემთხვევაში, ყველა ნიმუში ერდროულად იქნება მზად და მათ ყველას შეეძლება წვდომა ხდომილებასთან დაკავშირებულ მონაცემებზე. ამ დაკავშირებული მონაცემების ტიპი არსი თარგის პარამეტრი.

ერთდროულობის TS-ში არის ამ კლასების გავრცობილი სახეები: `std::experimental` სახელთა სივრცეში. ჰქვიათ `std::experimental::future<>` და `std::experimental::shared_future<>` და აქვთ დამატებითი მეთოდები.

<<< მნიშვნელობების დაბრუნება ზურგის ამოცანებიდან

ვთქვათ გვჭირდება ხანგრძლივი გამოთვლები რაც საბოლოოდ მოგვცემს შედეგს. ვუშვებთ ახალ დინებას გამოთვლების შესასრულებლად. `std::thread` არ იძლევა პირდაპირ საშუალებას შედეგის დასაბრუნებლად. აი აქ არის სასარგებლო `std::async` ფუნქცია (იგივე სათაურის მქონე ბიბლიოთეკაშია მოთავსებული).

`std::async` გამოიყენება ასინქრონული ამოცანის დასაწყებად, რომლისთვისაც შედეგი არ გვჭირდება მაინცდამაინც გაშვების მომენტში. `std::async` აბრუნებს `std::future` ობიექტს, რომელიც საბოლოოდ მიიღებს ფუნქციის დასაბრუნებელ მნიშვნელობას. როდესაც

დაგვჭირდება ეს მნიშვნელობა, მაშინვე გამოვიძახებთ ამ დამდეგის (Future) `get()` მეთოდს და მიმდინარე დინება შეჩერდება (დაიბლოკება) ვიდრე დამდეგი არ იქნება მზად (ready) და დააბრუნებს მნიშვნელობას.

განვიხილოთ მარტივი მაგალითი:

```
// future example
#include <iostream>       // std::cout
#include <future>         // std::async, std::future
#include <thread>
// a non-optimized way of checking for prime numbers:
bool is_prime(int x) {
      for (int i = 2; i<x; ++i) if (x%i == 0) return false;
      return true;
}

int main()
{
      // call function asynchronously:
      std::future<bool> fut = std::async(is_prime, 444444443);
      bool x = fut.get();     // retrieve return value
      std::cout << "444444443 " << (x ? "is" : "is not") << " prime.\n";
}
```

get() უნდა გამოვიძახოთ მაშინ, როდესაც დასაბრუნებელი მნიშვნელობა აუცილებლად გვჭირდება. ნაადრევმა გამოძახებამ შესაძლოა გამოიწვიოს მთავარი დინების შეჩერება. განხილული მარტივი მაგალითი ამას ვერ გამოხატავს. ამიტომ განვიხილოთ შემდეგი კოდი, რომელშიც ძირითადი და ზურგის დინებები, ორივე 5 წამს გრძელდება. ერთი ხმოვან სიგნალს გამოსცემს, მეორე წერტილებს ბეჭდავს. ორივე პროცესი ერთდროულად მიდის:

```
// future example
#include <iostream>       // std::cout
#include <future>         // std::async, std::future
#include <chrono>
#include <thread>

void sing5S()
{
      for(int i=0; i<5;++i)
      {
            std::cout << '\a' << std::flush;
            std::this_thread::sleep_for(std::chrono::seconds(1));
      }
}
void dot5S()
{
      for (int i = 0; i<5; ++i)
      {
            std::cout << '.' << std::flush;
            std::this_thread::sleep_for(std::chrono::seconds(1));
      }
      std::cout << std::endl;
}
int main()
{
      // call function asynchronously:
      std::future<void> fut = std::async(sing5S);
      dot5S();
      fut.get();
}
```

საკმარისია გადავაადგილოთ -ის გამოძახება

```
int main()
{
	// call function asynchronously:
	std::future<void> fut = std::async(sing5S);
	fut.get();
	dot5S();
}
```

და ყველაფერი იცვლება: ველოდებით ზურგის დინებაში ფუნქციის დასრულებას და შემდეგ იწყება წერტილების ბეჭდვა.

`std::async` გვიშვებს რომ ფუნქციას გადავაწოდოთ დამატებითი არგუმენტები, თავის გამოძახებაში დამატებითი არგუმენტების დამატებით, ისევე როგორც ამას `std::thread` აკეთებს. თუ პირველი არგუმენტი არის წევრი-ფუნქციის (მეთოდის) პოინტერი, მაშინ მეორე არგუმენტი გვაწვდის იმ ობიექტის მისამართს, რომელიც იძახებს მეთოდს - ან უშუალოდ, ან პოინტერით, ან შეფუთული `std::ref`-ში. დანარჩენი არგუმენტები გადაეცემა როგორც მეთოდის არგუმენტები. სხვა შემთხვევაში, მეორე და დანარჩენი არგუმენტები გადაეცემა როგორც იმ ფუნქციის ან გამოძახებადი ობიექტის არგუმენტები, რომელიც განსაზღვრულია პირველი არგუმენტით. ისევე როგორც `std::thread` აკეთებს, თუ არგუმეტები მარჯვენა მნიშვნელობებია, მაშინ ასლები იქმნება ორიგინალების გადაადგილებით. ამის წყალობით მხოლოდ-გადაადგილებად ობიექტებს ვიყენებთ როგორც ფუნქციურ ობიექტებად, ასევე არგუმენტებად. განვიხილოთ შემდეგი ლისტინგი:

ლისტინგი 3.6 ფუნქციებისთვის არგუმენტების გადაწოდება `std::async`-ით

```
#include <string>
#include <future>
struct X
{
	void foo(int, std::string const&);
	std::string bar(std::string const&);
};
X x;
auto f1 = std::async(&X::foo, &x, 42, "hello");		//  (1)
auto f2 = std::async(&X::bar, x, "goodbye");		//  (2)
struct Y
{
	double operator()(double);
};
Y y;
auto f3 = std::async(Y(), 3.141);			//  (3)
auto f4 = std::async(std::ref(y), 2.718);		//  (4)
X baz(X&);
std::async(baz, std::ref(x));				//  (5)

class move_only
{
public:
    move_only();
    move_only(move_only&&)
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;
    void operator()();
};
auto f5 = std::async(move_only());			//  (6)
```

აქ:

(1) `p->foo(42,"hello")`-ს გამოძახება, სადაც p არის &x

(2) `tmpx.bar("goodbye")` -ს გამოძახება, სადაც `tmpx` არის x-ის ასლი
(3) `tmpy(3.141)` გამოძახება, სადაც `tmpy` აგებულია Y()-ის გადაადგილების კონსტრუქტორით
(4) `y(2.178)` გამოძახება
(5) `baz(x)` გამოძახება
(6) `tmp( )` გამოძახება, სადაც `tmp` აგებულია `std::move(move_only())`-ით

გულისხმობის პრინციპით, იმპლემენტაციაზეა დამოკიდებული `std::async` გაუშვებს ახალ დინებას, თუ ამოცანა გაეშვება სინქრონულად როდესაც დამდეგი მას უცდის. პროგრამისტს შეუძლია თავად განუსაზღვროს პროგრამას საჭირო ყოფაქცევა `std::async`-ის დამატებითი პარამეტრის საშუალებით ფუნქციის გამოძახებამდე. ეს პარამეტრი არის `std::launch` ტიპის და არის ან `std::launch::deferred` იმის საჩვენებლად რომ რომ ფუნქციის გამოძახება უნდა შეყოვნდეს ვიდრე ან `wait()` ან `get()` არ გაეშვება დამდეგიდან, ან არის `std::launch::async` იმის მისათითებლად რომ ფუნქცია უნდა გაეშვას საკუთარ დინებაში, ან არის

`std::launch::deferred | std::launch::async`

რაც ნიშნავს რომ იმპლემენტაციამ უნდა აირჩიოს. ეს ბოლო არის ნაგულისხმევი მნიშვნელობა. მაგალითად, ბოლო ლისტინგს თუ გავაგრძელებთ ასე:

```
auto f6 = std::async(std::launch::async, Y(), 1.2);                    //  (7)
auto f7 = std::async(std::launch::deferred, baz, std::ref(x));         //  (8)
auto f8 = std::async(                                                  //  (9)
      std::launch::deferred | std::launch::async,
      baz, std::ref(x));
auto f9 = std::async(baz, std::ref(x));                                //  (9)
f7.wait();                                                             //  (10)
```

აქ:
(7) ეშვება ახალ დინებაში
(8) დაყოვნებულია, სჭირდება `wait()` ან `get()`
(9) გადაწყვეტილება იმპლემენტაციაზეა
(10) ვაღვიძებთ დაყოვნებულ ფუნქციას

ამ თავის ბოლოს და მომდევნო თავებში ვნახავთ, რომ `std::async` -ის გამოყენება აადვილებს ალგორითმების დაყოფას ამოცანებად, რომლებიც გაეშვებიან ერთდროულად. ახლა განვიხილოთ ამოცანებთან დამდეგის დაკავშირების სხვა გზა.

<<< ამოცანის დაკავშირება დამდეგთან

`std::packaged_task<>` მიაბამს დამდეგს ფუნქციას ან გამოძახებად ობიექტს. როდესაც `std::packaged_task<>` -ის ობიექტი გაიღვიძებს, იგი გამოიძახებს დაკავშირებულ ფუნქციას ან გამოძახებად ობიექტს და დამდეგს მოიყვანს მზა (ready) მდგომარეობაში, ხოლო დასაბრუნებელი მნიშვნელობა შენახული იქნება დაკავშირებულ მონაცემში. ასეთი რამეები გამოიყენება როგორც სამშენებლო ბლოკები დინებების აუზის (tread pools) ასამუშავებლად.

`std::packaged_task<>` კლასის თარგის პარამეტრი არის ფუნქციის ხელწერა. მაგალითად, `void()` ნიშნავს, რომ პარამეტრად გამოდგება ფუნქცია რომელიც არაფერს აბრუნებს და არც არგუმენტები აქვს; `int(std::string&,double*)` ნიშნავს, რომ ელოდება ფუნქციებს რომლებიც აბრუნებენ მთელს, ხოლო პარამეტრებად მიიღებენ სტრინგს რეფერენსით და პოინტერს ორმაგ ნამდვილზე. როდესაც შექმნით `std::packaged_task<>` -ს, მას უნდა გადააწოდოთ ფუნქცია ან გამოძახებადი ობიექტი რომელიც აკმაყოფილებს მითითებულ პარამეტრებს და აბრუნებს მითითებულთან თავსებად ტიპს. ზუსტი დამთხვევა არააა აუცილებელი, - შეგიძლიათ, მაგალითად, ააგოთ `std::packaged_task<double(double)>` ფუნქციისგან რომელიც მიიღებს მთელს და აბრუნებს მცოცავწერტილიანს `(float)`.

მითითებული ხელწერის დასაბრუნებელი მნიშვნელობა განსაზღვრავს `std::future<>`-ის ტიპს, რაც დაბრუნდება `get_future()` მეთოდით, ხოლო ხელწერის არგუმენტების სია გამოიყენება შეფუთული ამოცანის ფუნქციის გამოძახების ოპერატორში. მაგალითად, `std::packaged_task<std::string(std::vector<char>*,int)>` ნაწილობრივი კლასის ინტერფეისი არის:

```
template<>
class packaged_task<std::string(std::vector<char>*, int)>
{
public:
	template<typename Callable>
	explicit packaged_task(Callable&& f);
	std::future<std::string> get_future();
	void operator()(std::vector<char>*, int);
};
```

`std::packaged_task<>` არის გამოძახებადი ობიექტი. შესაბამისად, შიძლება მისი შეფუთვა `std::function` ობიექტში, გადაწოდება `std::thread`-ისთვის როგორც შესასრულებელი ფუნქცია, გადაწოდება ნებისმიერი სხვა ფუნქციისთვის რომელიც თხოულობს გამოძახებად ობიექტს, ან პირდაპირ გაღვიძება. როდესაც შეფუთული ამოცანა გამოიძახება როგორც ფუნქციური ობიექტი, მისთვის გადაწოდებული არგუმენტები გადაეცემა ამოცანაში ჩასმულ ფუნქციას და დასაბრუნებელი მნიშვნელობა ინახება როგორც `get_future()`-ით მიღებული ასინქრონული დასაბრუნებელი მნიშვნელობა.
განვიხილოთ რამდენიმე მარტივი მაგალითი.

```
#include <iostream>
#include <cmath>
#include <thread>
#include <future>
#include <functional>

// unique function to avoid disambiguating the std::pow overload set
int f(int x, int y) { return std::pow(x, y); }

void task_lambda()
{
	std::packaged_task<int(int, int)> task([](int a, int b) {
		return std::pow(a, b);
	});
	std::future<int> result = task.get_future();

	task(2, 9);

	std::cout << "task_lambda:\t" << result.get() << '\n';
}

void task_bind()
{
	std::packaged_task<int()> task(std::bind(f, 2, 11));
	std::future<int> result = task.get_future();

	task();

	std::cout << "task_bind:\t" << result.get() << '\n';
}

void task_thread()
{
	std::packaged_task<int(int, int)> task(f);
```

```
	std::future<int> result = task.get_future();

	std::thread task_td(std::move(task), 2, 10);
	task_td.join();

	std::cout << "task_thread:\t" << result.get() << '\n';
}

int main()
{
	task_lambda();
	task_bind();
	task_thread();
}
```

შედეგით:

```
task_lambda:    512
task_bind:      2048
task_thread:    1024
Press any key to continue . . .
```

გარდა იმისა რაც ვთქვით, ვხედავთ რომ შეფუთული ამოცანა დადაადგილებადი ტიპისაა.

განვიხილოთ მეორე მაგალითი:

```
// packaged_task example
#include <iostream>     // std::cout
#include <future>       // std::packaged_task, std::future
#include <chrono>       // std::chrono::seconds
#include <thread>       // std::thread, std::this_thread::sleep_for

// count down taking a second for each value:
int countdown(int from, int to) {
	for (int i = from; i != to; --i) {
		std::cout << i << '\n';
		std::this_thread::sleep_for(std::chrono::seconds(1));
	}
	std::cout << "Lift off!\n";
	return from - to;
}

int main()
{
	std::packaged_task<int(int, int)> tsk(countdown);   // set up packaged_task
	std::future<int> ret = tsk.get_future();            // get future
	std::thread th(std::move(tsk), 10,2);   //spawn thread to count down from 10 to 2

	int value = ret.get();              // wait for the task to finish and get result
	std::cout << "The countdown lasted for " << value << " seconds.\n";

	th.join();
}
```

ეს ამოცანა წინას მესამე შეფუთულ ამოცანას ჰგავს, როდესაც დინებას ვაწვდით შეფუთულ ამოცანას. მომდევნო ამოცანაც იგივე სტილისაა:

```
#include <string>

// Fetch some data from DB
std::string getDataFromDB(std::string token)
{
	// Do some stuff to fetch the data
	std::string data = "Data fetched from DB by Filter :: " + token;
```

```
	return data;
}

int main()
{
	// Create a packaged_task<> that encapsulated the callback i.e. a function
	std::packaged_task<std::string(std::string)> task(getDataFromDB);

	// Fetch the associated future<> from packaged_task<>
	std::future<std::string> result = task.get_future();

	// Pass the packaged_task to thread to run asynchronously
	std::thread th(std::move(task), "Arg");

	// Join the thread. Its blocking and returns when thread is finished.
	th.join();

	// Fetch the result of packaged_task<> i.e. value returned by getDataFromDB()
	std::string data = result.get();

	std::cout << data << std::endl;
}
```

ახლა შეგვიძლია განვიხილოთ უფრო შინაარსიანი მაგალითი ჩვენი ძირითადი წყაროდან:

ლისტინგი 3.7 std::packaged_task-ის მაგალითი

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>
std::mutex m;
std::deque<std::packaged_task<void()> > tasks;
bool gui_shutdown_message_received();
void get_and_process_gui_message();
void gui_thread()                                  ← (1)
{
	while (!gui_shutdown_message_received())   ← (2)
	{
		get_and_process_gui_message();   ← (3)
		std::packaged_task<void()> task;
		{
			std::lock_guard<std::mutex> lk(m);
			if (tasks.empty())                    ← (4)
				continue;
			task = std::move(tasks.front());   ← (5)
			tasks.pop_front();
		}
		task();     ← (6)
	}
}
std::thread gui_bg_thread(gui_thread);
template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
	std::packaged_task<void()> task(f);          ← (7)
	std::future<void> res = task.get_future();     ← (8)
	std::lock_guard<std::mutex> lk(m);
	tasks.push_back(std::move(task));    ← (9)
	return res;                          ← (10)
```

}

GUI პლატფორმები ისეა მოწყობილი, რომ განახლებებს აკეთებს ამ საქმისთვის გამოყოფილი დინება. თუ სხვა დინებას უნდა რაღაცის განახლება, ამისთვის შესაბამისი გზავნილი უნდა გაუგზავნოს საჭირო დინებას.

ფუნქცია `gui_thread()` (1) ტრიალებს ვიდრე არ მიიღებს გზავნილს (2) რომ უნდა დაიხუროს GUI. ტრიალებს და კითხულობს GUI-ის გზავნილებს რათა დაამუშაოს იხინი (3). ასეთებია მომხმარებლის დაწკაპუნებები და ამოცანები ამოცანების რიგში. თუ რიგში არაა ამოცანა (4), მაშინ მარყუში ისევ ტრიალდება; სხვა შემთხვევაში იგი გახსნის ამოცანას რიგიდან (5), მოხსნის ჩაკეტვას რიგზე და შემდეგ გაუშვებს ამოცანას (6). ამოცანასთან დაკავშირებული დამდეგი გამზადდება ამოცანის დამთავრების შემდეგ.

ამოცანის რიგში გადასაგზავნად, მოწოდებული ფუნქციიდან (7) იქმნება ახალი შეფუთული ამოცანა, დამდეგი მიიღება ამოცანისგან `get_future()` მეთოდის გამოყენებით (8) და ამოცანა მოთავსდება რიგში (9) მანამდე, ვიდრე დამდეგი დაუბრუნდება გამომძახებელს (10). კოდი, რომელიც გადაგზავნის გზავნილს GUI -ის დინებისთვის, შემდეგ დაელოდება დამდეგს თუ უნდა გაიგოს შესრულდა თუ არა ამოცანა, ან გააუქმებს დამდეგს თუ არ აინტერესებს შედეგის გაგება.

ამ ამოცანაში ამოცანა ძალიან მარტივია, არც დასაბრუნებელი მნიშვნელობა აქვს და არც არგუმენტები სჭირდება. თუმცა შეფუთული ამოცანები კეთდება ბევრად რთული ამოცანებსითვის.

ამ ამოცანაში შედეგი მოდიოდა ერთი დინებიდან. შემდეგ ნაკვეთში განვიხილავთ დამდეგის შექმნის მესამე გზას: დაპირების ანუ `std::promise` -ის გამოყენებას.

<<< დაპირებების (`std::promise`) გაკეთება

დაპირება არის კლასის თარგი `std::promise<>`, რაც გვაძლევს საშუალებას შევინახოთ მნიშვნელობა ან გამონაკლისი, რაც შემდეგში ასინქრონულად მოითხოვება ამ კლასის ობიექტის მიერ შექმნილი `std::future` ობიექტის მიერ. იგულისხმება, რომ `std::promise<>`-ის გამოყენება მოხდება მხოლოდ ერთხელ.

აგების პროცესში, დაპირების ობიექტი უკავშირდება ახალ საზიარო ადგილს, სადაც ინახავს ან T ტიპის მნიშვნელობას, ან გამონაკლისს.

ეს საზიარო ადგილი შესაძლოა დაკავშირებული იყოს დამდეგ ობიექტთან, `get future`-ის გამოძახებით. გამოძახების შემდეგ ორივე ობიექტი იზიარებს ერთი და იმავე საზიარო ადგილს.

განვიხილოთ მარტივი მაგალითი http://www.cplusplus.com/reference/future/promise/-იდან:

```
// promise example
#include <iostream>       // std::cout
#include <functional>     // std::ref
#include <thread>         // std::thread
#include <future>         // std::promise, std::future
#include <chrono>
void print_int(std::future<int>& fut) {
      int x = fut.get();
      std::cout << "value: " << x << '\n';
}

int main()
{
      std::promise<int> prom;                      // create promise

      std::future<int> fut = prom.get_future();    // engagement with future
```

```
	std::thread th1(print_int, std::ref(fut));  // send future to new thread
	prom.set_value(10);                         // fulfill promise
                                                // (synchronizes with getting the future)
	th1.join();
}
```

როგორც ვხედავთ, დინებას გადაეწოდა ფუნქცია და ფუნქციის არგუმენტად დაპირებასთან დაკავშირებული დამდეგი ობიექტი. ამის შემდეგ „შესრულდა პირობა“, ანუ განისაზღვრა მნიშვნელობა. პირობის შესრულების შემდეგ შეასრულა დინებამ ფუნქციაში მითითებული ბეჭდვა. საკმარისია დაყოვნება ჩავსვათ დინების გაშვებასა და პირობის შესრულებას შემდეგ

```
	std::thread th1(print_int, std::ref(fut));  // send future to new thread
	std::this_thread::sleep_for(std::chrono::seconds(6));
	prom.set_value(10);                         // fulfill promise
```

და დავინახავთ რომ დინება მუშობას იწყებს მხოლოდ პირობის შესრულების შემდეგ.

განვიხილოთ მეორე მარტივი მაგალითი https://en.cppreference.com/w/cpp/thread/promise-იდან:

```
#include <vector>
#include <thread>
#include <future>
#include <numeric>
#include <iostream>
#include <chrono>

void accumulate(std::vector<int>::iterator first,
	std::vector<int>::iterator last,
	std::promise<int> accumulate_promise)
{
	int sum = std::accumulate(first, last, 0);
	accumulate_promise.set_value(sum);  // Notify future
}

void do_work(std::promise<void> barrier)
{
	std::this_thread::sleep_for(std::chrono::seconds(1));
	barrier.set_value();
}

int main()
{
	// Demonstrate using promise<int> to transmit a result between threads.
	std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };
	std::promise<int> accumulate_promise;
	std::future<int> accumulate_future = accumulate_promise.get_future();
	std::thread work_thread(accumulate, numbers.begin(), numbers.end(),
		std::move(accumulate_promise));
	// accumulate_future.wait();  // wait for result
	std::cout << "result=" << accumulate_future.get() << '\n';
	work_thread.join();  // wait for thread completion

	// Demonstrate using promise<void> to signal state between threads.
	std::promise<void> barrier;
	std::future<void> barrier_future = barrier.get_future();
	std::thread new_work_thread(do_work, std::move(barrier));
	barrier_future.wait();
	new_work_thread.join();
}
```

როგორც ვხედავთ, დაპირებასთან დაკავშირებული `std::future` -ის ობიექტს მივიღებთ `get_future()` მეთოდის გამოძახების შემდეგ, ისევე როგორც შეფუთული ამოცანის შემთხვევაში. როდესაც დაპირების მნიშვნელობა დფაყდება, დამდეგი მაშინვე გახდება მზა და მისი გამოყენება

შეიძლება მნიშვნელობის ამოსაღებად. თუ დავშალეთ პირობას მისი მნიშვნელობის შენახვის გარეშე, მაშინ მნიშვნელობის ნაცვლად შეინახება გამონაკლისი.

<<< გამონაკლისის შენახვა დამდეგისთვის

განვიხილოთ კოდის შემდეგი მოკლე ნაგლეჯი:

```
double square_root(double x)
{
	if (x < 0)
	{
		throw ("out_of_range: x<0");
	}
	else return sqrt(x);
}
```

წარმოვიდგინოთ, რომ ძირითად დინებაში ამ ფუნქციის

```
	double x, y;
	std::cout << "Plz, enter one double" << std::endl;
	std::cin >> x;
	try
	{
		y = square_root(x);
		std::cout << y << std::endl;
	}
	catch(const char* msg)
	{
		std::cerr << msg << std::endl;
	}
```

გამოძახების ნაცვლად, მას ვიძახებთ ასინქრონულ რეჟიმში:

```
	double x, y;
	std::cout << "Plz, enter one double" << std::endl;
	std::cin >> x;
	try
	{
		std::future<double> f = std::async(square_root, x);
		double y = f.get();
		std::cout << y << std::endl;
	}
	catch(const char* msg)
	{
		std::cerr << msg << std::endl;
	}
```

როდესაც ასინქრონულ რეჟიმში ფუნქციის გამოძახება გამოისვრის გამონაკლისს, იგი შეინახება დამდეგში შესანახი მნიშვნელობის ადგილზე, დამდეგი გახდება მზა და `get()`-ის გამოძახება ხელახლა გამოისვრის შენახულ გამონაკლისს. საინტერესოა, რომ სტანდარტი არ აკონკრეტებს არის ეს ორიგინალური გამონაკლისის ობიექტი თუ მისი ასლი.

std-ს პირობა იძლევა იგივე საშუალებას. თუმცა აქ საჭირო ხდება `try/catch` ბლოკის გამოყენება. ვნახოთ მაგალითი:

```
// promise::set_exception
#include <iostream>       // std::cin, std::cout, std::ios
#include <thread>         // std::thread
#include <future>         // std::promise, std::future
#include <exception>      // std::exception, std::current_exception
#include <cmath>

double square_root(double x)
{
	if (x < 0)
```

```
	{
		throw ("out_of_range: x<0");
	}
	else return sqrt(x);
}

void print_double(std::future<double>& f)
{
	try
	{
		double y = f.get();
		std::cout << y << std::endl;
	}
	catch (const char* msg)
	{
		std::cerr << msg << std::endl;
	}
}


int main()
{
	double x, y;
	std::cout << "Plz, enter one double" << std::endl;
	std::cin >> x;
	std::promise<double> prom;
	std::future<double> fut = prom.get_future();

	std::thread t(print_double, std::ref(fut));

	try
	{
		prom.set_value(square_root(x));
	}
	catch (...)
	{
		prom.set_exception(std::current_exception());
	}

	t.join();
}
```

დამდეგში გამონაკლისის შენახვა შეიძლება პირობის გაუქმების შედეგად.

`std::future` -ს აქვს თავისი შეზღუდვები. ძირითადი ისაა რომ მხოლოდ ერთ დინებას შეუძლია შედეგისთვის მოცდა. თუ გვინდა რომ ერთი და იმავე ხდომილებას დაელოდოს რამდენიმე დინება, მაშინ გვჭირდება საზიარო დამდეგის, ანუ `std::shared_future` -ის გამოყენება.

<<< std::shared_future -ის გამოყენება

ცალკე აღებული `std::future` -ის ნიმუშის მეთოდების გამოძახებები არაა ერთმანეთთან სინქრონიზებული. თუ ასეთ ობიექტში შეღწევა ხდება რამდენიმე დინებიდან ერთდროულად დამატებითი სინქრონზაციის გარეშე, ადგილი აქვს მონაცემების რბოლას და განუსაზღვრელ ყოფაქცევას. `std::future` ამოდელებს ეულ საკუთრებას ასინქრონულ შედეგზე. შესაბამისად ეს არის მხოლოდ-გადაადგილებადი ობიექტი. მისგან განსხვავებით, `std::shared_future` -ის ნუმუშები არიან ასლირებადები.

`std::shared_future` -ის ბეთოდები ინდივიდუალურ ობიექტზე კვლავ არასინქრონულებია. ამიტომ, თუ გვინდა ერთ ცალ ობიექტზე მრავალი დინებიდან შეღწევიას უნდა დავკეტოთ იგი.

თუმცა, უპირატესი გზა არის ყოველი დინებისთვის საზიარო დამდეგის თითო ასლის გადაწოდება.

`std::shared_future` -ის ნიმუშები, რომლებიც მიუთითებს რაიმე ასინქრონულ ადგილს, იგება `std::future` -ის ნიმუშებისგან რომელიც იგივე ადგილს მიუთითებს. ამ დროს საკუთრების უფლებაც გადადის გადაადგილების გამოყენებით, რის შემდეგაც `std::future` რჩება ცარიელ მდგომარეობაში, როგორც ნაგულისხმევი კონსტრუქტორით აგების შემთხვევაში იქნებოდა. მაგალითად:

```
std::promise<int> p;
std::future<int> f(p.get_future());
std::shared_future<int> sf(std::move(f));
```

გადაადგილება არ გახდება საჭირო, თუ get_future()-ის დასაბრუნებელ მნიშვნელობას გამოვიყენებთ:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future());
```

განვიხილოთ მაგალითი https://www.modernescpp.com/index.php/promise-and-future-იდან. ეს მაგალიტი რამდენიმე რამითაა საინტერესო: გვიჩვენებს თუ როგორ მუშავდება გამონაკლისები (თითქმის იგივეა რაც ზემოთ განვიხილეთ); გვიჩვენებს თუ როგორ იქმნება ეული დამდეგისგან:

```
std::promise<int> divPromise;
```

საზიარო დამდეგი (ისევე როგორც ზემოთ იყო ახსნილი)

```
std::shared_future<int> divResult = divPromise.get_future();
```

და შემდეგ დინებებს გადაეწოდებათ ამ საზიარო დამდეგის ასლები (უსაფრთხოების მიზნით);

გვიჩვენებს თუ როგორ ელოდება რამდენიმე დინება ერთ-ერთი სხვა დინებიდან პირობის შესრულებას:

```
// sharedFuture.cpp
#include <exception>
#include <future>
#include <iostream>
#include <thread>
#include <utility>

std::mutex coutMutex;

struct Div
{
	void operator()(std::promise<int>&& intPromise, int a, int b) {
		try
		{
			if (b == 0) throw std::runtime_error("illegal division by zero");
			intPromise.set_value(a / b);
		}
		catch (...)
		{
			intPromise.set_exception(std::current_exception());
		}
	}
};

struct Requestor
{
	void operator ()(std::shared_future<int> shaFut)
	{
```

```
                // lock std::cout
                std::lock_guard<std::mutex> coutGuard(coutMutex);

                // get the thread id
                std::cout << "threadId(" << std::this_thread::get_id() << "): ";

                // get the result
                try
                {
                        std::cout << "20/10= " << shaFut.get() << std::endl;
                }
                catch (std::runtime_error& e) {
                        std::cout << e.what() << std::endl;
                }
        }
};

int main() {

        std::cout << std::endl;

        // define the promises
        std::promise<int> divPromise;

        // get the futures
        std::shared_future<int> divResult = divPromise.get_future();

        // calculate the result in a separat thread
        Div div;
        std::thread divThread(div, std::move(divPromise), 20, 10);

        Requestor req;
        std::thread sharedThread1(req, divResult);
        std::thread sharedThread2(req, divResult);
        std::thread sharedThread3(req, divResult);

        divThread.join();

        sharedThread1.join();
        sharedThread2.join();
        sharedThread3.join();

        std::cout << std::endl;
}
```

მხოლოდ ძირითადი საკითხზე ყურადღების გასამახვილებლად, შესაძლოა უმჯობესი იყოს, თუ ფუნქციებს შედარებით გავამარტივებთ (ნამრავლს ვიანგარიშებთ) და გამონაკლისებს ავერიდებით:

```
struct Div
{
        void operator()(std::promise<int>&& intPromise, const int a, const int b)
        {
                intPromise.set_value(a * b);
        }
};

struct Requestor
{
        void operator ()(std::shared_future<int> shaFut)
        {
                // lock std::cout
```

```
            std::lock_guard<std::mutex> coutGuard(coutMutex);

            // get the thread id
            std::cout << "threadId(" << std::this_thread::get_id() << "): ";

            // get the result
            std::cout << "20*10= " << shaFut.get() << std::endl;
        }
};
```

<<< ლოდინი დროსთან დაკავშირებულ შეზღუდვებში

აქამდე განხილული ნებისმიერი შემაჩერებელი გამოძახებს მუშაობდა დროის განუსაზღვრელი შუალედის განმავლობაში, აჩერებდა რა დინებას ვიდრე შესრულდებოდა ის ხდომილება, რომელსაც დინება ელოდებოდა. მაგრამ ბევრ შემთხვევაში საჭიროა რომ განვსაზღვროთ თუ რამდენად დიდხანს უნდა დაველოდოთ, ანუ რამდენად ხანგრძლივი შესვენება უნდა დავგეგმოთ.

არსებობს შესვენებების რამდენიმე სახე: *ხანგრძლივობაზე დაფუძნებული* შესვენება, სადაც იცდით დროის განსაზღვრული რაოდენობის განმავლობაში (მაგალითად 15 მილიწამი); ან უპირობო შესვენება, როდესაც იცდით დროის განსაზღვრულ მომენტამდე (მაგალითად, 2019 წლის 14 სექტემბრის 9.00 საათამდე). დამლოდებელი ფუნქციების უმეტესობას აქვს ვარიანტები ორივე სახის შესვენების დასამუშავებლად. ვარიანტებს, რომლებიც ხანგრძლივობაზეა დაფუძნებული, აქვთ `_for` თანდებული, ხოლო რომლებიც ამუშავებენ უპირობო შესვენებებს - აქვთ `_until` თანდებული.

<<< საათები

საათი არის კლასის თარგი, რაც გვაწვდის ოთხი სახის ცნობებს:

- *მიმდინარე* დრო
- საათისგან მიღებული დროის წარმოსადგენი ტიპი
- საათის წიკ-წიკის ხანგრძლივობა
- არის თუ არა წიკ-წიკი თანაბარი ხანგრძლივობის და არის თუ არა საათის *მდგრადი*

მიმდინარე დროს ვღებულობთ საათის კლასის `now()` სტატიკური მეთოდის გამოძახებით. მაგალითად, `std::chrono::steady_clock::now()` აბრუნებს მდგრადი საათის მიმდინარე დროს.

კონკრეტული საათის დროის წერტილის ტიპი განისაზღვრება საათის `time_point` მეთოდით. მაგალითად:

```
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();
```

საათის წიკ-წიკის ხანგრძლივობა განისაზღვრება წამის განზომილების მქონე წილადით, რომელიც ტიპიც მოიცემა საათის `period` მეთოდით. თუ საათი წიკწიკებს წამში 25-ჯერ, მაშინ მისი პერიოდი არის `td::ratio<1,25>`, ხოლო თუ წიკწიკებს ყოველ 1.5 წამში, მისი პერიოდია `td::ratio<3,2>`. არაა გარანტია, რომ მითითებული პერიოდი დაემთხვევა კონკრეტული გაშვების დროს რეალურად განხორციელებული წიკწიკის პერიოდს.

თუ საათი წიკწიკებს თანაბარი სიხშირით (სულერთია ემთხვევა თუ არა მითითებულ პერიოდს) და საათი ვერ რეგულისრდება, მაშინ საათი ითვლება მდგრადად. საათის `is_steady` სტატიკური მეთოდი თუ ჭეშმარიტია, მაშინ საათი მდგრადია და პირიქით. ჩვეულებრივ, `std::chrono::system_clock` ვერ იქნება მდგრადი, რადგან მისი დარეგულირება შიძლება. C++-ის ერთ-ერთი მდგრადი საათი არის `std::chrono::steady_clock`. საათი `std::chrono::system_clock` წარმოადგენს სისტემის „რეალური-დროის“ საათს და რომელისაც აქვს ფუნქციები დროის წერტილების `time_t` ტიპის მნიშვნელობებზე გადაყვანისთვის პირიქით გადმოყვანისთვის. ენაში არის აგრეთვე

`std::chrono::high_resolution_clock` საათი, რომელიც იძლევა წიკწიკის უმცირეს შესაძლო ხანგრძლივობას (და უდიდეს შესაძლო გარჩევადობას) ბიბლიოთეკის საათებს შორის. ეს საათები განსაზღვრულია ბიბლიოთეკაში `<chrono>` სათაურით.

<<< ხანგრძლივობები (durations)

ხანგრძლივობები დამუშავებულია `std::chrono::duration<>` კლასის თარგში

```
template<class Rep, class Period = std::ratio<1>>
class duration;
```

თარგის პირველი პარამეტრი არის წარმოდგენის ტიპი (int, long, double და ასეთები) და განსაზღვრავს ხანგრძლივობის რიცხვით მნიშვნელობას.

მაგალითად, განაცხადი

```
typedef std::chrono::duration<short, std::ratio<1, 1000000>> clockoid;
clockoid t{ SHRT_MAX+ };
std::cout << t.count() << std::endl;
```

გამართულია და ბეჭდავს მოკლე მთელის მაქსიმალურ მნიშვნელობას. სამაგიეროდ,

```
clockoid t{ SHRT_MAX+1 };
```

ასეთი ცვლილება უკვე იწვევს გადავსებას. ასევე გასაგებია, რომ როდესაც `Rep` არის ნამდვილი, მაშინ შესაძლებელია რომ ხანგრძლივობამ მიიღოს ნამდვილი მნიშვნელობები.

მეორე პარამეტრი არის პერიოდი, წილადი, რომელიც განსაზღვრავს თუ ხანგრძლლივობის თითოეული ერთეული რამდენ ხანს გრძელდება. პერიოდის ნაგულისხმევი მნიშვნელობა არის `std::ratio<1>`, იგივე `std::ratio<1,1>` და გამოხატავს ერთ წამს. `std::ratio<60>` არის წუთი, ხოლო `std::ratio<1,1000>` არის მილიწამი.

C++ ენაში ჩადებულია რამდენიმე მნიშვნელოვანი ხანგრძლივობა:

```
typedef duration<signed int, nano> nanoseconds;
typedef duration<signed int, micro> microseconds;
typedef duration<signed int, milli> milliseconds;
typedef duration<signed int> seconds;
typedef duration<signed int, ratio< 60>> minutes;
typedef duration<signed int, ratio<3600>> hours;
```

C++14-ში, სახელთა სივრცეში `namespace std::literals::chrono_literals` განსაზღვრულია აგრეთვე:

ტიპი	თანდებული	მაგალითი
`std::chrono:: hours`	h	3h
`std::chrono::minuts`	m	7m
`std::chrono::seconds`	s	55s
`std::chrono::milliseconds`	ms	2432ms
`std::chrono::microseconds`	us	5us
`std::chrono::nanoseconds`	ns	342300

მაგალითად,

```
#include <iostream>
#include <chrono>

using namespace std::literals::chrono_literals;

int main()
{
	auto t = 4min;
```

```
        std::cout  << "Time in minutes: " << t.count() <<  std::endl;
}
```

სხვადასხვა ერთეულებსი გაზომილი ხანგრძლივობების ერთმანეთზე გადაყვანა თავისით ხდება, ტუ ოპერაცია არ მოითხოვს წაკვეცას. სხვა შემთხვევაში ცხადი გადაყვანა გვიწევს `std::chrono::duration_cast<>`-ის გამოყენებით. მაგალითად,

```
        std::chrono::milliseconds ms(54805);
        std::chrono::seconds s = std::chrono::duration_cast<std::chrono::seconds>(ms);
```

შედეგი მოიკვეცება (არ დამრგვალდება). შედეგად მივიღებთ 54-ს.

ხანგრძლივობაზე დაფუძნებული მოცდები გაკეთებულია `std::chrono::duration<>`-ით. მაგალითად, თუ გვინდა რომ 30 წამამდე დაველოდოთ დამდეგის გამზადებას:

```
        std::future<int> f = std::async(f);
        if (f.wait_for(std::chrono::milliseconds(30)) == std::future_status::ready)
                do_something_with(f.get());
```

ყველა მოცდის ფუნქცია აბრუნებს სტატუსს რაც აჩვენებს მოლოდინის დრო დასრულდა თუ ის ხდომილება მოხდა რასაც ვუცდიდით. ამ შემთხვევაში, ჩვენ ველოდებით დამდეგს და ფუნქცია დააბრუნებს `std::future_status::timeout`-ს თუ მოცდის დრო ამოიწურა, `std::future_status::ready`-ს, თუ დამდეგი მზადაა, ან `std::future_status::deferred`-ს თუ დამდეგის ამოცანა შეჩერებულია. ხანგრძლივობაზე დაფუძნებული მოცდების დრო იზომება მდგრადი საათით.

<<< დროის მომენტები

დროის მომენტი არის კლასის თარგი

```
template< class Clock, class Duration= typename Clock::duration>
class time_point;
```

ამ კლასის ნიმუში არის დროის მომენტი და იგი ხასიათდება იმით, თუ რომელი საათია გამოყენებული და რა ერთეულებშია გაზომილი დროის ხანგრძლივობა. დროის მომენტის მნიშვნელობა არის დროის შუალედის სიგრძე (მიტითებულ ერთეულებში) დროის გარკვეული შუალდიდან, რომლსაც ეწოდება ამ საათის *ეპოქა*. ეპოქის გამოთვლა შეიძლება. მისი ტიპური მნიშვნელობა არის 1970 წლის 1 იანვარი.

მაგალითად, შეგვიძლია მივუთითოთ დროის მომენტი როგორც

```
        std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>.
```

დროის მომენტებს შეგვიძლია დავამატოთ ან გამოვაკლოთ დროის ხანგრძლივობები. დროის მომენტების გამოკლებით ვღებულობთ ხანგრძლივობებს. მაგალითად, კოდის ფრაგმენტის შესრულების ხანგრძლივობის გამოსათვლელად გამოიყენება რაიმე ასეთი:

```
        auto st = std::chrono::high_resolution_clock::now();
        . . . . . .
        auto fin = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::milliseconds>(fin-st);
        std::cout << "Time: " << time.count()  << " milliseconds" << std::endl;
```

როდესაც `wait` ფუნქციას (უპირობო შესვენებით) გადავაწვდით `std::chrono::time_point<>`-ის ნიმუშს, დროის მომენტის დროის პარამეტრი ზომავს დროს და `wait` ფუნქცია არ დაბრუნდება ვიდრე საათის `now()` ფუნქცია არ დააბრუნებს მითითებულ შესვენების შემდეგ დროის მომენტს.

დროის მომენტები გამოიყენება `wait` ფუნქციის `_until` ვარიანტებთან. მაგალითად, თუ საჭიროა რომ პირობიან ცვლადთან დაკავშირებულ რაიმე ხდომილებას დაველოდოთ მაქსიმუმ 300 მილლიწამი, მაშინ უნდა მოვიქცეთ შემდეგნაირად:

```
#include <condition_variable>
```

```
#include <mutex>
#include <chrono>
std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
	auto const timeout = std::chrono::steady_clock::now() +
		std::chrono::milliseconds(300);
	std::unique_lock<std::mutex> lk(m);
	while (!done)
	{
		if (cv.wait_until(lk, timeout) == std::cv_status::timeout)
			break;
	}
	return done;
}
```

ეს არის შემოწმებული გზა იმ შემთხვევისთვის, როდესაც პირობიან ცვლადს არ ვაწვდით პრედიკატს. თუ გამოვიყენებდით `wait_for()`-ს, მაშინ შესაძლოა მოხდეს ცრუ გაღვიძება თითქმის ბოლოში, რის შემდეგაც მოლოდინის დრო თავიან იწყებს ათვლას. თუ ასე ბევრჯერ გამეორდა, მტლიანი დრო პრაქტიკულად შემოუსაზღვრელი იქნება.

<<< ფუნქციები, რომლებიც აღიარებენ შესვენებებს

შესვენებების უმარტივესი გამოყენება არის რომელიმე კერძო დინების დაყოვნება. ამას აკეთებს ორი ფუნქცია: `std::this_thread::sleep_for()`,`std::this_thread::sleep_until()`. დინება იზინება ან განსაზღვრული ხანგრძლივობით (`sleep_for()`-ის შემთხვევა), ან გარკვეულ დროის მომენტამდე (`sleep_until()`-ის შემთხვევა). შესვენებების გამოყენება ზოგავს რესურსებს (როდესაც დინებას არაფერი აქვს საკეთებელი, დაიძინებს და არ დახარჯავს იმას რასაც სხვები გამოიყენებენ). სხვა თემა არის შესვენებების გამოყენება დამდეგებთან და პირობიან ცვლადებთან.

შემდეგი ცხრილი აღწერს C++ -ის ფუნქციებს, რომლებიც აღიარებენ შესვენებებს.

კლასი/სახელთა სივრცე	ფუნქციები
`std::this_thread` namespace	`sleep_for(`*`duration`*`)` `sleep_until(`*`time_point`*`)`
`std::condition_variable` `std::condition_variable_any` `wait_for(`*`lock`*`,`*`duration`*`)` `wait_until(`*`lock`*`,`*`time_point`*`)`	`std::cv_status::timeout` `std::cv_status::no_timeout` `wait_for(`*`lock`*`,`*`duration`*`,`*`predicate`*`)` *`wait_until(lock,time_point,`* *`predicate`*`)`
`std::timed_mutex`, `std::recursive_timed_mutex` or `std::shared_timed_` `mutextry_lock_for(`*`duration`*`)` `try_lock_until(`*`time_point`*`)`	`bool—true` თუ იყო ჩაკეტილი `false`--სხვა შემთხვევაში
`std::shared_timed_mutex`	`try_lock_shared_for(`*`duration`*`)` `try_lock_shared_until(`*`time_point`*`)`
`std::unique_lock<TimedLock` `able>unique_lock(`*`lockable`*`,` *`duration`*`)` `unique_lock(`*`lockable`*`,`*`time_`* *`point`*`)`	N/A—`owns_lock()` ახალ ობიექტზე აბრუნებს `true` თუ ჩაიკეტა, სხვა შემთხვევაში - `false` `try_lock_for(`*`duration`*`)` `try_lock_until(`*`time_point`*`)`
`std::shared_lock<`*`Shared-`* *`TimedLockable`*`>shared_lock`	N/A—`owns_lock()` ახალ ობიექტზე აბრუნებს `true` თუ ჩაიკეტა, სხვა შემთხვევაში - `false`

(*lockable*,*duration*) shared_lock(*lockable*,*time_point*)	 try_lock_for(*duration*) try_lock_until(*time_point*)
std::future<*ValueType*> or std::shared_future<*Value-Type*>wait_for(*duration*) wait_until(*time_point*)	std::future_status::timeout თუ დრო ამოიწურა, std::future_status::ready თუ დამდეგი მზადაა std::future_status::deferred თუ დამდეგი ამუშავებს დაყოვნებულ ფუნქციას, რომელიც ჯერ არ დაწყებულა

<<< სავარჯიშოები