

`std::unique_lock`

- владельца можно заменить
- но гарантирует: в каждый момент времени мьютексом может владеть только одна обертка **`std::unique_lock`** !
- => **`movable`**

Замечание:

если на момент вызова
`unique_lock& operator=(unique_lock&& other);`
у обертки уже был залоченный мьютекс, то он освобождается – `unlock()`

std::unique_lock – обертка для мьютекса с бОльшей функциональностью по сравнению с **lock_guard**.

std::lock_guard	std::unique_lock
заблокирован на всем протяжении своего существования	добавляет признак + возможность узнать свободен/захвачен мьютекс - owns_lock()
мьютекс блокируется один раз в конструкторе (или уже захвачен), освобождается тоже только один раз – в деструкторе	можно блокировать и освобождать посредством lock() и unlock() , а также try_lock() , try_lock_for() , try_lock_until
политика захвата мьютекса может быть только std::adopt_lock	может быть adopt_lock , defer_lock или try_to_lock
копирование, присваивание и перемещение запрещены	предоставляет перемещающие операции

По сравнению с `lock_guard` **`std::unique_lock` :**

- Занимает больше памяти (для хранения флага)
- Чуть медленнее, чем `lock_guard` (за счет выполнения дополнительных проверок флага)

=> менее эффективен

Функциональность **std::unique_lock**:

- `explicit unique_lock(mutex_type& m);` - `m.lock()`
- `unique_lock(unique_lock&& other);` - можно
- `unique_lock(mutex_type& m, std::adopt_lock_t t);` - аналогично `lock_guard`
- `unique_lock(mutex_type& m, std::defer_lock_t t);` - отложить захват мьютекса (если до вызова деструктора мьютекс не будет занят, то `unlock()` не вызывается). Захватить `mutex` можно позже посредством метода `lock()`
- `unique_lock(mutex_type& m, std::try_to_lock_t t);` - неблокирующая попытка захвата мьютекса + установка признака

Специфика `unique_lock`:

- `mutex_type* release();`
 - не выполняется `unlock()`
 - возвращается указатель на ассоциированный мьютекс или `nullptr`
 - обертка больше не ассоциируется с мьютексом
- `explicit operator bool() const; //- true, если`
 - мьютекс ассоциирован с объектом
 - и мьютекс захвачен
- `mutex_type* mutex() const; //`возвращается указатель на ассоциированный мьютекс или `nullptr`

Пример:

```
std::mutex m;
```

```
std::unique_lock<std::mutex> ul1;
```

```
if(ul1) ???
```

```
std::unique_lock<std::mutex> ul2(m);
```

```
if(ul2) ???
```

```
ul2.unlock();
```

```
if(ul2) ???
```

```
//или
```

```
ul2.release();
```

```
if(ul2) ???
```

Пример:

```
std::mutex m;
```

```
std::unique_lock<std::mutex> ul(m, std::defer_lock);
```

```
bool b = ul.owns_lock(); //???
```

```
ul.lock();
```

```
b = ul.owns_lock(); //???
```

Гранулярность защищаемого кода

Защищаемая секция должна быть
как можно **меньше!**

Модифицируем A::swap() с помощью std::unique_lock

```
void A::swap(A& other){  
    ...  
    //обмен  
}
```

Дополнительные возможности `unique_lock`. Конструкторы:

Замечание: только для `std::timed_mutex`!

- `template <class Rep, class Period>`
`unique_lock` (`mutex_type& m`,
`const chrono::duration<Rep,Period>& rel_time`);
вызывает `m.try_lock_for(rel_time)`
- `template <class Clock, class Duration>`
`unique_lock` (`mutex_type& m`,
`const chrono::time_point<Clock,Duration>& abs_time`);
– `m.try_lock_until(abs_time)`

std::timed_mutex

- lock()
- unlock()
- try_lock()

Добавляет по сравнению с std::mutex методы:

- **try_lock_for()** — если мьютекс занят, блокируется до истечения интервала или освобождения мьютекса
- **try_lock_until()**

Пример

```
void threadFunc(std::timed_mutex& m)
{
    bool b = m.try_lock_for(std::chrono::seconds (10));
    if(b){
        //... //под защитой
        m.unlock();
    }
    else //интервал истек, а мьютекс так и не освободился
}
```

Обертки для `std::timed_mutex`

- `unique_lock` - ???
- `lock_guard` - ???

lock_guard и timed_mutex

```
std::lock_guard<std::timed_mutex> ul1(m);  
    //ОК, но смысла не имеет
```

```
std::lock_guard<std::timed_mutex> ul2(m,  
    std::chrono::seconds(10));  
    //ошибка – такого конструктора нет!
```

Использование обертки для timed_mutex

```
void f(std::timed_mutex& m)
{
    std::unique_lock<std::timed_mutex>
        lk(m, std::chrono::milliseconds(3)); //ожидает до 3 мс
    if(lk) {...} //если блокировка получена, обрабатываем данные
} // ???
```

unique_lock в качестве обертки для mutex и timed_mutex

```
using namespace std::chrono_literals;  
std::mutex m;  
std::unique_lock<std::mutex> ul1(m); //OK  
//ul1.try_lock_for(1s); //ошибка - у mutex нет try_lock_for
```

```
std::timed_mutex tm;  
std::unique_lock<std::timed_mutex> ul2(tm);  
ul2.try_lock_for(1s);
```


`std::recursive_mutex`

- позволяет потоку-владельцу мьютекса многократно вызывать `lock()`
- следствие -> столько же раз нужно вызвать `unlock()`

Рекомендация: не использовать рекурсивные мьютексы. Если таковые понадобились, то стоит подумать об ошибках проектирования

???

std::mutex m; //обычный мьютекс

```
void f1(){
    m.lock();
    ...
    m.unlock();
}
void f2(){
    m.lock();
    f1();
    m.unlock();
}
```

Пример std::recursive_mutex

```
std::recursive_mutex m;
```

```
void f1(){  
    m.lock();  
    ...  
    m.unlock();  
}
```

```
void f2(){  
    m.lock();  
    f1();  
    m.unlock();  
}
```

std::recursive_timed_mutex

Дополнительная функциональность:

- try_lock()
- try_lock_for()
- try_lock_until()

C++17 shared_mutex

#include <shared_mutex>

Не поддерживается в VS15

Часто возникает задача:

- позволить нескольким потокам осуществлять **чтение** (разделение мьютекса) – при этом **нельзя только модифицировать**
- и только одному – **запись** (владение мьютексом) – при этом **нельзя ни читать, ни писать!**

=> **rw - мьютексы**

Инициализация shared_mutex:

- `shared_mutex();` //создает мьютекс в свободном состоянии
- `shared_mutex(const shared_mutex&)`
`= delete;`

Специфика:

Для **ЭКСКЛЮЗИВНОГО** использования (владение):

lock(), unlock(), try_lock()

Для **СОВМЕСТНОГО** использования (разделение):

**lock_shared(), unlock _shared(),
try_lock_shared()**

Замечание:

- в некоторых реализациях количество разделяемых пользователей может быть ограничено =>
- при превышении количества предусмотренных «читателей» очередной «читатель» блокируется, пока кем-то не будет вызван `unlock_shared()`

Простой пример:

```
class Shared {  
    int m_a=0;  
    mutable std::shared_mutex m;  
public:  
    int Get()const{ //читать могут все потоки, если никто не пишет  
        m.lock_shared();//если мьютекс заблокирован для записи, то засыпаем  
        int a = m_a;  
        m.unlock_shared();  
        return a;  
    }  
    void Set(int a) {//только один поток может модифицировать  
        m.lock();  
        m_a=a;  
        m.unlock();  
    }  
};
```

Продолжение простого примера:

```
class Shared {  
    int m_a=0;  
    mutable std::shared_mutex m;  
public:  
    Shared(const Shared& other); //???  
    ...  
};
```

Класс-обертка для `shared_mutex` – `std::shared_lock`

- в конструкторе - ???
- в деструкторе - ???

Обертка для shared_mutex – **shared_lock – C++14**

- **конструкторы:**
 - `explicit shared_lock(mutex_type& m); // m.lock_shared()`
 - в зависимости от параметра `defer_lock`, `try_to_lock`, `adopt_lock`
 - конструктор, принимающий интервал или момент времени
- **деструктор** – если мьютекс захвачен, `unlock_shared()`

Переписываем пример:

```
class Shared {  
    int m_a=0;  
    mutable std::shared_mutex m;  
public:  
    int Get()const{ //читать могут все потоки  
        std::shared_lock<std::shared_mutex> lock(m);  
        return m_a;  
    }  
    void Set(int a) {//только один поток может модифицировать  
        std::unique_lock<std::shared_mutex> lock(m);  
        m_a=a;  
    }  
};
```

C++14 shared_timed_mutex

#include <shared_mutex>

Добавляет функциональность:

Для эксклюзивного использования:

lock(), unlock(), try_lock(), **try_lock_for()**,
try_lock_until()

Для общего использования:

lock_shared(), unlock_shared(),
try_lock_shared(), **try_lock_shared_for()**,
try_lock_shared_until()

Пример использования r/w мьютекса

```
class Shared{
    mutable std::shared_timed_mutex m;
    Data sharedData;
public:
    Data read()const{
        std::shared_lock< std::shared_timed_mutex > sl(m);
        return sharedData;
    }
    void write(Data newData){ //по значению безопаснее
        std::lock_guard< std::shared_timed_mutex> sl(m);
        sharedData = newData;
    }
    ...
};
```

Продолжение примера: ???

Реализация?

```
Shared& Shared::operator= (const Shared& );
```


Обертка `std::scoped_lock` – C++17

`<mutex>`

```
template< class... MutexTypes >
```

```
    class scoped_lock;
```

В конструкторе – гарантированный захват всех перечисленных в качестве параметров мьютексов посредством функции `lock()`

В деструкторе – освобождение (в обратном захвату порядке)

Переписываем
`void A::swap(A& other);`

???

Рекомендации –блокировки и не потерять эффективность :

- разумно выбирать гранулярность защищаемого кода
- по возможности использовать один мьютекс (a.lock(); b.lock() - гонки)
- если требуется захват нескольких мьютексов, то порядок захвата везде должен быть одинаков, иначе – гонки
- избегать вызова другого пользовательского кода в защищенной секции
- в сложном многопоточном приложении использовать **иерархические** мьютексы (назначать мьютексам разные уровни и позволять захватывать только в порядке понижения уровня).

Локальная память потока

C++11 СПЕЦИФИКАТОР КЛАССА ХРАНЕНИЯ `THREAD_LOCAL`

Спецификаторы класса хранения C++ ???

- До стандарта C++11 ???
- **C++11** - спецификатор класса хранения **thread_local** (ключевое слово => подключать заголовочные файлы не требуется)

Время жизни

- все переменные, определенные со спецификатором `thread_local`, обладают потоковым временем жизни – **thread storage duration**
- `thread storage duration` имеет много общего со статическим временем жизни (`static storage duration`). Отличие:
 - статическое время жизни – до завершения программы
 - потоковое время жизни – до завершения потока

Инициализация “блочных” `thread_local` переменных
(похожа на инициализацию статических переменных)

```
void ThreadFunc(){  
    for(int i=0; i<10; i++)    {  
        thread_local int z=33;  
        z++;  
        ...  
    }  
}
```

- инициализируется один раз
- при первом выполнении в каждом потоке!
- уничтожается при завершении потока

Замечание:

```
void ThreadFunc()  
{  
    ...  
    [static] thread_local int z=33; //избыточно  
    z++;  
    ...  
}
```


«Глобальные» thread_local переменные

1.cpp

```
extern thread_local int x;  
int main()  
{  
    // x = ?  
    x++;  
    // x = ?  
    std::thread t1(ThreadFunc);  
    std::thread t2(ThreadFunc);  
    t1.join();    t2.join();  
    // x = ?  
    ...  
}
```

2.cpp

```
thread_local int x;  
void ThreadFunc()  
{  
    // x = ?  
    x++;  
    // x = ?  
    ...  
}
```

Разница thread_local и static:

1.cpp

```
thread_local int x;  
static thread_local int y;
```

2.cpp

```
extern thread_local int x;  
extern thread_local int y;
```

```
void f2(){  
    x++; //OK  
    y++; //???  
}
```

Статические thread_local переменные класса

```
class A {  
    static thread_local size_t count;  
public:  
    ...  
};
```

Разница:

A.h class A{ static size_t count; public: ... };	A.h class A{ static thread_local size_t count; public: ... };
A.cpp #include "A.h" size_t A::count; //на этапе компиляции в единственном экземпляре	A.cpp #include "A.h" thread_local size_t A::count; //на этапе выполнения для каждого потока

Нельзя объявлять со спецификатором `thread_local`:

- параметры функций
- переменные со спецификатором `register`
- не статические переменные класса

```
void f(thread_local int x); //ошибка
```

```
class A {  
    thread_local size_t count; //ошибка  
};  
int f1(){  
    thread_local register int y; //ошибка  
}
```

Специфика `thread_local` объекта

- копия `thread_local` переменной, определенной вне функции, должна быть проинициализирована (сконструирована) к моменту первого использования (на момент запуска потока или непосредственно перед первым использованием – стандарт не оговаривает)
- => может вычисляться во время выполнения
- => не может быть `constexpr`
- неявная инициализация `thread_local` переменных базового типа – «0»
- если при конструировании `thread_local` переменной пользовательского типа генерируется исключение, то вызывается `terminate()`
- передавать адрес другому потоку можно, но НЕ безопасно!

Пример использования `thread_local`

- при реализации иерархического мьютекса для решения проблемы dead-lock-ов

Иллюстрация проблемы взаимоблокировки (псевдокод)

Thread A	Thread B
<pre>Lock(x) Lock(y) //Используем X и Y Unlock(y) Unlock(x)</pre>	<pre>Lock(y) Lock(x) //Используем X и Y Unlock(x) Unlock(y)</pre>

Deadlock возможен, если

потоки блокируют более одного mutex-а:

- два mutex-а А и В защищают два разных ресурса, и, двум потокам, одновременно необходим доступ к этим двум ресурсам
- Блокировка одного mutex-а - атомарна, но блокировка двух mutex-ов - это два отдельных действия => если первый поток заблокирует mutex А, в то время, как второй заблокирует mutex В, оба потока «зависнут», ожидая друг друга.

Реализация иерархического мьютекса

- В C++11-14-17 отсутствует иерархический мьютекс
- Для его реализации нужно учесть:
 - для каждого потока должен быть индивидуальный «счетчик» уровней мьютексов в данном потоке => **TLS**
 - при этом нужно гарантировать, что в потоке нет двух мьютексов с одинаковым “приоритетом”
 - уровни логично задавать не случайно, а в соответствии с важностью защищаемого участка кода

Пример реализации иерархического мьютекса:

// hierarch_mutex.h

```
class hierarch_mutex{
    mutable std::mutex m;
    int prev_level;
    const int this_level;
    static thread_local int thread_cur_level;
public:
    hierarch_mutex(int l) : this_level(l), prev_level(0){}
    void lock();
    void unlock();
};
```

// hierarch_mutex.cpp

```
thread_local int hierarch_mutex ::thread_cur_level = INT_MAX;
```

Продолжение примера

```
void hierarch_mutex ::lock() {  
    if(thread_cur_level <= this_level)  
        {throw "error level";}   
    m.lock();  
    prev_level = thread_cur_level ;  
    thread_cur_level = this_level;  
}
```

Продолжение примера

```
void hierarch_mutex ::unlock() {  
    thread_cur_level = prev_level;  
    m.unlock();  
}
```

Продолжение примера

```
bool hierarch_mutex ::try_lock()  
    if(thread_cur_level <= this_level)  
        {throw "error level";} //или return false;  
    if(!m.try_lock()){return false;}  
    prev_level = thread_cur_level ;  
    thread_cur_level = this_level;  
    return true;  
}
```

Джеффри Рихтер:

«Не нужно плодить потоки, потому что переключение контекстов - вещь дорогая»

THREAD POOL

Проблемы нерационального использования потоков

Если для решения часто возникающих **небольших** заданий для каждого задания запускается отдельный поток:

- поток создается непосредственно при поступлении задания и разрушается по завершении его обработки => порождение и уничтожение потока приводят к **значительным издержкам**
- требуется «**вручную**» задавать/определять количество одновременно работающих потоков
- для некоторых задач требуется организовать досрочное завершение потока (в классе `std::thread` не реализовано)

Шаблон проектирования «пул потоков»

- позволяет создавать **разумное** количество потоков
- и многократно их использовать, пока существует пул
- при уничтожении самого пула потоки также удаляются.

В результате экономятся ресурсы, которые до этого шли на создание, переключение и уничтожение потоков.

Идея пула потоков:

- создаем N (как выбираем N?) потоков и помещаем их в хранилище (пул).
- по мере возникновения новой задачи «достаем» из хранилища готовый поток и отдаем ему задачу на выполнение (при этом не происходит каждый раз создание/уничтожение => **при частых обращениях выигрыш**)
- как только поток отрабатывает отведенную ему задачу - он высвобождается (но продолжает существовать) => можно использовать его для выполнения следующей задачи.

Простейшая реализация пула потоков

Сущности:

- рабочий поток,
- пул потоков,
- задание
- очередь заданий

В данной упрощенной реализации не предусмотрены:

- получение результатов выполнения задания
- генерация исключений заданиями

```
class thread_pool {  
    mutable std::mutex m; //для безопасной многопоточной работы с очередью  
    std::queue<std::function<void()>> tasks; //извне не  
        используется, а в методах обеспечивает корректный доступ  
        к разделяемым данным посредством мьютекса  
    std::vector<std::thread> threads; //совокупность рабочих потоков  
    bool stop;  
    void task_thread(<параметры>); //потоковая функция  
  
public:  
    thread_pool();  
    ~thread_pool();  
    void add_task(<параметры>);  
};
```

```
void thread_pool::task_thread(){  
    while (!stop){  
        std::function<void()> task;  
        m.lock();  
        if (tasks.empty()) {  
            m.unlock();  
            std::this_thread::yield(); //отдыхаем  
        }else {  
            task = tasks.front();  
            tasks.pop();  
            m.unlock();  
        }  
        if (task) task();  
    }  
}
```

```
thread_pool::thread_pool() {  
    size_t nThreads = <количество потоков в пуле>;  
    //Запускаем потоки:  
    for (size_t i = 0; i < nThreads; i++){  
        threads.emplace_back(&thread_pool::task_thread, this);  
        //может быть сгенерировано исключение => по-хорошему  
        нужно обработать  
    }  
}
```

```
thread_pool::~~thread_pool(){  
    ???  
}
```

Продолжение

```
void thread_pool:: add_task(параметры)
{
    m.lock();
    tasks.push(<задание_с_полученными_параметрами>);
    m.unlock();
}
```


#include <conditional_variable>

УСЛОВНЫЕ ПЕРЕМЕННЫЕ

Если один поток должен дожидаться
выполнения условия другим потоком:

```
bool ReadyFlag = false;  
std::mutex m;
```

```
void thread1(){  
    m.lock();  
    //подготовка данных  
    ReadyFlag = true;  
    m.unlock();  
    //...  
}
```

```
void thread2(){  
    std::unique_lock <std::mutex > l(m);  
    while(!ReadyFlag){  
        l.unlock();  
        std::this_thread::sleep_for(100ms);  
        l.lock();  
    }  
    //обработка данных  
}
```

Проблемы:

- флаг проверяется независимо от того, произошло событие или нет
- трудно подобрать время ожидания (в среднем поток будет ждать $\text{duration}/2$):
 - слишком короткий период между проверками -> не производительно
 - слишком длинный -> ожидание даже в том случае, когда условие уже давно выполнено
- ресурсы на `unlock()/lock()`

=> это неэффективный вариант

=> использование специальных средств для решения таких задач – **conditional variables!**

Цель использования условных переменных:

- когда нужно не просто избежать гонки (mutex), а синхронизировать параллельное выполнение задач
- требуется НЕ однократная передача результата от одного потока другому (future - однократная синхронизация), а многократное получение результатов от разных потоков
- => условные переменные, которые можно использовать для многократной синхронизации при обмене данными с несколькими потоками

Реализации условных переменных

std::condition_variable обеспечивает синхронизацию только посредством мьютекса

std::condition_variable_any обеспечивает синхронизацию посредством любого «мьютексоподобного» объекта => может потребовать дополнительных расходов по памяти, производительности, ресурсов ОС

`std::condition_variable`

`std::condition_variable_any`

это примитивы синхронизации,
предназначенные для блокировки одного
потока, пока

- он не будет оповещен о наступлении
некоего события из другого потока
- или не истечет заданный таймаут
- или не произойдет ложное пробуждение
(spurious wakeup)

Отправка/получение оповещения

Поток, формирующий условие:

- **notify_one()** – сообщить одному потоку
- **notify_all()** – сообщить всем потокам

Поток, ожидающий выполнения условия:

- **wait()** – ожидание выполнения условия
- **wait_for()**, **wait_until()**

Важно!

- в расширение возможностей мьютекса условная переменная позволяет **одному** из ожидающих или **всем** ожидающим потокам продолжить выполнение
- для реализации ожидания на условной переменной все равно необходим мьютекс
- «ложное срабатывание» == возврат управления потоку не всегда означает выполнение условия => необходимо дополнительно проверить выполнение условия

Ложное (spurious) пробуждение

- когда wait завершается, без участия notify_one() или notify_all()

То есть проверка условной переменной может сработать даже если условная переменная не осуществила уведомления => дополнительная проверка!

A.Williams: “Ложные срабатывания невозможно предсказать: с точки зрения пользователя они являются совершенно случайными. Однако они часто происходят, когда библиотека потоков не может гарантировать, что ожидающий поток не пропустит уведомления. Поскольку пропущенное уведомление делает условную переменную бесполезной, библиотека потоков активизирует поток, чтобы не рисковать”

`void wait(std::unique_lock<std::mutex>& lock);`

- освобождает ассоциированный мьютекс (это позволяет другим потокам изменять защищенные данные во время ожидания)
- добавляет этот поток в список ожидающих потоков во внутреннюю структуру данных условной переменной и блокирует вызвавший `wait()` поток
- при вызове для условной переменной `notify_...()` анализируется список и пробуждается один из или все ожидающие потоки, управление возвращается функции `wait()`, в которой снова захватывается мьютекс и выполнение продолжается

```
template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock,
          Predicate pred );
```

- используется для защиты от ложных пробуждений, так как
- проверяет дополнительное условие, и если условие==false, то снова освобождает мьютекс и переводит поток в состояние ожидания
- эквивалентно:

```
while (!pred()) {  
    wait(lock);  
}
```

Важно!

- на момент вызова `wait()` мьютекс должен быть захвачен! Иначе – неопределенное поведение
- во всех **ожидающих** потоках должен использоваться один и тот же мьютекс
- начиная с C++14 `wait()` не генерирует исключений

wait_for(), wait_until()

возврат управления и блокировка мьютекса происходит:

- если вызвана notify_...()
- сработало ложное пробуждение
- истек timeout - wait_for().
- или наступил заданный момент времени - wait_until()

Причину возврата можно узнать посредством возвращаемого значения - enum class **cv_status**; (значения: timeout, no_timeout)

notify_one(), notify_all()

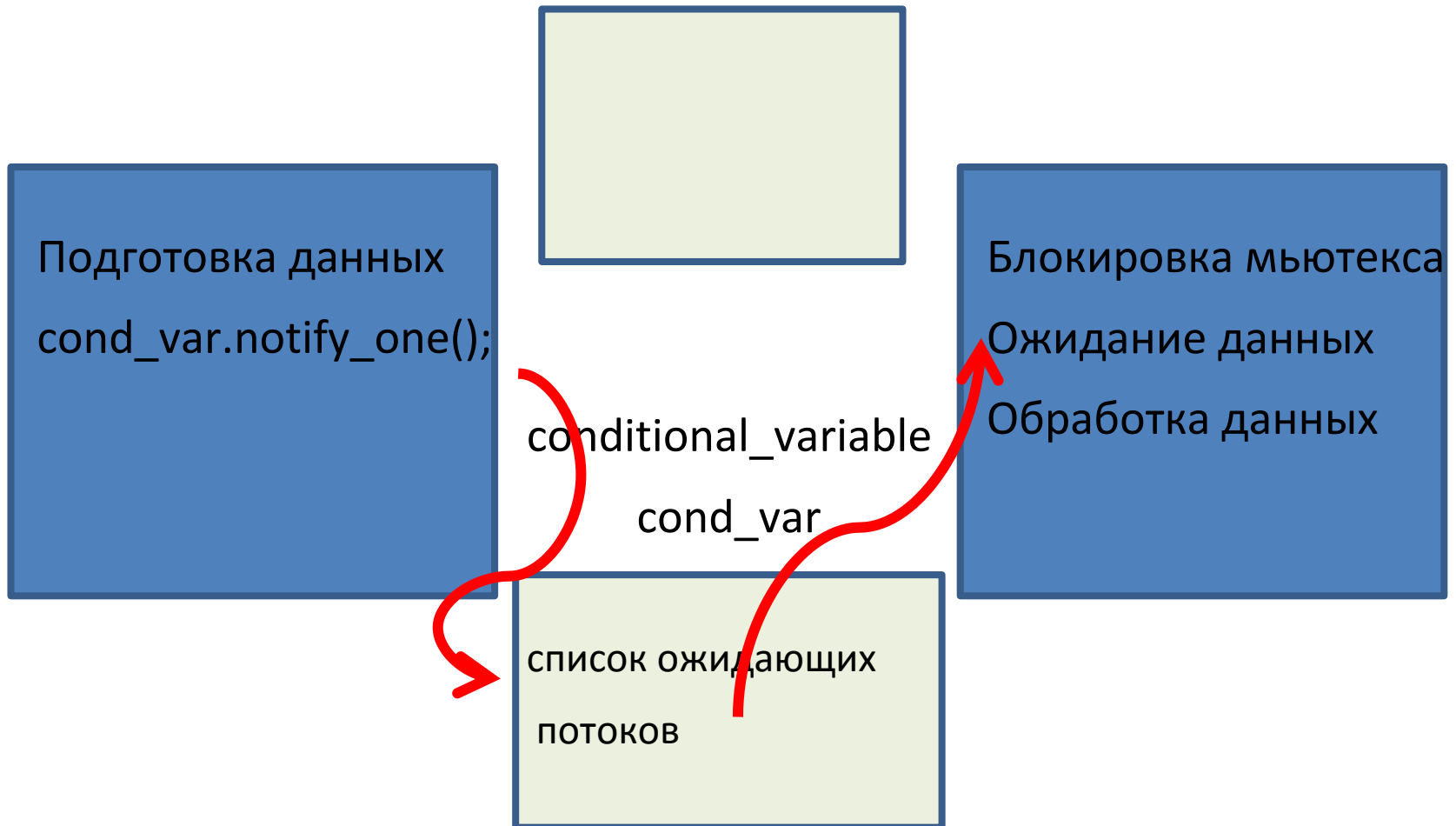
при вызове для условной переменной notify_...() анализируется список и пробуждается:

- один из ожидающих потоков
- или все ожидающие потоки

=> управление возвращается соответствующей функции wait()

Иллюстрация

Разделяемые данные



Данные «поставляет» только один поток
и обрабатывает только один поток :

```
int shared = 0;  
std::condition_variable cv;
```

```
void thread_Write() {  
    //подготовка данных  
    shared = 33;  
    //уведомление о готовности  
    cv.notify_one();  
}
```

```
void thread_Read () {  
    std::mutex local_m;  
  
    std::unique_lock <std::mutex > l(local_m);  
  
    cv.wait(l);  
  
    //обработка данных  
}
```


???

Почему нельзя использовать
lock_guard?

Данные «поставляет» только один поток и обрабатывает только один поток. Защита от ложных пробуждений

```
int shared = 0;  
std::condition_variable cv;
```

```
void thread_Write() {  
    //подготовка данных  
    shared = 33;  
    //уведомление о готовности  
    cv.notify_one();  
}
```

```
void thread_Read () {  
    std::mutex local_m;  
  
    std::unique_lock <std::mutex > l(local_m);  
  
    cv.wait(l, []{ return shared == 33; });  
  
    //обработка данных  
}
```

Данные «поставляют» несколько потоков :

```
std::mutex m;  
std::conditional_variable cv;  
int shared = 0;
```

```
void thread_Write1() {  
    //подготовка данных  
    shared = 33; //гонка!  
    //уведомление о готовности  
    cv.notify_all();  
}
```

```
void thread_Write2() {  
    //подготовка данных  
    shared = 44; //гонка!  
    //уведомление о готовности  
    cv.notify_all();
```

```
void thread_Read(){  
    std::unique_lock <std::mutex> l(m);  
    cv.wait(l, [] { return shared !=0; });  
    ...  
}
```

Уведомление с помощью условной переменной в **нескольких** потоках:

```
std::mutex m;  
std::conditional_variable cv;  
int shared = 0;
```

```
void thread_Write() {  
  std::lock_guard<std::mutex> l(m);  
  //подготовка данных  
  shared = 33;  
  //уведомление о готовности  
  cv.notify_all();  
}
```

```
void thread_Read(){  
  std::unique_lock <std::mutex> l(m);  
  cv.wait(l,  
          [] { return shared == 33; });  
  ...  
}
```

А можно так:

```
std::mutex mR,mW;  
std::conditional_variable cv;  
int shared = 0;
```

```
void thread_Write() {  
  std::lock_guard<std::mutex>  
    l(mW);  
  //подготовка данных  
  shared = 33;  
  //уведомление о готовности  
  cv.notify_all();  
}
```

```
void thread_Read(){  
  std::unique_lock <std::mutex>  
  l(mR);  
  cv.wait(l,  
    [] { return shared == 33; });  
  ...  
}
```

Глобальная функция

`std::notify_all_at_thread_exit()`

```
void notify_all_at_thread_exit(  
    std::condition_variable& cond,  
    std::unique_lock<std::mutex> lk );
```

Пробуждает все ожидающие потоки (`cond.wait()`) **при завершении** текущего потока. Эквивалентно:

```
lk.unlock();  
cond.notify_all();
```

Важно! Действия выполняются гарантированно **после** вызова деструкторов всех потоко-локальных объектов (thread local storage duration)

Пример std::notify_all_at_thread_exit

```
std::mutex m;  
std::conditional_variable cv;  
int shared = 0;
```

```
void thread_Write() {  
    std::lock_guard<std::mutex> l(m);  
  
    //использование thread_locals  
  
    //уведомление о готовности выдать только  
    после завершения потока  
    std::notify_all_at_thread_exit(cv,  
        std::move(l));  
    ...  
} //1. деструкторы для thread_locals, 2.  
unlock mutex, 3. notify_all()
```

```
void thread_Read(){  
    std::unique_lock <std::mutex>  
        l(m);  
    cv.wait(l, <проверка усл.>);  
    ...  
}
```

Пример std::condition_variable

```
std::vector<int> data;  
std::condition_variable cv;  
std::mutex m;
```

```
void thread_func1() {  
    std::lock_guard<std::mutex> lock(m);  
    data.push_back(<вычисленное_значение>);  
    cv.notify_one();  
}  
  
void thread_func2() {  
    std::unique_lock<std::mutex> lock(m);  
    cv.wait( lock, []() { return !data.empty(); } );  
    std::cout << data.back() << std::endl;  
}
```


`std::condition_variable::native_handle()`

`native_handle_type` `native_handle();`

В ОС Windows условные переменные на системном уровне поддерживаются синхронизирующим примитивом **CONDITION_VARIABLE**

Системные функции для работы с CONDITION_VARIABLE:
`InitializeConditionVariable()`,
`SleepConditionVariableSRW ()`, `WakeConditionVariable()`

```
#include <mutex>
```

```
STD::CALL_ONCE  
STD::ONCE_FLAG
```

Защита разделяемых данных во время инициализации

Для «дорогостоящих» разделяемых данных (например, получение доступа к БД или установка сетевого соединения) актуальна проблема «отложенной инициализации» (сначала проверка: была ли произведена инициализация, и выполнение инициализирующих действий только при первом вызове).

В однопоточной программе – проблем нет.

В многопоточной программе – защита!

Совсем неэффективное решение:

```
std::shared_ptr<Resource> resource_ptr;  
std::mutex resource_mutex;
```

```
void f()  
{  
    resource_mutex.lock();  
    if(! resource_ptr)  
    {  
        resource_ptr.reset(new Resource);  
    }  
    resource_mutex.unlock();  
    ...  
}
```

Решение – Double-Checked Locking

```
std::shared_ptr<Resource> resource_ptr;
```

```
std::mutex resource_mutex;
```

```
void f()
```

```
{
```

```
    if(! resource_ptr) /*1
```

```
    {
```

```
        std::lock_guard<std::mutex> lk(resource_mutex);
```

```
        if(! resource_ptr)
```

```
        {
```

```
            resource_ptr.reset(new Resource); /*2
```

```
        }
```

```
    }
```

```
...
```

```
}
```

Проблемы Double-Checked Locking

может привести к гонкам (data race), так как:

- чтение без мьютекса ***1** не синхронизировано
- с защищенной мьютексом записью в другом потоке ***2**

=> в C++11 включены

- `std::call_once()`
- `std::once_flag`

std::call_once()

- std::call_once создана для того, чтобы защищать общие данные во время инициализации
- это техника, позволяющая вызвать нечто callable один раз, независимо от количества потоков, которые пытаются выполнить этот участок кода

Специфика – variadic template:

```
template< class Callable, class... Args >  
void call_once( std::once_flag& flag,  
                Callable&& f, Args&&... args );
```

Напоминание: параметры передаются по значению. Если нужно передать ссылку, то `std::ref()` или `std::cref()`

std::once_flag

- если один и тот же объект `std::once_flag` передается при разных вызовах `std::call_once()` – функции, гарантируется, что тело указанной функции будет выполнено один раз!

Эффективность

Утверждается, что издержки использования `std::call_once()` ниже, чем явное использование мьютекса

Модификация примера с использованием `std::call_once()`

```
std::shared_ptr<Resource> resource_ptr;
```

```
std::once_flag resource_flag;
```

```
void init_resource()
```

```
{ resource_ptr.reset(new Resource); }
```

```
void f()
```

```
{  
    std::call_once(resource_flag, init_resource);
```

```
    ...
```

```
}
```

???

```
std::once_flag flag;
```

```
void do_once(char c){  
    std::call_once(flag, [c]() { std::cout << c << std::endl; });  
}
```

```
int main(){  
    std::thread t1(do_once, 'A');  
    std::thread t2(do_once, 'B');  
    std::thread t3(do_once, 'C');  
  
    t1.join();    t2.join();    t3.join();  
}
```

Пример `std::call_once()` для методов класса

```
class A {  
    std::once_flag fl;  
    void open_connection();  
public:  
    void send(<парам>){  
        std::call_once(fl, &A:: open_connection, this);  
        //отправка данных  
    }  
    <возвр_знач> receive(){  
        std::call_once(fl, &A:: open_connection, this);  
        //прием данных  
    }  
    ...  
};
```

ПРОГРАММИРОВАНИЕ НА ОСНОВЕ ЗАДАЧ

Скотт Мейерс:

«Предпочитайте
программирование на основе
задач программированию на
основе потоков»

Проблемы подхода на основе ПОТОКОВ:

- Получение формируемого потоком результата?
- Обработка сгенерированного потоком исключения?
- Количество запускаемых в системе потоков – ограниченный ресурс => при превышении – `std::system_error`
- «Превышение подписки» - количество программных потоков (активных и в состоянии готовности) значительно превышает количество аппаратных потоков => частые переключения контекста => время на переключение + перезагрузка кэшей процессора => неэффективно

Большинство перечисленных проблем

- Решается посредством распараллеливания на основе задач =>
- Переложить проблемы программиста на средства, предоставляемые для этих целей стандартной библиотекой

Когда актуально программирование
посредством ПОТОКОВ (требуется управление «вручную»):

- Доступ к нижележащим средствам ОС (например, задание приоритетов)
- Выигрыш по эффективности в задачах с фиксированным (заранее известным числом потоков => можно оптимизировать)
- Реализация пулов потоков

Высокоуровневые средства запуска и взаимодействия потоков. Асинхронное программирование. Ожидание одноразовых результатов

```
#include <future>
```

STD::ASYNC()

STD::FUTURE

STD::SHARED_FUTURE

Проблема 1: результат работы потоковой функции посредством `std::thread`?

в качестве одного (нескольких) параметра в потоковую функцию можно отправить адрес, по которому функция «положит» сформированный ею результат

Проблемы:

- при ожидании завершения запущенного потока???
- если `detach()`???

Проблема 2: если дочерний поток сгенерировал исключение

А обработка требуется в родительском потоке?

Проблема 3: ограниченное количество потоков

- Для каждого процесса
 - Для системы в целом
- => Требуется управление потоками вручную!

При исчерпании потоков – `std::system_error`

Проблема 4 – «превышение подписки»

большое количество запущенных потоков вынуждает ОС все время осуществлять переключение потоков =>

ресурсы на переключение:

- Время на сохранение/восстановление контекста
- Перегрузка кэшей

Задача:

Требуется запуск длительной операции и получение результата. При этом результат понадобится позже (а может быть и вообще не понадобится...)

Варианты:

- вызвать синхронно =>
 - ‘+’ - результат гарантированно сформирован после вызова
 - ‘-’ – время, потраченное на вычисления, можно было бы потратить на выполнение какой-нибудь другой полезной работы, не требующей результата

Проблемы получения результата **detached** потока:

- как узнать о том, что результат сформирован?
- как обеспечить существование объекта, адрес которого передан в поток?

Асинхронное программирование

стиль программирования, при котором :

- «тяжеловесные» задачи исполняются в detached дочерних (фоновых) потоках,
- и результат выполнения которых может быть получен родительским потоком, когда он того пожелает (при условии, что результат доступен)

Основа асинхронного программирования

- - это **разделяемое состояние** (зависит от конкретной реализации – обычно класс с подсчетом ссылок) ,
- доступ к которому осуществляется посредством шаблонов `std::future<>` и `std::promise<>`

Связь <future> с разделяемым состоянием

```
template<class _Ty> class future<_Ty&>  
: public _State_manager<_Ty *>  
{ ...};
```

```
template<class _Ty> class _State_manager // class for managing possibly non-existent associated asynchronous  
state object  
...  
private:  
_Associated_state<_Ty> *_Assoc_state;  
bool _Get_only_once;  
};
```

```
template<class _Ty> class _Associated_state  
{ // class for managing associated synchronous state  
_Atomic_counter_t _Refs;  
...  
};
```

шаблон функции `std::async()`

- позволяет **синхронно** или **асинхронно** запустить задачу (потенциально в отдельном потоке)
- передать ей любое количество параметров (аналогично `std::thread`)
- и получить возвращаемое потоком значение или обработать сгенерированное задачей исключение с помощью шаблона `std::future` (когда оно понадобится и гарантированно будет сформировано)

Формы `async()`:

```
template< class Function, class... Args>  
std::future<std::result_of_t<std::decay_t<Function>  
(std::decay_t<Args>...)>>  
    async( Function&& f, Args&&... args );
```

```
template< class Function, class... Args >  
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>  
    async( std::launch policy, Function&& f,  
          Args&&... args );
```

Специфика **std::launch** :

- По умолчанию **std::launch::deferred** | **std::launch::async** реализация решает сама: запускать новый поток или выполнять синхронно
- **std::launch::deferred** – отложить вызов функции до вызова методов `wait()` или `get()` объекта `future` (*lazy evaluation*)
- **std::launch::async** – запуск функции в отдельном потоке

Важно!

Использование политики

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

и `thread_local` объектов

комбинируются ПЛОХО!

Для асинхронного взаимодействия и обмена данными thread support library предоставляет:

- шаблон **future**, который предоставляет доступ к универсальному хранилищу (placeholder)
 - для значения любого типа, которое будет сформировано в будущем
 - или исключения любого типа, которое будет сгенерировано в будущем
- **futures** – средство для однократного
 - получения возвращаемых потоками значений
 - и обработки исключений при асинхронном выполнении

Будущие результаты (future)



`std::future<> & std::shared_future<>`

Что такое future?

- средство **однократной** коммуникации двух потоков посредством «разделяемого состояния» == канал, соединяющий два Потока (саморазрушается после приема данных)

Замечание:

- аналог `std::unique_ptr`, который уничтожается после «чтения»

Иллюстрация передачи/приема данных между потоками



Важно!

- Будущие результаты сами по себе не обеспечивают синхронизацию доступа к разделяемым данным
- => если несколько потоков используют один и тот же объект `future`,
 - то они сами должны синхронизировать доступ с помощью `future::valid()`, мьютекса или других средств синхронизации
 - или нужно позволить нескольким потокам пользоваться одним и тем же объектом **`std::shared_future`** или каждому потоку работать со своей копией - **`std::shared_future`** без дополнительной синхронизации

Шаблон класса `std::future`:

- обеспечивает доступ к **обертке** для значения любого типа, вычисление или получение которого происходит отложено (то есть в неизвестном будущем)
- фактически поддерживает механизм однократного уведомления, является получателем будущего значения
- сам по себе `future` пассивен, он просто предоставляет доступ к некоторому **разделяемому состоянию**, которое состоит из 2-х частей: собственно интересующие **данные** + **флаг готовности** (как только значение вычислено, устанавливается флаг)
- для многих задач менее ресурсоемко, чем `condition variable + mutex`

Шаблон `std::future`

предоставляет механизм для однократного получения результата асинхронной операции, инициированной: **`std::async()`**, **`std::packaged_task()`**, **`std::promise()`**

- готовность результата посредством методов **`std::future`**:
 - **`wait()`** - блокирует вызвавший поток до получения результата
 - **`wait_for()`** — ждет завершения или истечения timeout-а
 - **`wait_until()`** - ждет завершения или наступления момента
 - **`get()`** — для ожидания вызывает **`wait()`** и возвращает результат

`std::future::get()`

- **`T get();`** //генеральный шаблон (возвращаемое значение формируется посредством **`std::move`**) => повторный вызов == неопределенное поведение
- **`T& get();`**//для специализации `future<T&>`
- **`void get();`** // для специализации `future<void>`

Важно!

- Любой из вариантов дожидается результата
- Если было сохранено исключение, оно заново генерируется

std::future::get()

При вызове get() могут быть три ситуации:

- если выполнение функции происходило в отдельном потоке и уже закончилось (или результат уже сформирован) – сразу получаем результат
- если выполнение функции происходит в отдельном потоке и еще не закончилось (или результат еще не сформирован) – поток-получатель блокируется
- если запуск был задан синхронно, то происходит просто вызов функции + формирование future

Важно: вторичный вызов get() – неопределенное поведение!

Пример `std::async()`.

Поведение по умолчанию

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(sum, std::cref(v));
    //...какие-то вычисления
    std::cout<<f.get(); //ждемся окончания,
                        получаем результат
}
```

Пример `std::async()` Явное задание условий вызова - `std::launch::async`:

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(std::launch::async, sum,
                                   std::cref(v));

    //...какие-то вычисления

    std::cout<<f.get(); //ждемся окончания,
                        получаем результат
}
```

Пример `std::async()`. Явное задание условий вызова- `std::launch::deferred`

```
int sum(const std::vector<int>& v){  
    int res = 0;  
    for (auto i:v) { res += i;}  
    return res;  
}
```

```
int main(){  
    std::vector<int> v = { 1, 2, 3, 4 };  
    std::future<int> f = std::async( std::launch::deferred, sum,  
                           std::cref(v));  
  
    //...какие-то вычисления  
    if(<условие>) { std::cout<<f.get(); } //синхронный вызов + прием  
                                   возвращаемого значения  
}
```

std::launch::deferred и std::launch::async

```
void Consumer(std::future<int> f)
    { if(условие) {std::cout << f.get();} }
int Producer() { return rand() % 10; }
int HeavyProducer() { std::this_thread::sleep_for(1s); return 0; }

int main(){
    auto f1 = std::async(std::launch::deferred, Producer);
    std::thread th1(Consumer, std::move(f1));
    auto f2 = std::async(std::launch::async, HeavyProducer);
    std::thread th2(Consumer, std::move(f2));
    //полезная работа
    th1.join(); th2.join();
}
```

Специфика: если функция ничего не
возвращает

```
void func(void);
```

```
int main(){
```

```
    std::future<void> f = std::async( func);
```

```
    //...какие-то вычисления
```

```
    f.get(); //эквивалентно f.wait(); ???
```

```
}
```

Пример использования wait_for()

```
//Политику запуска определяет реализация!  
using namespace std::chrono_literals;  
auto f = std::async([]() {std::this_thread::sleep_for(3s);  
    std::cout << "Working hard!" << std::endl;});  
while (f.wait_for(1s) == std::future_status::timeout) {//если  
    политика запуска была deferred, то выход из цикла  
    std::cout << "in while" << std::endl; // выполняем  
        другую работу, не требующую результата  
}  
std::cout << "main" << std::endl;  
f.wait();
```

Проблемы: отложенный вызов и future::wait_for() или future::wait_until()

Специфика: вызов future::wait_for() или future::wait_until()
возвращает для отложенных задач

std::future_status::deferred =>

```
int main(){  
    using namespace std::chrono_literals;  
    auto ft = std::async(std::launch::deferred,[](){...});  
    while(ft.wait_for(100ms) !=  
           std::future_status::ready){/*вечно!*/}  
    //а сюда мы не попадем НИКОГДА!!!  
}
```


Как узнать – какая была политика
запуска?

???

Как узнать – какая была политика запуска?

```
auto f = std::async(sum,1,2);  
if (f.wait_for(0ms) != std::future_status::deferred)  
{  
    while (f.wait_for(100ms) != std::future_status::ready) {  
        //полезная работа  
    }  
}  
auto res = f.get();
```

std::future::valid()

```
int func(int);  
int main(){  
    int res=0;  
    std::future<int> f1 = std::async( func, 5);  
    bool b = f1.valid(); //true  
    std::future<int> f2 = std::move( f1);  
    b=f1.valid(); //false  
    if(f2.valid()){ res = f2.get();} //true  
    b=f2.valid(); //false  
}
```

Сохранение исключения в объекте future

Если функция, вызванная посредством `async()`, генерирует исключение,

- это исключение сохраняется в объекте `future` вместо значения (результата)
- вызов `get()` повторно возбуждает **сохраненное исключение** (при этом стандарт не оговаривает: в объекте `future` создается копия исключения или сохраняется ссылка на оригинал)

Сохранение исключения в объекте future. Пример:

```
double square_root(double x) {  
    if (x<0) { throw std::out_of_range("x<0"); }  
    return sqrt(x);  
}  
int main(){  
    std::future<double> f = std::async(square_root, -1);  
    double y=0;  
    try {  
        y = f.get();  
    }  
    catch (std::out_of_range& e){ std::cout << e.what(); }  
    //или catch (std::exception& e){ std::cout << e.what(); }  
}
```

Пример более жизненный:

```
int main() {  
    std::vector<std::future<int>> futures;  
    size_t N = <большое!!!>;  
    futures.reserve(N);  
  
    for(int i = 0; i < N; ++i) {  
        futures.push_back (std::async([](int x){return x*x;},i));  
    }  
    //... занимаемся полезной работой  
    //нужны ВСЕ результаты!  
    for(auto &f : futures) {  
        std::cout << f.get() << std::endl;  
    }  
    ...  
}
```

`std::future::~~future()`

Если деструктор вызывается для последнего «владельца» разделяемого состояния, то разделяемое состояние уничтожается

Как и где происходит «самоуничтожение разделяемого состояния»

Вариант 1.

Caller	Callee
<pre>{ auto ft = std::async(callable); //ref_count++ (2) int res=ft.get(); //результат «перемещается» из shared_state в res, ref_count– (0) => последний владелец => уничтожение! } //~ft – «пустой»</pre>	<pre>//создается анонимный функтор, в который запаковываются callable + параметры для callable + указатель на shared_state => ref_count=1 int callable() //ref_count=1 { ... return 1; //установка флага готовности } ~анонимный функтор => ref_count– (1)</pre>

Замечание:

- Если деструктор `future` вызывается до вызова `get()` или `wait()`, деструктор блокирует поток до завершения задачи

Как и где происходит «самоуничтожение разделяемого состояния»

Вариант 2.

Caller	Callee
<pre>{ /*auto ft*/??? = std::async(callable); //ref_count++ (2) }//~ft – Callee еще выполняется! ref_count– (1) ref_count!=0 => Caller блокируется, пока ref_count не станет =0</pre>	<pre>//создается анонимный функтор, в который запаковываются callable + параметры для callable + указатель на shared_state => ref_count=1 int callable() //ref_count=1 { ... return 1; //установка флага готовности }~анонимный функтор => ref_count– (0); => последний владелец => уничтожение!</pre>

Пример:

```
double sumN(int n){  
    double s = 0;  
    for (int i = n; i > 0; i--) { s += i; }  
    std::cout << "done";  
    return s;  
}
```

```
int main(){  
    {  
        std::future<double> f = std::async(sumN, 0x7fffffff);  
        } //~f заблокирует родительский поток до завершения  
        дочернего  
    }
```

STD::SHARED_FUTURE

Если требуется в нескольких потоках ожидать
получения одного и того же результата ???

```
int threadFunc(int x, int y){ return x + y; }
```

```
void thread1(std::future<int>& f){ int res = f.get(); }
```

```
void thread2(std::future<int>& f){ int res = f.get(); }
```

```
int main(){
```

```
    std::future<int> f = std::async(threadFunc, 1, 2);
```

```
    std::thread t1(thread1, std::ref(f));
```

```
    std::thread t2(thread2, std::ref(f));
```

```
    t1.join();  t2.join();
```

```
}
```

Шаблон класса `std::shared_future`

- позволяет получить результат асинхронной операции, но,
 - в отличие от `future` (вторичный вызов `get()` – неопределенное поведение)
 - **`shared_future`** допускает несколько вызовов `get()` и возвращает один и тот же результат или генерирует одно и то же исключение
- создать и проинициализировать можно:
 - посредством конструкторов `shared_future`
 - или методом `future::share()`

Аналогия:

- `std::future` – аналог `unique_ptr`
- `std::shared_future` – аналог `shared_ptr` =>
Реализован как класс с подсчетом ссылок =>
первый пользователь создает, последний
уничтожает

Формирование `std::shared_future` посредством `std::future`

```
std::future<int> f1 = ...;  
std::shared_future<int> sf1 ( std::move(f1));  
bool b1 = f1.valid(); //???
```

```
std::future<int> f2 = ...;  
std::shared_future<int> sf2= f2.share();  
bool b2 = f2.valid(); //???
```


Объекты `shared_future` можно:

- копировать
- присваивать
- перемещать

Модифицируем пример посредством `std::shared_future`

```
int threadFunc(int x, int y){ return x + y; }
```

```
void thread1(std::shared_future<int>& f){ int res = f.get(); ...}
```

```
void thread2(std::shared_future<int>& f){ int res = f.get(); ...}
```

```
int main(){
```

```
    std::future<int> f = std::async(threadFunc, 1, 2);
```

```
    std::shared_future<int> sf(std::move(f));
```

```
    std::thread t1(thread1, sf);
```

```
    std::thread t2(thread2, sf);
```

```
    t1.join();  t2.join();
```

```
}
```

Или по значению:

```
int threadFunc(int x, int y){ return x + y; }
```

```
void thread1(std::shared_future<int> f){ int res = f.get(); ...}
```

```
void thread2(std::shared_future<int> f){ int res = f.get(); ...}
```

```
int main(){
```

```
    std::future<int> f = std::async(threadFunc, 1, 2);
```

```
    std::shared_future<int> sf(std::move(f));
```

```
    std::thread t1(thread1, sf);
```

```
    std::thread t2(thread2, sf);
```

```
    t1.join();  t2.join();
```

```
}
```

```
#include <future>
```

```
STD::PACKAGED_TASK
```

Зачем нужен `packaged_task<>`

Проблема: есть обычная callable, которую хочется выполнить в отдельном потоке и получить **результат!**

Варианты:

1. Изменить сигнатуру функции, чтобы она принимала параметр типа `promise` — не хочется
2. Изменить сигнатуру функции, чтобы она принимала адрес, по которому сформирует результат — тоже не хочется
3. «Завернуть» callable в обертку `packaged_task`

std::packaged_task

- реализует понятие «задачи» - является хранилищем для пары callable + future => удобно использовать при асинхронном программировании
- расширяет возможности async()
- используется также, как объект function<> (перегружен operator() – возвращаемое значение или исключение запаковывается в «разделяемое состояние» => доступны посредством future<>)

Пояснение:

При создании объекта типа `packaged_task`:

```
std::packaged_task<int(int, int)> task1(sum);
```

компилятор генерирует анонимный функтор типа:

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

Специфика `std::packaged_task`

- исключительно перемещаемый класс
- перегружен `operator()` таким образом, что он:
 - вызывает callable с сохраненными в объекте `packaged_task` значениями в качестве параметров
 - сохраняет возвращаемое значение или исключение в разделяемом состоянии и устанавливает в разделяемом состоянии признак готовности
- попытка повторного использования объекта `packaged_task` – исключение `std::future_error`

Попытка повторного использования объекта `packaged_task`:

```
std::packaged_task<double(int)> task1(sumN);  
task1(1000);  
try{  
    task1(2000);  
}  
catch (std::future_error){ ... }
```

Для получения future из packaged_task

```
std::future<R>
```

```
std::packaged_task::get_future();
```

Важно! Можно вызвать только один раз, так как происходит **перемещение**, а не копирование ассоциированного с packaged_task объекта future (то есть счетчик ссылок в разделяемом состоянии не инкрементируется)

Разница в использовании `packaged_task`:

```
int sum(int x, int y) { return x + y; }
```

```
int main(){
```

```
    std::packaged_task<int(int, int)> task(sum); //это просто  
                                                подготовка к запуску
```

```
    std::future<int> f = task.get_future(); //получаем доступ к  
                                           разделяемому состоянию
```

```
task(1, 2); //!!! но! вызов функции в ТЕКУЩЕМ потоке!!!
```

```
//делаем что-то полезное
```

```
    int res = f.get(); //получаем результат
```

```
}
```

Разница в использовании `packaged_task`:

```
int sum(int x, int y) { return x + y; }
```

```
int main(){
```

```
    std::packaged_task<int(int, int)> task(sum); //это просто  
                                                подготовка к запуску
```

```
    std::future<int> f = task.get_future(); //получаем доступ к  
                                           разделяемому состоянию
```

```
    std::thread th(std::move(task), 1, 2); //!!! запуск потока!!!  
    th.detach();
```

```
    //делаем что-то полезное
```

```
    int res = f.get(); //получаем результат
```

```
}
```

Важно! «Заготовить» future необходимо до move(task) !!!

```
std::packaged_task<int(int, int)> task(sum);  
bool b1 = task.valid(); //???  
std::thread th(std::move(task), 1, 2); // запуск потока  
th/  
bool b2 = task.valid(); //???
```

//task стал «недействительным»! Его разделяемое состояние и callable перемещены в другой поток

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

//до «полезного» дело не дойдет, так как будет сгенерировано исключение `std::future_error`

Специфика future и packaged_task:

- деструктор future, полученного из packaged_task НЕ блокирует родительский поток до завершения дочернего

Специфика `packaged_task`

так как перегружен `operator()`, объект `std::packaged_task` в свою очередь можно использовать везде, где требуется callable:

- в частности «завернуть» в `std::function`
- объекты такого типа можно хранить в контейнерах

Храним задачи в контейнере:

```
double sumN(int n) { double s = 0; for (int i = n; i > 0; i--) {s += i;} return s; }
int main(){
    std::vector<std::packaged_task<double(int)> > tasks;
    tasks.reserve(10);
    std::vector<std::future<double>> futures;
    reserve

    for (int i = 0; i < 10; i++) {
        tasks.emplace_back(sumN);
        futures.push_back(tasks[i].get_future());
    }

    int n = 10;
    for(auto& t:tasks){ std::thread th(std::move(t),n); th.detach(); n *= 2; }
    for (auto& f : futures) { std::cout<< f.get()<<" "; }
}
```


Тип callable в шаблоне packaged_task и неявное приведение типа

```
double sumN(int n);
```

```
int main(){  
    std::packaged_task<double(int)> task1(sumN); //OK  
    std::packaged_task<int(int)> task2(sumN); //OK  
    std::packaged_task<double(double)>  
        task3(sumN); //OK  
    ...  
}
```

Пример packaged_task:

```
double accum(const std::vector<double>& v) {  
    typedef decltype(v.begin()) IT;  
    std::packaged_task<double(IT, IT, double)> pt1{ std::accumulate<IT,double> };  
    std::packaged_task<double(IT, IT, double)> pt2{ std::accumulate<IT,double> };  
  
    auto f0 = pt1.get_future();  
    auto f1 = pt2.get_future();  
  
    std::thread t1(std::move(pt1),v.begin(), v.begin() + v.size() / 2, 0); // запускаем  
ПОТОКИ  
    std::thread t2(std::move(pt2), v.begin() + v.size() / 2,v.end(), 0);  
    t1.detach(); t2.detach();  
    //Полезная работа  
    return f0.get() + f1.get(); // получаем результаты  
}
```

Пример межпоточного использования `packaged_task`

```
int f(){ return 1;}
```

```
std::packaged_task<int()> task(f);
```

```
void thread_func1(){ task();}
```

```
void thread_func2(){  
    std::cout << task.get_future().get() << std::endl;  
}
```

```
int main(){  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
    th1.join(); th2.join();  
}
```

```
void std::packaged_task::  
make_ready_at_thread_exit( ArgTypes... args );
```

- принимает параметры для вызова *packaged_task*.
- выполняет код функции, сохраненной в *packaged_task*, результат (возвращаемое значение или исключение) сохраняется в разделяемом состоянии (future), НО флаг готовности будет выставлен только после того, как все деструкторы локальных, для данного потока, объектов будут выполнены (т.е. прямо перед завершением потока)

#include <future>

STD::PROMISE
PROMISES, PROMISES...

Специфика `std::promise`:

- `std::promise/std::future` обеспечивает механизм (один из возможных), позволяющий передавать значения между потоками
- в отличие от `async()`, которая явно возвращает объект `future`, `promise` инкапсулирует `future`
- механизм формирования результата разный:
 - при вызове `async()` в объект `future` **неявно** запаковывается возвращаемое потоком значение или сгенерированное исключение
 - при использовании `promise` – значение нужно сформировать явно (`set_value()`, `set_exception()`)

std::promise<> и разделяемое состояние:

```
template<class _Ty> class promise{// class that defines an asynchronous provider that holds a value
...
private:
    _Promise<_Ty> _MyPromise;
}
template<class _Ty>
class _Promise
{
    // class that implements core of promise;
    ...
private:
    _State_manager<_Ty> _State;
    bool _Future_retrieved;
};
```

Связь promise и future

- метод `get_future()` возвращает объект `future`, ассоциированный с результатом

Формирование/получение результата

- **std::promise** отвечает за установку значения:
 - **std::promise::set_value()** – сохраняет значение в разделяемом состоянии и выставляет флаг готовности
 - **std::promise::set_value_at_thread_exit()** – «-», но не выставляет флаг до завершения потока
 - **std::promise::set_exception()**
- а **std::future** - за его получение:
 - **std::future::wait()**
 - **std::future::get()**

Пример promise

```
int main(){  
    std::promise<int> p;  
    std::future<int> f = p.get_future(); //можем сразу же  
                                     получить доступ к разделяемому состоянию  
    std::thread th( [&p]() {p.set_value(33);} );  
    th.detach(); //можем подождать завершения – join()  
    //делаем что-то полезное  
    int res = f.get(); //получаем результат  
}
```

Важно!

Если для одного и того же объекта `promise` повторно вызвать `set_value()` или `set_exception()`,
будет сгенерировано исключение!

Пример некорректного использования promise:

```
void fPromise(std::promise<int>& p){
    p.set_value(1);
    //...
    p.set_value(2); //или p.set_exception(std::exception_ptr());
}
int main(){
    std::promise<int> Promise1;
    std::future<int> fut1 = Promise1.get_future();
    std::thread th(fPromise, std::ref(Promise1));
    th.detach();
    std::this_thread::sleep_for(1s);
    int res1 = fut1.get();
}
```

Еще один пример межпоточного обмена данными посредством promise

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

```
void thread_func1(){ promise.set_value(33); }
```

```
void thread_func2(){ std::cout << promise.get_future().get(); }
```

```
int main(){  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
  
    th1.join(); th2.join();  
}
```

Пример передачи исключения посредством promise

```
void sum_promise(const std::vector<int>& v,  
                std::promise<int>& p)  
{  
    try {  
        if (v.empty()) { throw std::runtime_error("empty"); }  
        int res = 0;  
        for (const auto& i : v){ res += i; }  
        p.set_value(res);  
    }  
    catch (...) { p.set_exception(std::current_exception()); }  
}
```

Продолжение примера

```
int main(){  
try {  
    std::vector<int> v = { 1,2,3,4,5,6,7 };  
    std::promise<int> p;  
    std::thread th(sum_promise, std::cref(v),  
                   std::ref(p));  
  
    th.detach(); //!!!  
    //...  
    int sum = p.get_future().get();  
}catch (std::exception& e){...}  
}
```

Важно:

- `f.get()`; блокирует вызвавший поток **до формирования результата** (когда в другом потоке будет вызвана `p.set_value()`; или `p.set_exception()`), а не до завершения поставляющего результат потока!!!
- для ожидания завершения – `p.set_value_at_thread_exit(res);`

Создание потока в приостановленном состоянии

Важно! В тех ситуациях когда, перед тем, как поток начнет выполнение (вернее будет включен в диспетчирование) нужно произвести его «настройку», например:

- установить приоритет (системными средствами)
- получить для других системных действий дескриптор потока

Для этого можно использовать специализацию **`std::promise<void>`**

Пример создания потока в приостановленном состоянии

```
void SimpleThreadF() { std::cout << "Child" << std::endl; }  
int main(){  
    std::promise<void> suspendPromise;  
    std::thread th([&suspendPromise] {  
        suspendPromise.get_future().wait(); SimpleThreadF();});  
  
    // "Настройка" потока:  
    HANDLE h = th.native_handle();  
    ::SetThreadPriority(h, THREAD_PRIORITY_ABOVE_NORMAL);  
    std::cout << "Parent" << std::endl;  
    // Запуск:  
    suspendPromise.set_value();  
    // Возможно еще какая-то полезная работа  
    th.join();  
}
```

Как получить несколько результатов?

```
void fMultiplePromises(std::promise<int>& p1, std::promise<int>& p2,  
std::promise<int>& p3){  
    p1.set_value(1);  
    p2.set_value(2);  
    p3.set_value(3);  
}  
  
int main(){  
    std::promise<int> Promise1, Promise2, Promise3;  
    std::future<int> fut1 = Promise1.get_future(), fut2 = Promise2.get_future(),  
        fut3 = Promise3.get_future();  
    std::thread th(fMultiplePromises, std::ref(Promise1), std::ref(Promise2),  
        std::ref(Promise3));  
  
    th.detach();  
    int res1 = fut1.get();  
    int res2 = fut2.get();  
    int res3 = fut3.get();  
}
```

Передача исключения посредством promise

- Для передачи исключения - метод `std::promise::set_exception` (который принимает объект типа `std::exception_ptr`)
- Напоминание: получить объект `std::exception_ptr` можно:
 - вызовом `std::current_exception()` из блока `catch`
 - либо создать объект этого типа напрямую посредством `std::make_exception()`

Пример обработки исключения

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

```
void thread_func1(){  
    promise.set_exception(std::make_exception(std::runtime_error("error"))); }  
}
```

```
void thread_func2(){  
    try {  
        std::cout << promise.get_future().get() << std::endl;  
    } catch (const std::exception& e) { std::cout << e.what() << std::endl; }  
}
```

```
int main(){  
    std::thread th1(thread_func1);  
    std::thread th2(thread_func2);  
    th1.join(); th2.join();  
}
```

СРАВНЕНИЕ НИЗКОУРОВНЕВЫХ И ВЫСОКОУРОВНЕВЫХ СРЕДСТВ СОЗДАНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Когда и что удобнее использовать: `std::thread`

`std::thread` — программирование на уровне потоков:

- «+»
 - для получения доступа к системным средствам
 - для организации пула потоков
 - для задания thread local - данных
- «-»
 - трудно/небезопасно получать результат работы потока
 - только два варианта взаимодействия

condition_variable + mutex

- Для многократного уведомления
- в задачах, где есть субъект, гарантированно/постоянно ожидающий наступления некоторого события

Когда и что удобнее использовать: `std::async` и `std::future`

`std::async` — программирование на уровне задач (однократное получение результата):

- «+»
 - взаимодействие посредством разделяемого состояния (данные + флаг готовности)
 - возможность синхронного/асинхронного запуска
- «-»
 - по умолчанию неизвестно, как (синхронно/асинхронно) выполняется задача (=> проблемы `thread_local` переменных)
 - если запуск асинхронный, стартует сразу!
 - `std::future::get()` (`wait()`) блокирует родительский поток до завершения дочернего (`return <значение>` или `throw <исключение>`)
 - `~future()` может блокировать родительский поток

Когда и что удобнее использовать: `std::shared_future`

для многократного получения одного и того же результата (в разных потоках)

Когда и что удобнее использовать: `std::promise`

`std::promise` — хранилище для результата (который можно получить посредством `std::future`):

- позволяет подготовить/настроить пару callable + future
- явная установка результата (или генерация исключения) — `set_value()` / `set_exception()`
- родительский поток блокируется не до завершения дочернего, а до формирования результата

Когда и что удобнее использовать: `std::packaged_task`

`std::packaged_task` – похож на `function<>`,
«упаковывает» callable + future

- можно «вызвать» напрямую (перегружен `operator()`) в текущем потоке
- можно в свою очередь «завернуть» в объект `function`
- можно передать в качестве параметра в потоковую функцию (и в любую другую функцию)
- объединить в коллекцию

Пример:

```
std::future<bool> SomeFunc(const std::string& str)
{
    auto callable = [](const std::string& s){return s<"abc";};
    std::packaged_task<bool(const std::string&)>
        task(callable);

    auto fut = task.get_future();
    std::thread thread(std::move(task), str);
    thread.detach();
    return std::move(future);
}
```