

# **Стандарт C++11 – C++14 – C++17 multithreading**

Полубенцева Марина Игоревна

```
#include <chrono>  
namespace std::chrono
```

# **CHRONO**

## **ЗАДАНИЕ ИНТЕРВАЛОВ И МОМЕНТОВ ВРЕМЕНИ**

# До C++11

## #include <ctime>

Типы:

- clock\_t
- time\_t
- tm

Функции:

- clock()
- time()
- localtime()
- ctime()
- mktime()
- ...

# Библиотека <chrono> предоставляет:

- Три основных сущности:
  - Часы
  - моменты времени (time points)
  - Интервалы времени (duration)
- Функции для получения и задания моментов и интервалов (для указанных часов)

# #include <chrono>

## C++11 предоставляет три вида часов:

- ***system\_clock*** - системное время в используемой ОС. Предоставляет функции для преобразования моментов времени в `time_t` и обратно
- ***high\_resolution\_clock*** - максимально возможное **на данной** системе разрешение (то есть наименьший тактовый период)
- ***steady\_clock*** - «стабильные часы», так как гарантированно не допускают коррекции времени системой, то есть перевода стрелок

# Замечание:

в зависимости от конкретной реализации  
high\_resolution\_clock могут быть реализованы  
посредством системных часов

# Функции использующие интервалы и моменты времени:

`std::this_thread::sleep_for()`  
`std::this_thread::sleep_until()`  
`std::conditional_variable::wait_for()`  
`std::conditional_variable::wait_until()`  
`std::timed_mutex::try_lock_for()`  
`std::timed_mutex::try_lock_until ()`  
`std::recursive_timed_mutex::try_lock_for()`  
`std::recursive_timed_mutex::try_lock_until ()`  
`std::unique_lock:: try_lock_for ()`  
`std::unique_lock:: try_lock_until ()`  
`std::future::wait_for()`  
`std::future::wait_until()`  
`std::shared_future::wait_for()`  
`std::shared_future::wait_until()`

# Замечание:

Для перечисленных функций интервалы и моменты времени задаются посредством стабильных часов!



# Информация, предоставляемая chrono:

- текущее время – статический метод **now()**
- тип, для хранения времени – **time\_point** (для часов конкретного типа)
- величина такта часов – **Period** (задается в любом, подходящем для задачи виде)
- тип для хранения интервалов времени - **duration**
- признак равномерного хода времени – не допускают коррекции времени системой, то есть перевода стрелок (такие часы называются стабильными) – статическая константа **is\_steady**

# Пример использования now()

```
std::chrono::time_point<std::chrono::system_clock>  
    start, end;
```

```
//или typedef - std::chrono::system_clock::time_point
```

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

```
//измеряемый фрагмент
```

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

# Реализация концепции нейтральной точности `std::chrono`

- отделение понятия интервала (**duration**) и момента времени (**timepoint**) от конкретных часов:

- шаблон **duration** – количество тактов заданной продолжительности/дискретности (в любых единицах, в частности в долях)
- шаблон **timepoint** – момент времени, измеренный для конкретных часов. Комбинация начала отсчета (эпохи) и интервала

# Реализация интервалов и моментов времени:

типы **duration** и **time\_point** используют пакет работы с рациональными числами времени компиляции, чтобы:

- иметь возможность выразить время в любых градациях (будь то век или пикосекунда),
- избежать неоднозначностей в единицах измерения,
- а так же ошибок при округлении

# Интервалы времени

## `std::chrono::duration`

```
template<
    class Rep, //тип представления (int, double...)
    class Period = std::ratio<1,1> //длительность одного тика –
                                   дробь – <число секунд>/<число тиков>
> class duration;
```

Замечания:

- Period – количество секунд от одного тика до другого
- константа, известная компилятору на этапе компиляции!

```
#include <ratio>
```

```
template<  
    std::intmax_t Num,  
    std::intmax_t Denom = 1  
> class ratio;
```

Числитель и знаменатель, заданные  
константами на этапе компиляции

# Примеры использования `std::ratio`:

- при `std::ratio<1,25>`  
часы «тикают» 25 раз в секунду
- при `std::ratio<5,2>`  
часы «тикают» один раз в 2.5 секунды

# Пример использования duration:

```
std::chrono::time_point<std::chrono::steady_clock>  
    start, end;
```

```
start = std::chrono::steady_clock::now();
```

```
//измеряемый фрагмент
```

```
end = std::chrono::steady_clock::now();
```

```
std::chrono::duration<double> elapsed_seconds =  
    end-start;
```



# typedefs:

type	Representation	Period
<a href="#"><u>hours</u></a>	<i>signed integral type</i> of at least 23 bits	<a href="#"><u>ratio</u></a> <3600,1>
<a href="#"><u>minutes</u></a>	<i>signed integral type</i> of at least 29 bits	<a href="#"><u>ratio</u></a> <60,1>
<a href="#"><u>seconds</u></a>	<i>signed integral type</i> of at least 35 bits	<a href="#"><u>ratio</u></a> <1,1>
<a href="#"><u>milliseconds</u></a>	<i>signed integral type</i> of at least 45 bits	<a href="#"><u>ratio</u></a> <1,1000>
<a href="#"><u>microseconds</u></a>	<i>signed integral type</i> of at least 55 bits	<a href="#"><u>ratio</u></a> <1,1000000>
<a href="#"><u>nanoseconds</u></a>	<i>signed integral type</i> of at least 64 bits	<a href="#"><u>ratio</u></a> <1,1000000000>

# Задание интервалов времени

## Примеры:

```
std::chrono::duration<int> sec(10); //10 сек по умолчанию 1/1
```

```
std::chrono::duration<double, std::ratio<60>> min(0.5);  
//0.5 мин 60/1
```

```
std::chrono::duration<long, std::ratio<1,1000>> ms(2);  
//2 мс 1/1000
```

```
int n = 1000;
```

```
std::chrono::duration<long, std::ratio<1, n>> msn(2); //???
```

//Псевдонимы

```
std::chrono::seconds sec(10);
```

```
std::chrono::hours h(5);
```

```
std::chrono::milliseconds ms(2);
```

# Замечание:

- между типами `duration` существует **НЕЯВНОЕ** преобразование, если точность при этом повышается:
  - то есть часы в секунды можно
  - секунды в часы нельзя
- для явного преобразования используется `std::chrono::duration_cast<...>(...)`  
при явном преобразовании время не округляется, а «отсекается»

# duration::count()

## КОЛИЧЕСТВО ТИКОВ В ИНТЕРВАЛЕ

```
std::chrono::milliseconds ms{3};
```

```
std::cout << ms.count(); //3 (тика)
```

```
std::chrono::duration<double, std::ratio<1, 30>> hz30(3.5);
```

```
std::cout << hz30.count(); //3.5 (тика)
```

# Примеры преобразования duration:

```
std::chrono::milliseconds ms(60800);
```

```
//std::chrono::seconds s1 = ms; //ошибка – нет  
    подходящего преобразования
```

```
std::chrono::seconds s2 =  
std::chrono::duration_cast<std::chrono::seconds>(ms);  
    //60 сек
```

# Продолжение примера измерения длительности выполняемого фрагмента

```
std::chrono::time_point<std::chrono::steady_clock> start, end;  
start = std::chrono::steady_clock::now();  
        //измеряемый фрагмент  
end = std::chrono::steady_clock::now();  
  
std::chrono::duration<double/*,std::ratio<1,1>*/> elapsed =  
        end - start;  
  
double t_mks =  
std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();  
double t_ms =  
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();  
double t_s = elapsed.seconds.count();
```

# C++14 – стандартные суффиксы для chrono-литералов

- «**h**», «**min**», «**s**», «**ms**», «**us**» и «**ns**» для создания соответствующих временных интервалов `std::chrono::duration`

```
using namespace std::chrono_literals;  
std::chrono::seconds s1 = 10s;  
auto s2 = 20s;
```

# Действия с интервалами

$-, +, ==, <, \% \dots$

```
std::chrono::seconds s(10);
```

```
std::chrono::hours h(5);
```

```
std::chrono::milliseconds ms(2);
```

```
std::chrono::milliseconds msres = ms + s;
```

```
//Ho!
```

```
//std::chrono::seconds sres = ms + s;
```



# Пример представления интервала в привычном виде:

```
using namespace std::chrono;
milliseconds ms(7255042);
hours h = duration_cast<hours>(ms);
minutes m =
    duration_cast<minutes>(ms%hours(1));
seconds s =
    duration_cast<seconds>(ms%minutes(1));
milliseconds mss =
    duration_cast<milliseconds>(ms%seconds(1));
```

# Моменты времени

## `std::chrono::time_point`

```
template<
    class Clock, // используемые часы
    class Duration = typename Clock::duration
                // единица измерения
> class time_point;
```

- промежуток времени (в указанных единицах) с некоторой точки на временной ОСИ — ЭПОХИ (обычно 01.01.1970 )

# Эпоха:

- в стандарте это понятие не определено и не регламентировано
- напрямую запросить это значение невозможно (чему равна эпоха)
- но! можно получить время между указанным моментом `time_point` и началом эпохи – **`time_since_epoch()`**

# Действия с моментами времени:

при условии, что они относятся к одним и тем же часам!

- можно вычитать
- к моменту времени можно прибавить/вычесть интервал
- $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$

# Преобразования

**std::chrono::time\_point <-> std::time\_t**

```
static std::time_t to_time_t( const time_point& t );  
static std::chrono::system_clock::time_point from_time_t(  
    std::time_t t );
```

//Получение текущего момента времени

```
{  
    std::chrono::system_clock::time_point tp =  
        std::chrono::system_clock::now();  
    std::time_t t =  
        std::chrono::system_clock::to_time_t(tp);  
    std::string s = std::ctime(&t);  
}
```

# Формирование момента времени

```
//static std::chrono::time_point from_time_t(std::time_t t );
```

```
std::tm tTm = { 0 };
```

```
tTm.tm_year = 2017 - 1900;
```

```
tTm.tm_mon = 5;
```

```
tTm.tm_mday = 12;
```

```
tTm.tm_hour = 12;
```

```
tTm.tm_min = 0;
```

```
time_t t = std::mktime(&tTm);
```

```
std::chrono::system_clock::time_point tp =  
    std::chrono::system_clock::from_time_t(t);
```

# Измерение времени

```
auto start = std::chrono::high_resolution_clock::now();
```

```
//в секундах
```

```
//измеряемый фрагмент
```

```
auto stop = std::chrono::high_resolution_clock::now();
```

```
//а результат интересует в мс
```

```
std::chrono::milliseconds t=
```

```
std::chrono::duration_cast<std::chrono::milliseconds>  
    (stop - start);
```

# Интервалы и моменты времени используются:

- **this\_thread::sleep\_for()** и **this\_thread::sleep\_until()** – для блокировки текущего потока
- **try\_lock\_for()** и **try\_lock\_until()** – для задания timeout-а для мьютекса
- **wait\_for()** и **wait\_until()** – для задания timeout-а для переменной условия или объекта future

Замечание: для функций \*\_until – если до наступления момента время будет скорректировано, то изменения будут учтены



# Важно!

Если используемая ОС не является ОСРВ, то заданный интервал или момент времени гарантируют только «не раньше, чем»!

# **РЕАЛИЗАЦИЯ МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ**

# Кроссплатформенные многопоточные библиотеки

- Набор библиотек Boost
- OpenMP
- OpenThreads
- POCO Thread (часть проекта POCO — <http://pocoproject.org/poco/info/index.html> )
- Zthread
- Pthreads
- Qt Threads
- Intel Threading Building Blocks
- **Стандартная библиотека C++ (C++11-14-17)**

# Причины ограниченного набора возможностей C++11 для создания параллельных программ:

- Необходимость обеспечить стандартное поведение на всех платформах =>
- Ограничения для разработчиков компиляторов

# **ПРОПИСНЫЕ ИСТИНЫ (КОРОТКО)**

Цель у разработчиков схемотехники и у программистов  
общая - **повышение производительности**

- задача разработчика схемотехники —  
предоставить средства,  
**задача программиста** — наиболее  
эффективно эти средства задействовать
- современный компьютер — это  
многопроцессорное оборудование  
=> **распараллеливание** обработки данных  
=> распараллеливание приводит к  
возникновению общих (**shared**) данных,  
доступ к которым программист должен  
**синхронизировать**

# Средства синхронизации, предоставляемые ОС, и параллельность

- ОС не предоставляет **средств построения параллельных структур данных**, обеспечивающих совместный доступ
- вместо этого ОС предоставляет средства **ограничения** доступа к данным в виде примитивов синхронизации
- => в некотором смысле синхронизация – это **антипод** параллельности

# Синхронизация –антипод параллельности

- распараллеливая алгоритмы, мы работаем с последовательными структурами данных, регулируя доступ к данным примитивами синхронизации – критическими секциями, мьютексами (mutex), условными переменными (condvar), ...
- в результате мы выстраиваем все наши потоки в очередь на доступ к структуре данных,
- тем самым убивая параллельность



# Важно!

- При возрастании числа потоков синхронизированный доступ к данным становится узким местом программы;
- => при увеличении степени параллельности вместо пропорционального увеличения производительности можно получить её ухудшение в условиях большой нагрузки (high contention)

# Стандарт C++11

- добавляет в стандартную библиотеку thread library:
  - поддержку параллельного выполнения (потоки)
  - средства блокирующей синхронизации потоков в рамках одного приложения
  - средства «неблокирующей» синхронизации: атомарные операции и атомарные типы
  - средства для межпоточной обработки исключений
  - ...
- но! **не** поддерживает коммуникацию между разными процессами

# Параллелизм может быть реализован:

- Аппаратно. На компьютере может быть:
  - несколько процессоров
  - многоядерный процессор
  - комбинация двух предыдущих вариантов
- Программно:
  - «псевдо-параллельно» - переключение потоков ОС
- Класс thread – это «обертка» для системных потоков

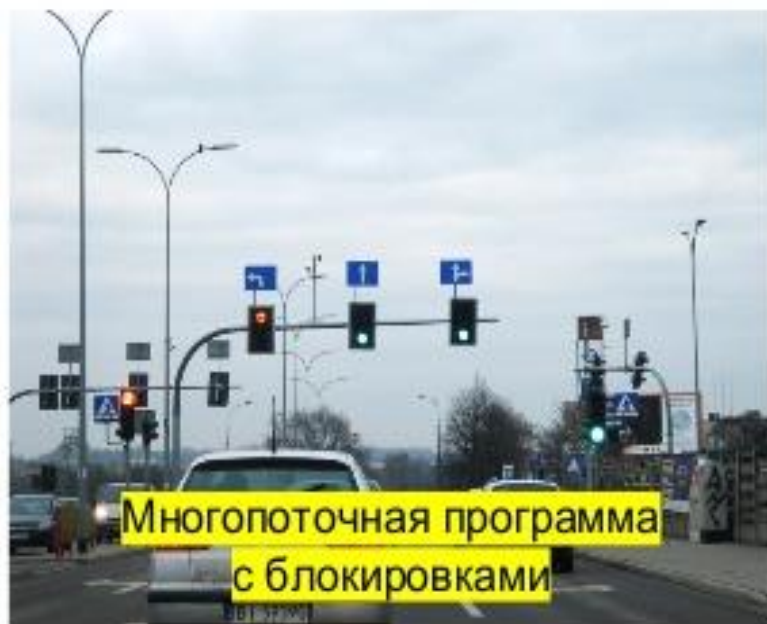
Средства C++11 применимы **независимо** от аппаратной поддержки, но всегда нужно оценивать cost/benefit от использования параллелизма! => прикладной программист должен уметь корректно и эффективно проектировать параллельный код

# Основные способы распараллеливание задач:

- Посредством **процессов** –  
«+»/«-» ???
- Посредством **потоков** –  
«+»/«-» ???



Однопоточная программа



Многопоточная программа  
с блокировками



Многопоточная программа  
без блокировок

# РЕАЛЬНЫЙ ПАРАЛЛЕЛИЗМ

# Важно!

Для любого использования параллелизма  
(реального или псевдо) программисту необходимо на  
этапе проектирования программы заложить в  
структуру программы возможность  
распараллеливания!

# Повышение производительности программ на многоядерных системах

достигается за счет:

- распараллеливания выполнения посредством потоков
- векторизации данных
- учета аппаратных особенностей



# Предупреждение – cost/benefit:

- структура параллельных программ сложнее  
(=> дополнительное время на проектирование, разработку, отладку... + ошибки)
- накладные расходы на запуск и переключение потоков
- ограниченное количество вычислительных ядер в системе
- ограниченные ресурсы процесса (для каждого потока формируется свой стек => память + ОС может накладывать ограничения на количество запускаемых процессом потоков)

# **STD::THREAD**

# Наступила эпоха «стандартной» параллельности!

```
void threadHello()  
{ std::cout<<"Hello, parallel world!";}
```

```
int main()  
{  
    std::thread th(threadHello);  
    ...  
}
```

# Назначение многопоточных приложений:

- использование одновременно нескольких вычислительных ядер для параллельного выполнения задач => **повышение производительности** приложения в целом за счет параллельного выполнения задач
- возможность выполнения одной задачи, пока другая задача ожидает какого-нибудь события => **повышение производительности** приложения в целом за счет использования вынужденных простоев

# До стандарта C++11

«поддержка» параллелизма только с помощью платформенно-зависимых расширений  
( например, `_beginthreadex()` )

Проблемы:

- портируемость
- не сформулирована модель памяти с учетом многопоточности
- отсутствует поддержка механизма обработки межпоточных исключений

# Преемственность от Boost

Из Boost Thread Library заимствованы:

- Функциональность
- Имена классов
- Имена методов и функций

# Отличия boost и C++11

- Boost поддерживает принудительное корректное завершение потока, C++11 нет
- в C++11 реализован `std::async`, в Boost нет
- Boost - `boost::shared_mutex` для multiple-reader/single-writer locking, в C++11 – нет, в C++14 `shared_timed_mutex`
- тайм-ауты организованы по-разному
- Отличия некоторых имен: `boost::unique_future` - `std::future`
- Каким образом передаются параметры потоковой функции: `boost::bind`, C++11 – variadic templates.
- в деструкторе `boost::thread` объект «отключается» (`detach()`) от системного потока.  
C++11 – если на момент вызова деструктора объект не отключен от системного потока, `std::terminate()` => приложение аварийно завершается

# Способы распараллеливания с точки зрения прикладного программиста:

- по задачам - каждый поток выполняет свою подзадачу (=> возникают зависимости между частями общей работы => **синхронизация выполнения**)
- по данным – каждый поток выполняет свою специфическую операцию над общими данными (=> возникает конкуренция за данные => **синхронизация доступа к данным**)



# Накладные расходы на запуск потока:

- ОС должна создать соответствующие служебные структуры данных (Windows – объекты исполняющей системы)
- выделить память для стека + доп. служебная память (TLS)
- «сообщить» о новом потоке планировщику
- время на переключение потока
- потоки – это ограниченный ресурс (ОС может накладывать квоты на количество потоков в процессе, адресное пространство – не «безразмерное»...). Частично эту проблему решают пулы потоков

=> если накладные расходы на запуск потока соизмеримы с временем его выполнения, то производительность приложения в целом может **ухудшиться** по сравнению с выполнением задач в одном потоке! => для достижения максимальной производительности нужно выбирать количество потоков исходя из аппаратного параллелизма

# На системном уровне

- Объект исполняющей системы thread описывается (ОС Windows):
  - дескриптор
  - идентификатор
- Задание приоритета только системными средствами (в C++11 соответствующих средств нет)
- Возможность задания ядер для выполнения – только системными средствами (Windows – `SetProcessAffinityMask()`, `SetThreadAffinityMask()` )

# Специфика:

1. выделение потоку времени для выполнения производится исключительно ОС, и ее в общем случае не интересует: относятся переключаемые потоки к одному процессу или к разным процессам
2. если в системе несколько ядер, ОС сама распределяет потоки по разным вычислительным ядрам
3. все потоки одного процесса выполняются в едином адресном пространстве, поэтому
  - «+» ???
  - «-» - ???
4. очередность выполнения потоков с одинаковым приоритетом определяется ОС => синхронизация
5. у каждого потока свой стек =>
  - с локальными переменными ???
  - глобальные и динамические ???

# ПОНЯТИЯ, СВЯЗАННЫЕ С ПОТОКАМИ

1. Первичный и вторичные потоки ???
2. Контекст потока ???
3. Состояние потока ???

# ПОТОКОВАЯ ФУНКЦИЯ

- Для **первичного** потока – стартовый код стандартной библиотеки (из которого вызывается `main()` )
- Для **вторичного** потока – программист должен предоставить функцию, с которой начнется выполнение потока

# Специфика потоковой функции

- порядок вызова обычных функций определяет программист, **потоковую функцию запускает система.**

# Отличия вызова функции и запуска потока

Вызов обычной функции	Запуск потока
Вызывающая функция и вызываемая пользуются одним и тем же стеком.	Стек у каждого потока (порождающего и порожденного) свой

# Отличия вызова функции и запуска потока

Вызов обычной функции	Запуск потока
Последовательность выполнения вызывающей и вызываемой функций предопределена и гарантируется компилятором.	Последовательность выполнения порождающего и порожденного потоков определяется системой, они относительно независимы.



# Отличия вызова функции и запуска потока

Вызов обычной функции	Запуск потока
Параметры передаются в том же стеке => компилятор гарантирует, что локальные данные вызывающей функции существуют, пока выполняется вызываемая (=> безопасно передавать адреса локальных переменных)	Параметр формируется в стеке потоковой функции. Данные, передаваемые потоковой функции должны гарантированно существовать, пока ими пользуется порожденный поток

Подводим итог:

Какие данные можно  
передавать дочернему  
потoku?

# Памятка программисту:

- Чем меньше разделяемых данных, тем меньше проблем!
- С локальными для потока данными проблем нет (если только адреса этих локальных данных не передаются другому потоку)!

# Преимущества/недостатки использования thread library C++11

- «+»
  - независимость от платформы,
  - абстрагирование действий
- «-»
  - потеря эффективности (несущественная) по сравнению с чисто системными средствами за счет использования классов-оберток,
  - ограниченность средств поддержки многопоточности по сравнению с системными возможностями

# C++11

Новая часть стандартной библиотеки предоставляет **платформенно-независимые объектно-ориентированные**:

- управление потоками
- защиту разделяемых данных
- синхронизацию потоков
- обработку межпоточных исключений
- атомарные операции

**но!** не поддерживает платформенно-независимое межпроцессное взаимодействие!

# Средства C++11 для поддержки МНОГОПОТОЧНОСТИ:

- **threads**,
- **mutexes**,
- **lock()** - операции,
- **packaged\_tasks**,
- **futures**
- **promises**
- **atomic**
- **thread\_local**
- ...

# Специфика

- Платформенная независимость
- Эффективность (несмотря на абстрагирование)
- Возможность «спускаться» на системный уровень

# Специфика реализация потоковой функции в C++11

- На системном уровне можно реализовать только посредством глобальной функции или статического метода класса
- Средствами высокоуровневой обертки C++11 шаблона `function` – все, что «можно вызвать» **callable**:
  - глобальную функцию
  - статический метод класса
  - функциональный объект
  - лямбда-функцию
  - `bind-er`
  - метод класса



# Специфика передачи параметров потоковой функции в C++11

- На системном уровне единственный параметр типа `void*`
- Реализация C++11 – **любое количество параметров любого типа**, так как реализована посредством **variadic template**

```
#include <thread>
```

```
namespace std
```

# НИЗКОУРОВНЕВЫЙ ИНТЕРФЕЙС ДЛЯ ЗАПУСКА ПОТОКОВ - КЛАСС **THREAD**

# std::thread

## Конструкторы:

- `thread();` //резервируется и инициализируется память под объект, привязки к потоку нет! **поток НЕ стартует!**
- `thread(const thread&) = delete; ???`
- `thread( thread&& other );`
- `template< class Function, class... Args >  
explicit thread( Function&& f, Args&&... args );`

## operator= :

- `thread& operator=(const thread&) = delete;`
- `thread& operator=(thread &&) ;`

# move конструктор копирования и move operator=

Передача «владения» потоком другому объекту. Это гарантия, что в каждый момент времени с потоком ассоциирован только один объект thread

Используется:

- для передачи в качестве параметра
- для возврата по значению
- чтобы объединить объекты thread в коллекцию

# Пример передачи в функцию:

```
void threadFunc(int n, char c)
{ for(int i=0; i<n; i++) {std::cout<<c;} }
```

```
void f(std::thread t){ ... }
```

```
int main(){
```

```
    std::thread t(threadFunc, 10, 'A');
```

```
    f(t); ///???
```

```
    ...
```

```
}
```

# Замечание:

Механизм **не** поддерживает задание параметров потоковой функции по умолчанию:

```
void threadFunc(char, size_t =10 );
```

```
int main()  
{  
    std::thread t1(threadFunc, 'A', 5);  
    //std::thread t2(threadFunc, 'B'); //"too few parameters "  
    ...  
}
```

# Пример – собираем объекты thread в контейнер:

```
void threadFunc(char, int);
```

```
int main(){  
    std::vector<std::thread> v;  
    for(int i=0; i<10; i++)  
    {  
        std::thread th(threadFunc, 'A'+i, i+10 );  
        v.push_back(th); //???  
    }  
    ...  
}
```

# Запуск потока

Посредством:

- конструктора с параметром (параметрами):
  - первым (обязательным) параметром должен быть callable (указатель на потоковую функцию, функциональный объект, ...)
  - все остальные параметры – параметры, которые требуется передать потоковой функции
- в случае неудачи генерирует `std::system_error`

При вызове конструктора с параметрами:

- значения параметров копируются в локальный анонимный объект (если при этом генерируются исключения, то генерация происходит в текущем потоке!)
- вызывается перегруженный `operator()`, который запускает системный поток с заданной потоковой функцией



# Замечание важное!

Системные потоки являются ограниченным ресурсом =>

если программист пытается создать потоков больше, чем предусмотрено в ОС,

генерируется исключение `std::system_error`

(даже если в потоковой функции не генерируется исключения - поexcept)

# Завершение потока

- естественное завершение - при завершении потоковой функции
- C++11 - принудительного завершения нет (только системными средствами, Windows – `ExitThread()` )
- аварийное завершение (при генерации исключения)
- Важно! в зависимости от используемой ОС при завершении первичного потока
  - процесс завершается => все вторичные принудительно завершаются
  - процесс не завершается => вторичные потоки продолжают работать

# Пример запуска потока посредством конструктора с параметром

```
void SimpleThreadFunc(){  
    std::cout<<"Hello, parallel world!";  
}  
  
int main(){  
    std::thread th(SimpleThreadFunc);  
  
    ...  
}
```

# Разница???

```
void SimpleThreadFunc() { ... }
```

```
int main()
```

```
{
```

```
    SimpleThreadFunc();
```

```
    std::thread th(SimpleThreadFunc);
```

```
    ...
```

```
}
```

# Использование лямбда-функции

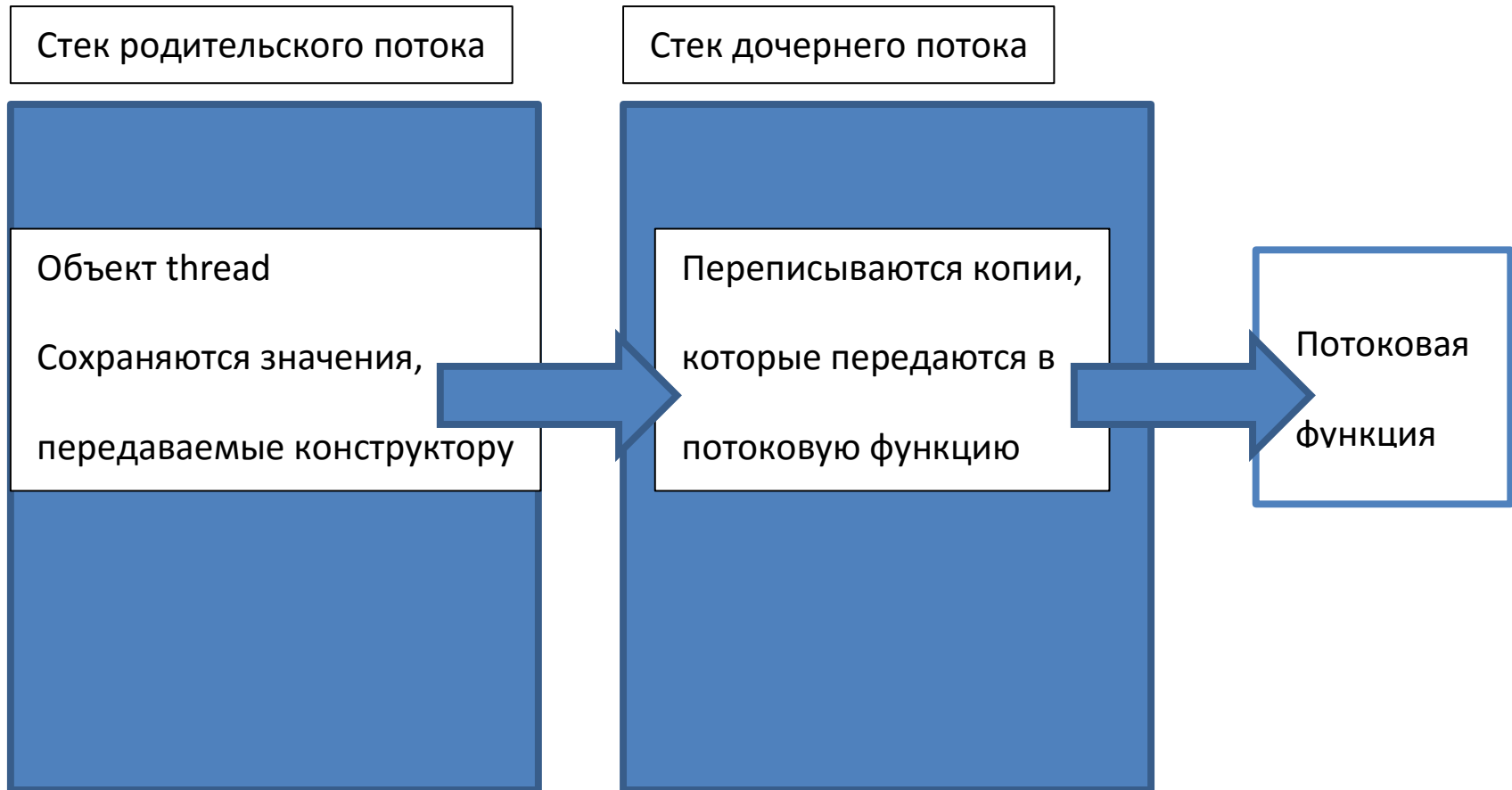
```
int main()
{
    std::thread th(
        [](){ std::cout<<"Hello, parallel World!"; }
    );
    ...
}
```

# Передача потоковой функции параметров по значению

```
void ThreadFuncParamsByValue(int n, double d)
{ std::cout << n << ' ' << d << std::endl; }
```

```
int main(){
    ...
    std::thread th(ThreadFuncParamsByValue, 5, 3.3);
    ...
}
```

# Передача параметров потоковой функции



# Использование семантики перемещения

```
void ThreadFuncByValue(std::string s) { ... }
```

```
int main() {
```

```
    ...
```

```
    std::string str("abc");
```

```
    std::thread th(ThreadFuncByValue,  
                   std::move(str));
```

```
    //str???
```

```
    ...
```

```
}
```



# Параметр – функциональный объект

```
class A {
```

```
...
```

```
void operator() ();
```

```
};
```

```
int main()
```

```
{
```

```
    std::thread th1( A() ); //???
```

```
    std::thread th2( ( A() ) );
```

```
    std::thread th3{ A() };
```

```
    ...
```

```
}
```

# Передача потоковой функции параметров по ссылке

```
class A{
    int m_a;
public:
    A(int=0);
    void Inc() { m_a++;}
};

void ThreadFuncParamsByRef(A& a) { a.Inc(); }

int main() {
    A a(1);
    std::thread th(ThreadFuncParamsByRef, a); //конструктор создает копию, а адрес
                                              //этой копии по ссылке передается в потоковую функцию
    ...
    // модификация a???
}
```

# Передача потоковой функции параметров по ссылке

```
class A{
    int m_a;
public:
    A(int=0);
    void Inc() { m_a++;}
};

void ThreadFuncParamsByRef(A& a) { a.Inc(); }

int main() {
    A a(1);
    std::thread th(ThreadFuncParamsByRef, std::ref(a));
    // модификация a???
    ...
}
```

# `std::ref()` и `std::cref()`

- шаблоны функций, которые создают объект типа `std::reference_wrapper`
- в котором (в свою очередь) запоминаются адреса передаваемых параметров, а не их копии

# Потоковая функция – шаблон

## ???

```
template<typename C> void templateThreadFunc(C& c)
{
    for (auto& e : c)    { e = -e; }
}
int main()
{
    std::vector<int> v{-1,5,-7,2,-9,0};
    std::thread th(templateThreadFunc<decltype(v)>, v); //???
    ...
}
```

# Потоковая функция - шаблон

```
template<typename C> void templateThreadFunc(C& c)
{
    for (auto& e : c)    { e = -e; }
}

int main()
{
    std::vector<int> v{-1,5,-7,2,-9,0};
    std::thread th(templateThreadFunc<decltype(v)>, std::ref(v));
    ...
}
```

# Варианты действий после запуска потока:

`std::thread th(ThreadFunc, <параметры>);`

- ***th.join();***  
родительский поток должен ожидать завершения дочернего (блокировка родительского потока)
- ***th.detach();***  
«отключает» дочерний поток от объекта `th` => дальнейшее взаимодействие с потоком посредством объекта `th` невозможно (не блокирует родительский поток)
- **Если ничего не вызвано**, в деструкторе объекта `th` будет вызвана `std::terminate()`

# thread::join()

```
void Draw(char c)
{ for (int i = 0; i < N; i++){std::cout << c;} }
```

```
int main()
{//если одно вычислительное ядро?
    std::thread t1(Draw, 'A');
    std::thread t2(Draw, 'B');
    std::thread t3(Draw, 'C');
    t1.join();
    t2.join();
    t3.join();
}
```



# thread::join()

```
void Draw(char c, int N)
{ for (int i = 0; i < N; i++){std::cout << c;} }

int main()
{
    std::vector<std::thread> v;
    for(char c='A'; c<='Z'; c++)
    { v.emplace_back(Draw, c,10); } //push_back ???
    for(auto t:v) //???
    { t.join(); }
}
```

# Специфика join:

- При вызове обнуляются данные объекта thread => с потоком объект уже не ассоциирован!
- => дважды вызвать нельзя, так как после первого вызова данные (дескриптор и идентификатор потока) обнуляются
- чтобы определить: связан объект thread с потоком или нет -> joinable()

# Специфика

## `bool thread::joinable() const`

- `true` – если объект ассоциирован с потоком
- если ПОТОК (на момент вызова `joinable()`) уже завершился - `true`
- после вызова default-конструктора – `false`
- после вызова `join()` – `false`
- после вызова `detach()` - `false`

# join() и исключения

## Проблемы

```
try{  
std::thread th(threadFunc, <параметры>);  
  
//код, который нужно выполнить до join()  
// и который может сгенерировать исключение  
  
th.join(); //если сгенерировано исключение, не  
будет выполнено! => в деструкторе th - terminate()  
} catch(...){}  

```

# Решение «в лоб»

```
std::thread th(threadFunc, <параметры>);
```

```
try{
```

```
//код, который нужно выполнить до join()
```

```
//и который может сгенерировать исключение
```

```
} catch(...){th.join();}
```

```
if(th.joinable()) { th.join();}
```

# Возможное решение (идиома RAII):

## Вариант №1

```
class thread_guard{  
    std::thread m_t;  
public:  
    explicit thread_guard(std::thread&& t)  
        : m_t(std::move(t) ){}  
    ~thread_guard(){if(m_t.joinable()) { m_t.join();} }  
    thread_guard(const thread_guard&) = delete;  
    thread_guard& operator=(const thread_guard&) = delete;  
};
```

# Иллюстрация использования:

```
{  
    std::thread th(threadFunc, <параметры>);  
    thread_guard tg( std::move(th) );  
  
    //код, который нужно выполнить до join()  
    // и который может сгенерировать исключение  
  
} //?
```

# Почему решили так НЕ делать:

```
void f() {... while(true){...} ...}  
int main()  
{  
...  
  {  
    std::thread t(f);  
    thread_guard thg(std::move(t));  
    ...  
  }//???  
...  
}
```



# Возможное решение (идиома RAII):

## Вариант №2

```
class thread_guard{  
    std::thread m_t;  
public:  
    explicit thread_guard(std::thread&& t)  
        : m_t(std::move(t) ){}  
    ~thread_guard(){if(m_t.joinable())  
                    { m_t.detach();} }  
    thread_guard(const thread_guard&) = delete;  
    thread_guard& operator=(const thread_guard&) = delete;  
};
```

# Почему решили и так НЕ делать:

```
void f(std::string* s) {...}
int main()
{
...
    {
        std::string x("abc");
        std::thread t(f,&x);
        thread_guard thg(std::move(t));

        ...
    }///???
...
}
```

А решили создатели стандарта  
**переложить ответственность на  
программиста** — в конце концов ему виднее,  
как программа должна обрабатывать  
подобные случаи...

# ФОНОВЫЕ ПОТОКИ - `thread::detach()`

```
void Draw(size_t n, char c)
{ for (int i = 0; i < n; i++){std::cout << c;} }
```

```
int main(){
    std::thread t1(Draw, 10, 'A');
    std::thread t2(Draw, 10, 'B');
    std::thread t3(Draw, 10, 'C');
    t1.detach();
    t2.detach();
    t3.detach();
    ...
}
```

# Проблемы при передаче потоковой функции параметров по ссылке

```
class A{
    int m_a;
public:
    A(int=0);
    void Inc() { m_a++;}
};

void ThreadFuncParamsByRef(A& a) { a.Inc(); }

void f() {
    ...
    A a(1);
    std::thread th(ThreadFuncParamsByRef, std::ref(a));
    th.detach(); //плохо!!! - почему?
    ...
}
```

# Передача потоковой функции параметров по ссылке

```
class A{
    int m_a;
public:
    A(int=0);
    void Inc() { m_a++;}
};

void ThreadFuncParamsByRef(A& a) { a.Inc(); }

void f{
    ...
    A a(1);
    std::thread th(ThreadFuncParamsByRef, std::ref(a));
    th.join(); //ждем завершения потока
    ...
}
```

# Передача параметров в лямбда-функцию по ссылке

```
int n=0;  
std::thread th( [&n]() { n++; } );  
th.join();
```

# Осторожно! Висячие ссылки и указатели!

```
class th{
    char* m_str;
public:
    th(char* s):m_str(s){}
    void operator()(){std::cout<<m_str;}
};

void func() {
    char s[] = {"qwerty"};
    std::thread t{ th(s) };
    t.detach(); //???
}
```



# Вызов метода класса в отдельном потоке

```
class A{
    int m_a;
public:
    A(int a= 0){ m_a = a; }
    void Inc() { m_a++; }
};

int main()
{
    A a(1);
    std::thread th(&A::Inc, &a);
    th.join();
    ...
}
```

# Вызов метода класса в отдельном потоке

```
class A{
    int m_a;
public:
    A(int a= 0){ m_a = a; }
    void Inc(int d) { m_a += d; }
};

int main()
{
    A a(1);
    std::thread th(&A::Inc, &a,5);
    th.join();
    ...
}
```

# Проблемы при передаче параметров (о которых нужно знать!)

```
void f(const std::string& );
```

Разница??? Проблемы???

```
char str[] = "abc";  
std::thread th(f,str);  
th.detach();
```

```
std::string str("abc");  
std::thread th(f,str);  
th.detach();
```

# Деструктор ~thread()

Важно!

- если объект ассоциирован с потоком (даже если поток уже завершился на момент вызова деструктора), **ТО** вызывается `std::terminate()`

# «Привязка» к системному потоку

`native_handle_type` thread::**native\_handle**();

- тип зависит от ОС. В ОС Windows – HANDLE
- может быть использован
  - для получения дополнительной (зависящей от ОС) функциональности (например, для задания приоритета)
  - в ущерб кроссплатформенности

# Пример `native_handle()`

```
#include <Windows.h>
```

```
int main()
```

```
{
```

```
    std::thread th(SimpleThreadFunc);
```

```
    HANDLE h = th.native_handle();
```

```
    SetThreadPriority(h, THREAD_PRIORITY_LOWEST);
```

```
    ...
```

```
}
```

# Класс `std::thread::id`

- хранит уникальный идентификатор потока (обертка для системного идентификатора)
- формируется при вызове конструктора
- возвращается методом `thread::get_id()`

## Специфика:

- может быть не ассоциирован с потоком (если default конструктор `thread` или после отключения потока от объекта)
- может быть использован как ключ для хранения объектов в ассоциативных контейнерах
- при завершении ассоциированного потока может быть использован для другого потока

# Идентификатор потока

```
#include <Windows.h>
{
    std::thread th(SimpleThreadFunc);

    DWORD idSystem= ::GetThreadId(th.native_handle());
    std::thread::id idC11 = th.get_id(); //запакован идентификатор

    std::cout << idC11; //public доступа нет, но перегружен operator<<
    ...
}
```



# Использование `std::thread::id` для хранения данных в `std::map`

```
std::vector<std::thread> th;  
std::map<std::thread::id, size_t> table;  
for(size_t i=0; i<5; i++){  
    th.emplace_back(threadFunc, <параметры>);  
    table[th.back().get_id()] = i;  
}  
//пользоваться table можно только до вызова join()  
for (auto& t : th)  
{ t.join(); }
```

# Количество ядер <-> количество ПОТОКОВ

```
static unsigned  
    std::thread::hardware_concurrency() ;
```

Замечание: если значение не определяется,  
то 0 => нужно учитывать такой вариант

# Параллельная реализация `std::accumulate()`

```
#include <numeric>
template<class InputIt, class T>
T accumulate(InputIt first, InputIt last, T init){
    for (; first != last; ++first) {
        init = init + *first;
    }
    return init;
}
```

```
std::vector<int> v;
//формирование значений
```

Параллельное суммирование - ???

# Параллельная реализация `std::accumulate()`

//Потоковая функция

```
template<typename It, typename T> void  
sum_block (It first, It last, T& result)  
{  
    result = std::accumulate(first, last, result);  
}
```

# Параллельная реализация std::accumulate(). Продолжение

```
int main(){  
    std::vector<int> v{ 1, 2, 3, 4, 5 , ...}; //большой!  
    std::vector<std::thread> th;  
    size_t nThreads = <лучше вычислить, исходя из количества  
                        ядер>  
    std::vector<int> res(nThreads+1); //для частичных сумм  
    size_t part = v.size() / nThreads ; //целое количество  
        элементов для суммирования каждым потоком  
    auto first = v.begin();
```

# Параллельная реализация `std::accumulate()`. Продолжение

```
for (size_t i = 0; i < nThreads; i++){
    auto last = first;
    std::advance(last, part);
    th.emplace_back(sum_block<decltype(first),int>,first, last, std::ref(res[i]));
    first = last;
}
//остаток досуммируем в данном потоке
if(first != v.end()){
    res[nThreads]=(std::accumulate(first, v.end(), 0));
}
for (auto& t : th) { t.join(); }
int sum = std::accumulate(res.begin(), res.end(), 0); //суммируем частичные суммы
```

```
#include <thread>
```

**ПРОСТРАНСТВО ИМЕН  
THIS\_THREAD**

# Специфика:

набор глобальных функций, которые

- можно использовать «внутри» любого потока:
  - как в коде любого порожденного потока,
  - так и в первичном потоке
- «извне» посредством объекта thread вызвать невозможно



# `void std::this_thread::yield();`

- отказ от остатка кванта времени => планировщик может назначить другой поток на выполнение

например:

```
while(!условие)
{std::this_thread::yield();}
```

# Функции `std::this_thread::`

- `std::this_thread:: sleep_for()` - исключает из планирования на указанный интервал (отправляет в спячку)
- `std::this_thread:: sleep_until()` - исключает из планирования до указанного момента времени
- `std::this_thread:: get_id()` - возвращает идентификатор текущего потока

# Пример использования sleep\_for()

```
using namespace std::chrono_literals;  
auto start =  
    std::chrono::high_resolution_clock::now();  
std::this_thread::sleep_for(1s);  
auto end =  
    std::chrono::high_resolution_clock::now();  
std::chrono::duration<double, std::milli>  
    elapsed = end-start;///  
///???
```

# **ОБРАБОТКА МЕЖПОТОЧНЫХ ИСКЛЮЧЕНИЙ**

# Некорректная обработка исключений между потоками ???

```
try
{
    std::thread th(threadFunction); //threadFunction может
                                    сгенерировать исключение

    th.join();
}
catch (const std::exception &ex)
{
    std::cout << ex.what() << std::endl;
}
```

`std::exception_ptr, std::current_exception(),  
std::rethrow_exception()`

`#include <exception>`

- **`std::current_exception()`**
  - обычно вызывается в обработчике исключения (catch)
  - возвращает объект `std::exception_ptr`, который содержит адрес объекта-исключения (реализован как класс с подсчетом ссылок)
  - если вызывается вне catch, то возвращается «пустой» объект `std::exception_ptr`
- **`std::rethrow_exception()`** — генерирует заново исключение, сохраненное в объекте `std::exception_ptr`

# Корректная (но не очень красивая) обработка исключений между потоками

```
std::vector<std::exception_ptr> g_exceptions;
```

```
void throw_function(){  
    throw std::exception("Error"); //здесь возможна гонка => нужно  
        «защищать» этот код, чтобы он выполнялся только одним потоком!  
}
```

```
void threadFunction(){  
    try  
    {  
        throw_function();  
    }  
    catch (...) { g_exceptions.push_back(std::current_exception()); }  
}
```

# Продолжение. Корректная обработка исключений

```
int main(){
    std::thread th(threadFunction);
    th.join();

    for(auto& e: g_exceptions) {
        try
        {
            if(e != nullptr)
                std::rethrow_exception(e);
        }
        catch (const std::exception &e){std::cout << e.what()<<std::endl;}
    }
}
```



# СИНХРОНИЗАЦИЯ ЗАДАЧ

# Синхронизация осуществляется посредством:

- задания приоритетов потоков – не поддерживается C++11
- синхронизирующих объектов
- условных переменных
- блокировок

# Общие правила безопасности многопоточной программы:

- без синхронизации доступ к одним и тем же данным
  - чтение данных можно разрешить **всем** потокам
  - для записи – опасно (соостояние за данные) => синхронизация

# Какие данные требуют синхронизации доступа?

- локальные?
- статические?
- динамические?

# Дж. Рихтер: «СИНХРОНИЗАЦИЯ ПОТОКОВ. ХУДШЕЕ, ЧТО МОЖНО СДЕЛАТЬ» - для одноядерной системы

```
BOOL gFlag = FALSE;           //глобальный флаг, синхронизирующий потоки

void threadFunc()//Потоковая функция
{
    ...           //выполняем какие-то вычисления
    gFlag = TRUE;    //с этого момента может продолжаться порождающий поток
    ...
    return 0;
}
//Порождающий поток
int main()
{
    ...
    std::thread(threadFunc);
    ...
    while(gFlag == FALSE); //ждем установки флага
    //продолжаем выполнение
}
```

# Чем плох такой подход, если в системе одно вычислительное ядро?

- вызывающий поток при таком решении проблемы не исключается из планирования => бесполезная работа;
- если у порождающего потока приоритет выше, чем у порожденного???
- если потоки выполняются в разных процессах, тогда эта переменная должна быть разделяемой;
- так как два потока пользуются одной и той же переменной - запретить компилятору оптимизировать работу с такой переменной (???)

# Вывод:

- для системы с **одним вычислительным** ядром потоки следует синхронизировать, исключая их из диспетчирования («**В спячку**»)!
- для системы с несколькими вычислительными ядрами это может быть не эффективно! => lock-free программирование

```
#include <mutex>
```

```
STD::MUTEX
```



# Главные проблемы конкурентного программирования:

- гонка за данными (**data race**)
- «голодание» (**resource starvation**) – поток блокируется, а условие освобождения не выполняется
- взаимоблокировка (**deadlock**) – более изощренная форма голодания, когда группа потоков не могут продолжить выполнение, так как они блокируют друг друга
- **livelock** – группа потоков не блокируется, при этом выполняют операции, которые не могут завершиться из-за ожидания захваченного другим потоком ресурса

## Важно!

- мьютексы решают (не всегда эффективно) гонку за данными (на высоком уровне), предоставляя возможность блокировки других потоков на время чтения/записи текущим потоком
- но **не** решают проблемы голодания и взаимоблокировок

# Классический пример взаимоблокировок

История обедающих философов – иллюстрация того, как несколько синхронизированных потоков соревнуются за лимитированные ресурсы:

- пять философов садятся за круглый стол
- перед каждым – тарелка риса
- между каждой парой философов – палочка для риса
- чтобы съесть порцию, у каждого философа должны быть две палочки (слева и справа)
- => deadlock

# data race

- Под состоянием гонки понимается любая ситуация, исход которой зависит от относительного порядка выполнения операций двух и более потоков (конкуренция за право выполнить операцию первым)
- Попытка модификации разделяемых данных может привести к неопределенному поведению

# Пример data race

```
std::vector<int> v ; //глобальный  
//формирование значений
```

//поток 1

```
std::cout<< v.top();
```

```
v.pop();
```

//поток 2

```
std::cout<< v.top();  
v.pop();
```

# мьютекс – взаимноисключающая блокировка

- позволяет выполнять защищенный мьютексом код только одному потоку
- => все синхронизируемые мьютексом потоки должны пользоваться одним и тем же объектом-мьютексом!!!
- => нужно гарантировать существование мьютекса, пока есть хотя бы один использующий его поток

# Варианты мьютексов:

- **std::mutex** - не рекурсивный
- **std::recursive\_mutex** - рекурсивный
- **std::timed\_mutex** - не рекурсивный, допускающий таймауты для блокирующих функций `try_lock_for()` и `try_lock_until()`
- **std:: recursive\_timed\_mutex** - рекурсивный мьютекс, допускающий таймауты для блокирующих функций.
- **std:: shared\_timed\_mutex** - разделяемый C++14
- **std::shared\_mutex** – C++17

# Предупреждение

- Опасно передавать указатели или ссылки на «защищенные» данные за пределы блокировки!
- Так как, если адрес будет запомнен, то возникает возможность обращения к разделяемым данным вне защищенной секции

# Инициализация мьютекса:

- `constexpr mutex();` - создает мьютекс в свободном состоянии
- `mutex( const mutex& ) = delete;`

Замечание:

- `mutex& jperator=( const mutex& ) = delete;`



# Важно!

Мьютекс должен гарантированно существовать, пока хотя бы один поток его использует, иначе «behavior of a program is undefined», т.е.

не должно быть ситуации, когда поток, захвативший мьютекс, аварийно завершается не освободив мьютекс!

# `std::mutex::lock()`

- если мьютекс свободен, то поток «вступает во владение» мьютексом и продолжает выполнение
- если мьютекс занят (другой поток успел вызвать для него `lock()`), то поток блокируется (исключается из диспетчирования)

**Замечание:** если владелец такого мьютекса еще раз вызовет `lock()`, поведение не определено!

# `std::mutex::unlock()`

- переводит мьютекс в свободное состояние
- вызов обязателен! Иначе мьютекс останется для всех ожидающих потоков в занятом состоянии!
- Важно! Мьютекс «передается во владение» захватившему его потоку => только захвативший поток может освободить занятый мьютекс

# std::mutex::try\_lock()

- неблокирующая попытка захвата мьютекса:
  - true – если мьютекс свободен, поток захватывает мьютекс и **продолжает** выполнение
  - false – если мьютекс занят, поток **продолжает** выполнение

# Получение дескриптора системного мьютекса

```
native_handle_type mutex::native_handle();
```

**Важно!** Это единственная возможность  
использовать мьютекс за пределами  
процесса

# Без мьютекса

```
void Draw(char c){ for (size_t i = 0; i < 20; ++i){ cout << c ; } }
```

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

//??? Порядок вывода? Если одно вычислительное ядро? Если несколько?

# Пример использования мьютекса

```
void DrawMutex(char c, std::mutex& m){  
    m.lock();  
    for (int i = 0; i < 20; i++){std::cout << c;}  
    m.unlock();  
}
```

```
int main(){  
    std::mutex m;  
    std::thread t1(DrawMutex, 'A', std::ref(m));  
    std::thread t2(DrawMutex, 'B', std::ref(m));  
    std::thread t3(DrawMutex, 'C', std::ref(m));  
    t1.join();      t2.join();      t3.join();
```

} //есть гарантия, что деструктор мьютекса не будет вызван пока потоки им синхронизируются? Имеет ли значение количество ядер?

# Пример try\_lock()

```
void f(std::mutex& m)
{
    while(m.try_lock() == false)
    {
        //делаем что-то другое (безопасное)
    }
    //работаем под защитой
    m.unlock();
}
```



# Взаимная блокировка

```
class A{  
    int* p;  
    size_t size, cap;  
public:  
    ...  
    void swap(A& other){???} //во время обмена  
                                другой поток/потоки могут читать/писать  
                                оба меняющихся данными объекта!  
};
```

# Взаимная блокировка

```
class A{
    int* p;
    size_t size, cap;
    std::mutex m;
public:
    ...
    void swap(A& other)
    {
        //нужно заблокировать оба объекта!
        //обмен
        //освобождение блокировки
    }
};
```

# Проблема!

Поток № 1

```
...  
a1.swap(a2);  
...
```

А a1, a2;

Поток № 2

```
...  
a2.swap(a1);  
...
```

# std::lock() для борьбы с deadlock-ами:

```
template< class Lockable1, class Lockable2, class... LockableN >  
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );  
    // Lockable* - должны иметь методы lock() , try_lock() и unlock()
```

последовательно пытается заблокировать все перечисленные объекты. Если при этом генерируется исключение, то для всех на этот момент заблокированных объектов вызывается unlock()

```
void A::swap(A& other){  
    if(this !=&other)  
    {  
        std::lock(this->m, other.m);  
        //обмен  
        this->m.unlock();  
        other.m.unlock();  
    }  
}
```

# std::try\_lock()

```
template< class Lockable1, class Lockable2,  
          class... LockableN> int  
try_lock( Lockable1& lock1, Lockable2& lock2,  
          LockableN&... lockn);
```

//возвращаемое значение

- -1 – ОК
- индекс мьютекса, который не удалось заблокировать

# Классы - «обертки» для мьютекса

Специфика: конструкторы классов «оберток» могут принимать параметр, определяющий политику блокировки:

- **defer\_lock\_t**: не захватывать мьютекс (отложить захват), а только «обернуть», чтобы в случае дальнейшего захвата мьютекса в деструкторе был вызван `unlock()`
- **try\_to\_lock\_t**: попытаться захватить мьютекс без блокировки
- **adopt\_lock\_t**: предполагается, что вызывающий поток уже захватил указанный мьютекс => требуется только в деструкторе вызвать `unlock()`

Типы и соответствующие объекты,  
предоставляемые стандартной библиотекой:

```
struct defer_lock_t { };  
struct try_to_lock_t { };  
struct adopt_lock_t { };
```

```
constexpr std::defer_lock_t defer_lock =  
    std::defer_lock_t();  
constexpr std::try_to_lock_t try_to_lock =  
    std::try_to_lock_t();  
constexpr std::adopt_lock_t adopt_lock =  
    std::adopt_lock_t();
```

# std::lock\_guard – самая простая обертка для мьютекса C++11

```
template< class Mutex > class lock_guard;
```

идиома **RAII** – «захват ресурса есть» инициализация

- в конструкторах:

lock\_guard(mutex\_type& m); – **lock()**

lock\_guard( mutex\_type& m, std::adopt\_lock\_t t ); -  
“оборачивает” мьютекс без вызова lock()

- в деструкторе – **unlock()**



# std::lock\_guard начиная с C++17

template< class... MutexTypes > class **lock\_guard**;  
обертка для любого количества мьютексов -  
аналогично

# Важно!

**//Глобальный мьютекс**  
**std::mutex m;**

```
void threadFunc()
{
    m.lock();
    //...
    //здесь может быть
    сгенерировано исключение!
    m.unlock(); //если
    сгенерировано исключение,
    мьютекс?
}
```

```
void threadFunc()
{
    std::lock_guard<std::mutex>
        l(m);
    //...
    //здесь может быть
    сгенерировано исключение!
    // мьютекс?
}
```

# Другие возможности lock\_guard

конструктор

```
lock_guard( mutex_type& m, std::adopt_lock_t t );
```

Специфика:

на момент создания объекта lock\_guard  
вызывающий поток уже **должен** владеть  
мьютексом, так как в деструкторе - ???

# Пример std::adopt\_lock

```
void f(std::mutex& m)
{
    while(m.try_lock() == false)
    {
        //делаем что-то другое (безопасное)
    }
    //сюда попадаем с захваченным мьютексом
    std::lock_guard(m, std::adopt_lock);
    //работаем под защитой
} // ???
```

# Модифицируем A::swap() с помощью std::lock\_guard

```
void A::swap(A& other){  
    if(this==&other) return;  
    std::lock(this->m, other.m);  
  
    std::lock_guard<std::mutex>  
        lock_this(this->m, std::adopt_lock );  
    std::lock_guard<std::mutex>  
        lock_other(other.m, std::adopt_lock );  
    //обмен  
} ???
```

# `std::unique_lock`

- гарантирует: в каждый момент времени мьютексом может владеть только один объект `std::unique_lock` !

# **std::unique\_lock** – обертка для мьютекса с бОльшей функциональностью по сравнению с **lock\_guard**

отличия от **std::lock\_guard**:

- **std::lock\_guard** заблокирован на всем протяжении своего существования  
**std::unique\_lock** – добавляет **признак** + возможность узнать свободен/захвачен мьютекс - **owns\_lock()**
- **std::lock\_guard** - мьютекс блокируется один раз в конструкторе, освобождается тоже только один раз – в деструкторе  
**std::unique\_lock** – можно блокировать и освобождать посредством **lock()** и **unlock()**, а также **try\_lock()**, **try\_lock\_for()**, **try\_lock\_until**
- **std::lock\_guard** – политика захвата мьютекса может быть только **std::adopt\_lock**  
**std::unique\_lock** – может быть **adopt\_lock**, **defer\_lock** или **try\_to\_lock**

# 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 выполняются ни `lock()`, ни `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;`

# Пример:

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

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

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

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

```
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(); //???
```

# Модифицируем A::swap() с помощью std::unique\_lock

```
void A::swap(A& other){  
    std::unique_lock <std::mutex>  
        lock_this(this->m, std::defer_lock );  
    std::unique_lock <std::mutex>  
        lock_other(other.m, std::defer_lock );  
    std::lock(lock_this, lock_other);  
        //обмен  
} ???
```

# Дополнительные возможности `unique_lock`:

## Конструкторы:

- `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){
        //...
        m.unlock();
    }
}
```

# Обертки для `std::timed_mutex`

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



# Использование обертки для timed\_mutex

```
std::timed_mutex m;
```

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

Можно (имеет смысл) использовать lock\_guard?

# 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**

# Специфика:

Для эксклюзивного использования (владение):

`lock()`, `unlock()`, `try_lock()`

Для совместного использования (разделение):

`lock_shared()`, `unlock_shared()`,  
`try_lock_shared()`

# Простой пример:

```
class Shared {  
    int m_a=0;  
    mutable std::shared_mutex m;  
public:  
    int Get()const{ //читать могут все потоки  
        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();  
    }  
};
```



# 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& );
```

# 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).