Стандарт C++11 – C++14 – C++17 – C++20 multithreading

Полубенцева Марина Игоревна

История C++

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
• Templates • STL with containers and algorithms • Strings • I/O Stream	• Move semantic • Unified initialization • `auto` and `decltype` • Lambda expressions • `constexpr` • **Multithreading and the memory model** • Regular expressions • Smart pointers • Hash tables • `std::array`	• **Reader-writer locks** • Generic lambdas • Extended `constexpr` functions	• Fold expressions • `constexpr if` • Structured binding • `std::string_view` • **Parallel algorithms of the STL** • Filesystem library • `std::any`, `std::optional` and `std::variant`	• **Coroutines** • ~~Contract~~ • Modules • Concepts • Ranges library

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

CHRONO
ЗАДАНИЕ/ИЗМЕРЕНИЕ ИНТЕРВАЛОВ И МОМЕНТОВ ВРЕМЕНИ

М.Полубенцева

До C++11 - C-style date and time library
#include <ctime>

Типы:

- clock_t
- time_t
- tm

Функции:

- clock()
- time()
- difftime()
- localtime()
- ctime()
- mktime()
- …

Проблема:

- работа со временем без явного указания единиц измерения!

Например: clock() возвращает количество временных тактов, прошедших с начала запуска программы.
Чтобы получить секунды, нужно поделить на CLOCKS_PER_SEC (а если хочется получить значение в мкс?, мс?...)

Библиотека <chrono> предоставляет:

- Три основных сущности (шаблоны классов):
  - часы (**clocks**)
  - моменты времени (**time points**)
  - интервалы времени (**duration**)
- Функции для получения и задания
  - моментов (для указанных часов)
  - и интервалов времени

Часы:

- определяют:
  - эпоху (точку отсчета)
  - и продолжительность такта
- момент времени может быть задан только для конкретных часов

Например:

- часы могут иметь точку отсчета 1 января 1970г и такт, измеряемый в 100нс

#include <chrono>
C++11 предоставляет три вида часов:

- ***system_clock*** (не стабильны!) - 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 - time stamp counter)
- ***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

- эквивалентом:
  - системных часов
  - или стабильных часов

- или реализованы третьим образом

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:

Если для перечисленных функций **моменты времени** задаются посредством **НЕ**стабильных часов (system_clock),

то!

механизм ожидания **учитывает** тот факт, что часы могли «подвести» => срабатывание может произойти как раньше (перевели часы вперед), так и позже (перевели часы назад)

Информация, предоставляемая chrono:

- текущее время – статический метод «часов» - **now**()
- тип, для хранения «абсолютного» 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 также минимизировать потерю точности 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
- constexpr!

#include <ratio>

Вспомогательный класс для хранения дроби:

```
template<
    std::intmax_t Num,          //числитель
    std::intmax_t Denom = 1 //знаменатель
> 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> sec = end-start; //в сек
std::chrono::duration<int> sec = end-start; //ош
std::chrono::duration<long long, std::ratio<1, 1000000000>>
                    nanosec = end - start; //в наносек
```

Полезные псевдонимы:

type	Representation	Period
hours	*signed integral type* of at least 23 bits	using hours = duration<int, ratio<3600,1>>;
minutes	*signed integral type* of at least 29 bits	ratio<60,1>
seconds	*signed integral type* of at least 35 bits	ratio<1,1>
milliseconds	*signed integral type* of at least 45 bits	ratio<1,1000>
microseconds	*signed integral type* of at least 55 bits	ratio<1,1000000>
nanoseconds	*signed integral type* of at least 64 bits	ratio<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); //???

//Использование псевдонимов:
std::chrono::seconds sec(10);
std::chrono::hours h(5);
std::chrono::milliseconds ms(2);
```

Замечания:

- 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)==3мс

std::chrono::duration<double, std::ratio<1, 30>> dur(3.5);
std::cout << dur.count(); //3.5 (тика) *  (1/30)
```

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.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
-, +, ==, <, % ….

```
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;
```

C++17 !!! Добавлены шаблоны глобальных функций (std::chrono::):

floor(std::chrono::duration) – преобразование с округлением «вниз»

```
std::chrono::minutes m4 = floor<std::chrono::minutes>(100s); //1m
```

ceil(std::chrono::duration) – преобразование с округлением «вверх»

```
std::chrono::minutes m2 = ceil<std::chrono::minutes>(-100s); //-1m
std::chrono::minutes m3 = ceil<std::chrono::minutes>(100s); //2m
```

round(std::chrono::duration) – преобразование с округлением «к ближайшему»

```
std::chrono::minutes m1 = round<std::chrono::minutes>(100s); //2m
```

abs(std::chrono::duration)

C++20 !!!

- будут удалены ==, !=, <, <=, >, >=

  вместо них **<=>**

- добавлен operator<<
- добавлен шаблон глобальной функции

  std::chrono::from_stream()

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));
std::cout << h.count() << ':' << m.count() << ':' << s.count()
                                        << ':' << mss.count();
```

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

- ==, !=, <, <=, >, >= (в C++20 будут удалены!) => **<=>**
- C++20 – operator++(), operator++(int), operator--(), operator--(int)
- статические методы min(), max()

C++17 Шаблоны функций

- **floor**(std::chrono::time_point) – преобразует один момент времени в другой, округляя «вниз»
- **ceil**(std::chrono::time_point) – «вверх»
- **round**(std::chrono::time_point) – использует ближайшее значение

Преобразования std::chrono::**time_point** <-> std::**time_t**

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

```
std::tm tTm = { 0 };
tTm.tm_year = 2020 - 1900;
tTm.tm_mon = 4;
tTm.tm_mday = 20;
tTm.tm_hour = 19;
//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();
//std::chrono::time_point<std::chrono::
high_resolution_clock,std::chrono::duration<__int64,std::ratio<1,1000000000> > >

//измеряемый фрагмент

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

//а результат интересует в нс

long long n= (**stop – start).count()**;

// или в мс

long long ms= (std::chrono::duration_cast<std::chrono::milliseconds>
(stop - start)). count();

???

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

//измеряемый фрагмент

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

std::chrono::duration<int> t = end - start; //???
```

std::chrono::time_point_cast

- преобразует интервал из одних единиц в другие
- для одних и тех же часов!
- при потере точности значение не округляется, а отсекается

Пример time_point_cast

```
using namespace std::chrono_literals;
std::chrono::
   time_point<std::chrono::steady_clock> tpSec (1s); //неявное
                                                преобразование в нс
std::cout << tpSec.time_since_epoch().count() <<
        " ns"<<std::endl; //1.000.000.000 ns
std::cout << (
std::chrono::time_point_cast<std::chrono::seconds>(tpSec).
                time_since_epoch()
                ).count() << " s"; //1 s
```

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

М.Полубенцева

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++11-14-17-20)**

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

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

Замечание (специфика Microsoft):

Auto-Parallelizer – позволяет компилятору обнаруживать в тексте и распараллеливать циклы (если cost/benefit). Например:

```
void f (int n){//неизвестно – насколько велико n
for (int i=0; i<n; ++i)  {      A[i] = B[i] * C[i]; } }

//программист знает, что n всегда будет большим! =>
void f(int n) {
#pragma loop(hint_parallel(4))  //4 потока
   for (int i=0; i<n; ++i)  {  A[i] = B[i] * C[i]; }
}
```

ПРОПИСНЫЕ ИСТИНЫ (КОРОТКО)

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

- задача разработчика схемотехники – предоставить средства, **задача компилятора и продвинутого программиста** — наиболее эффективно эти средства задействовать
- с другой стороны: 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:

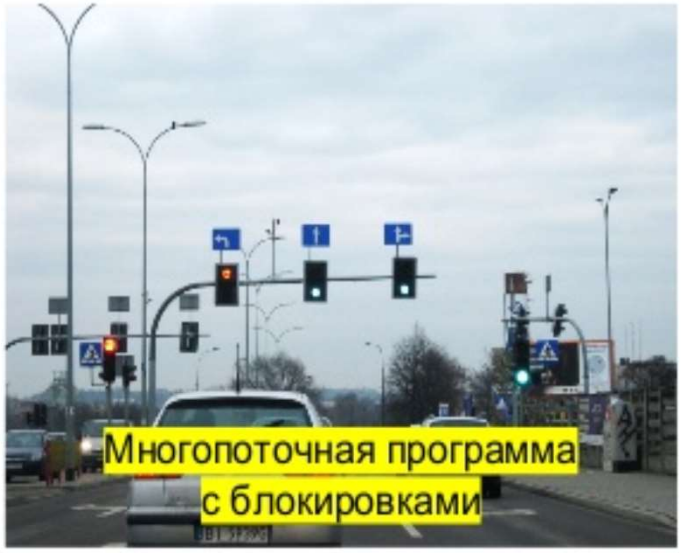
- Аппаратно. На компьютере может быть:
  - несколько процессоров
  - многоядерный процессор
  - комбинация двух предыдущих вариантов
- Программно:
  - «псевдо-параллельно» - переключение потоков ОС
- Класс thread – это низкоуровневая «обертка» для системных потоков

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!

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

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)

Низкоуровневые средства организации многопоточности

КЛАСС STD::THREAD

М.Полубенцева

Класс std::thread
<thread>

- это кроссплатформенная обертка для запуска системного потока!

Наступила эпоха «стандартной» параллельности!

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

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

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 (C++20 - да)
- 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, C++17 - shared_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 (tuple).
- 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. Но! переключение потоков, принадлежащих разным процессам, более ресурсоемко!

2. если в системе несколько ядер, ОС сама распределяет потоки по разным вычислительным ядрам. Но! чем дальше ядро от шины данных?

3. все потоки одного процесса выполняются в едином адресном пространстве, => +/- ???

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 ???

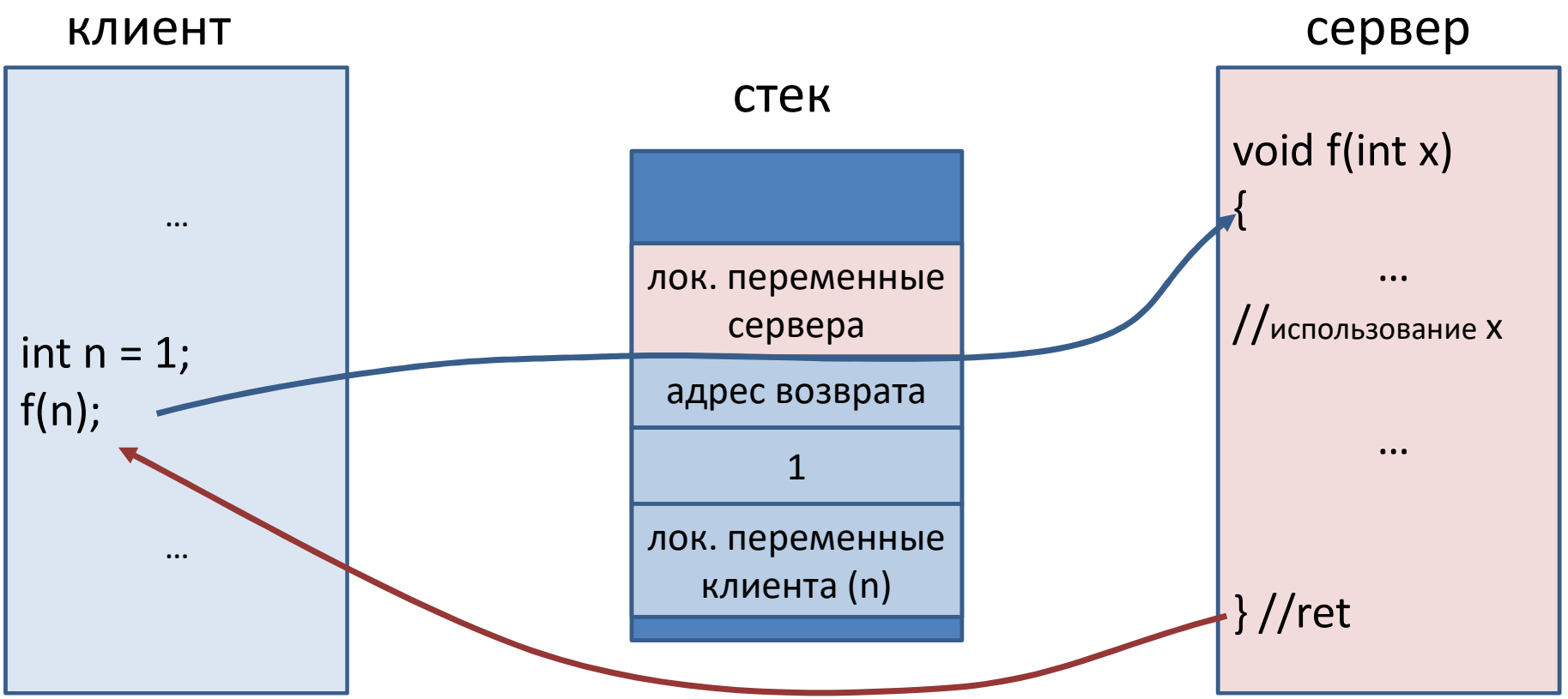
ПОТОКОВАЯ ФУНКЦИЯ

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

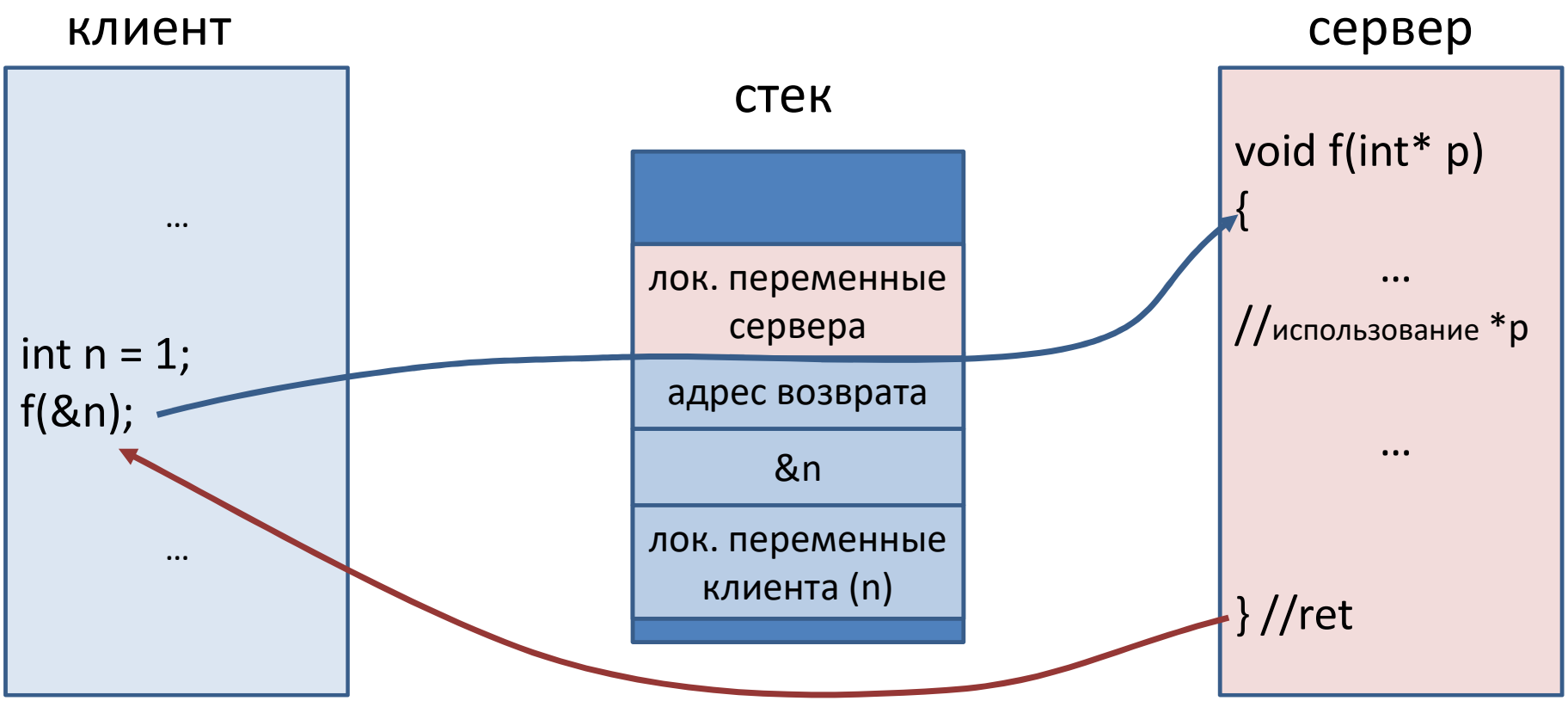
Специфика потоковой функции

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

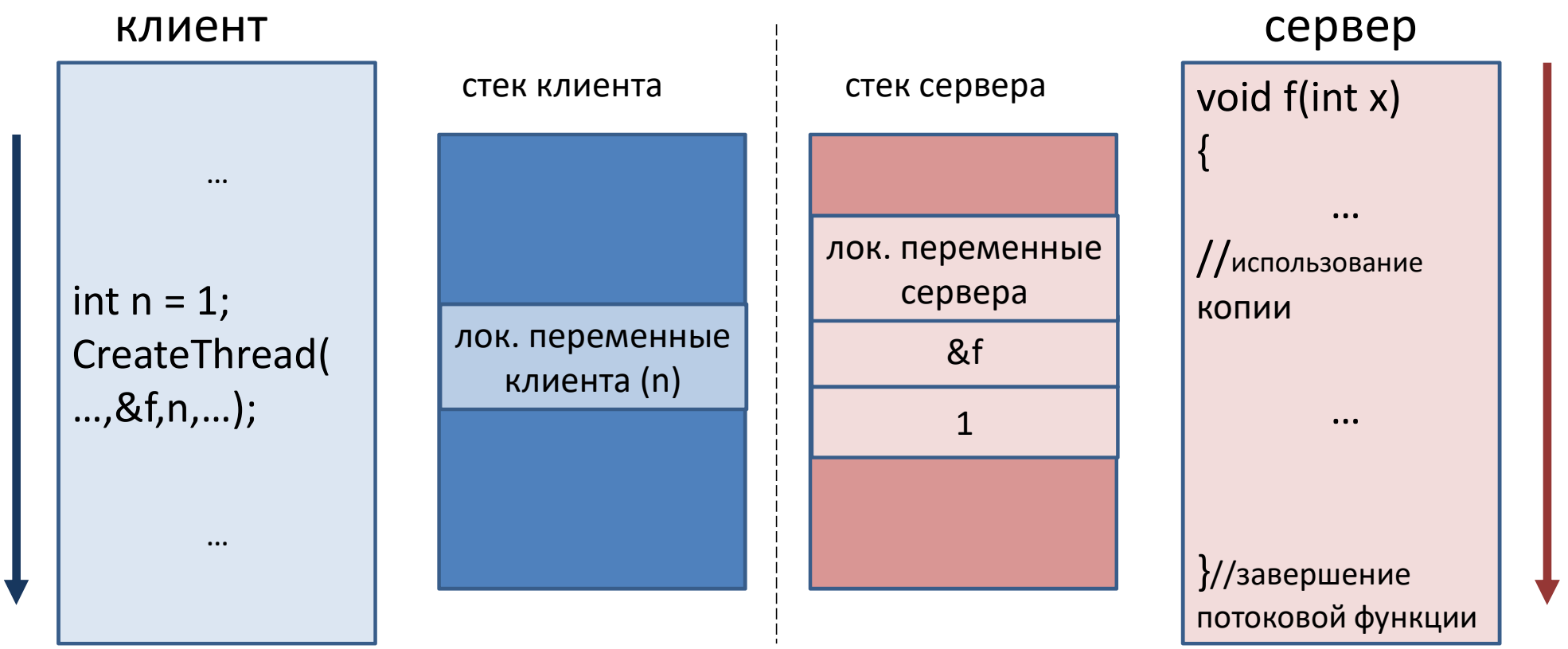
Отличия вызова функции и запуска потока.
Вызов обычной функции:



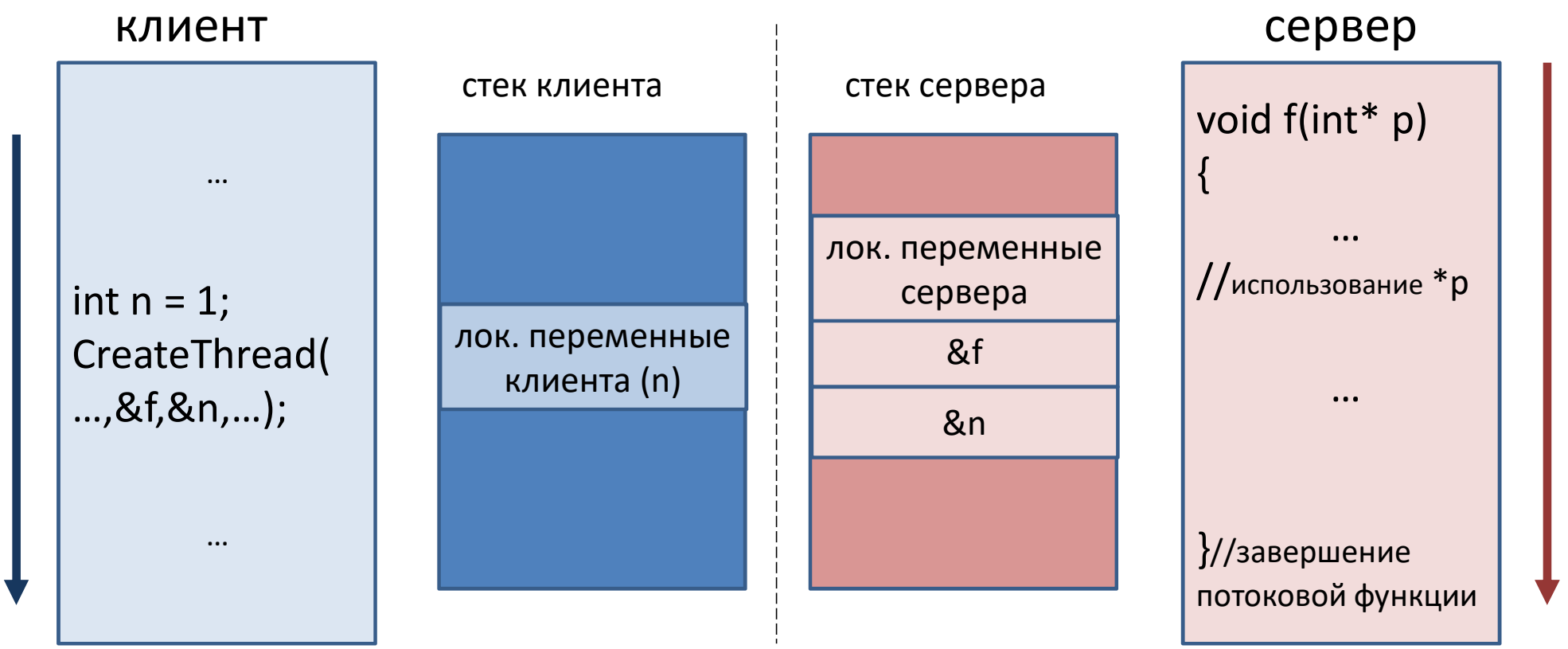
Отличия вызова функции и запуска потока.
Вызов обычной функции:



Отличия вызова функции и запуска потока.
Запуск потока:



Отличия вызова функции и запуска потока. Запуск потока:



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 потоку?

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:

- **thread**s,
- **mutex**es,
- **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-ер
  - 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 (tuple)**

#include <thread>

namespace std

НИЗКОУРОВНЕВЫЙ ИНТЕРФЕЙС ДЛЯ ЗАПУСКА ПОТОКОВ - КЛАСС THREAD

М.Полубенцева

Класс std::thread – private данные для Win32:

```
void* _Hnd; // Win32 HANDLE
unsigned int _Id;
```

Класс std::thread – основная функциональность:

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 &&) ;

«Под капотом» (упрощенно):

```
void f(int a, int b){ … }

int main(){
  int x=1, y=2;
  std::thread th1(f1, x, y); //в конструкторе:
        //1. динамически создается std::tuple<void (__cdecl*)(int,int),int,int>
        //2. запускается системный поток (Win32 - _beginthreadex()),  при этом в качестве
        указателя на потоковую функцию – указатель на служебную функцию,
        //3. а в качестве указателя на данные – указатель на динамический tuple
        //4. которая начнет выполнение в новом потоке с вызова f() с сохраненными в tuple
        параметрами

  …
}
```

Иллюстрация «под капотом»:

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

void f(std::thread t){ … }

int main(){
        std::thread t(threadFunc, ‘A’ , 10 );
        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); //OK
        //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:

1.

объект thread создается в стеке родительского потока (=> если при этом генерируется исключение, то генерация происходит в родительском потоке!).

объект thread обеспечивает:

- взаимодействие дочернего и родительского потоков
- связь с системным потоком (хранение системного дескриптора и идентификатора)

При вызове конструктора с параметрами – продолжение:

2. Подготовка к запуску.

динамически создается std::tuple, в который запаковывается

- callable
- + параметры, предназначенные для потоковой функции

При вызове конструктора с параметрами – продолжение:

3. Инициирование запуска потока (системо-зависимо – в Win32 _beginthreadex()):

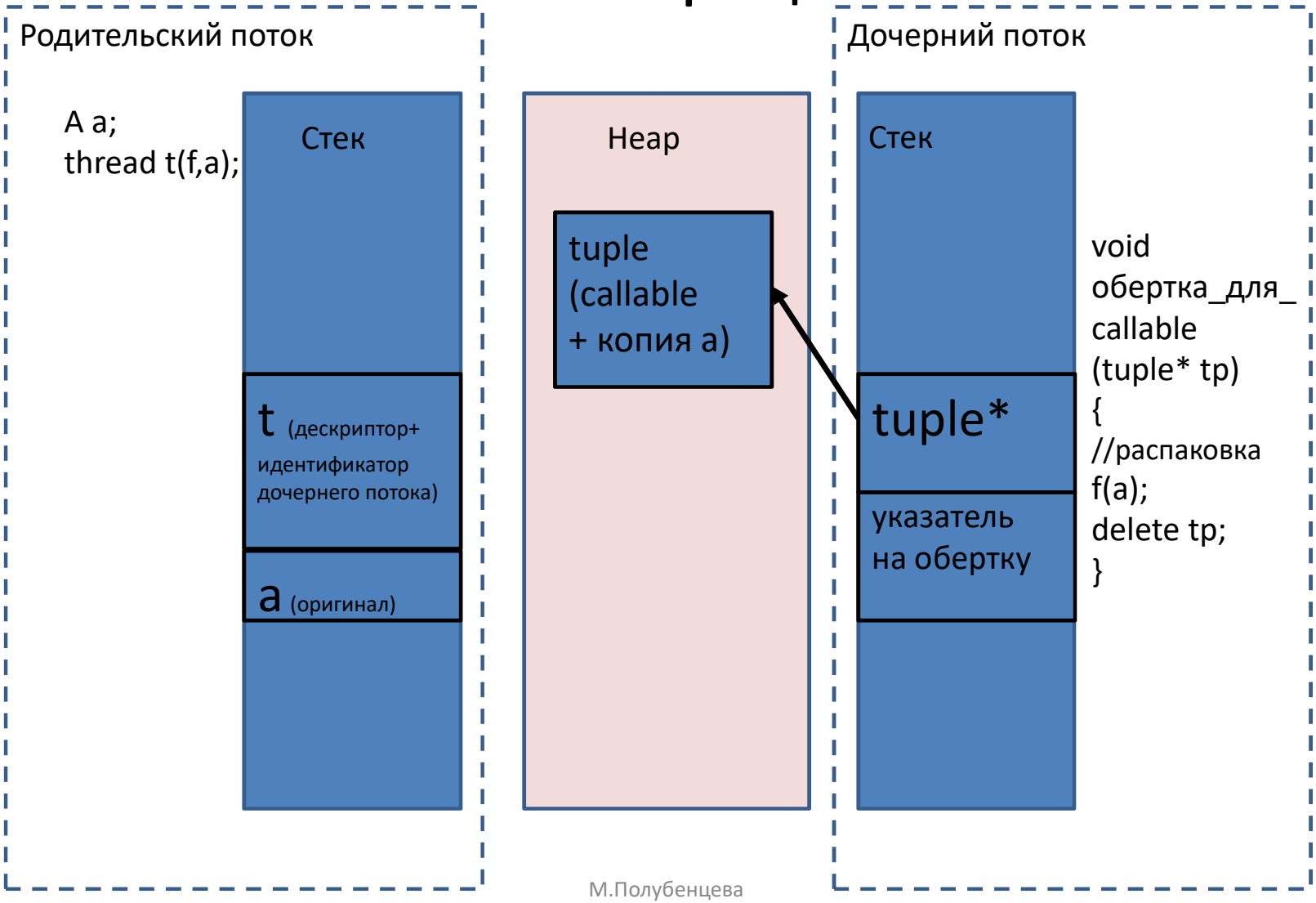
- в качестве указателя на потоковую функцию - указатель на служебную функцию (__Launch())
- в качестве указателя на данные – указатель на динамический tuple
- + формируемые функцией _beginthreadex() системные дескриптор и идентификатор сохраняются в объекте thread родительского потока (_Thrd_t _Thr; - структура, предназначенная для хранения системного дескриптора и идентификатора потока)

При вызове конструктора с параметрами – продолжение:

4. В дочернем потоке выполняется служебная (вспомогательная) функция, которая:

- принимает указатель на tuple
- использует первый член tuple в качестве callable
- все остальные члены tuple в качестве параметров callable
- после завершения callable уничтожает tuple

Иллюстрация:



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 - noexcept)

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);
        …
}
```

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 указателю

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

void ThreadFuncParamsByPointer(A* a) {  a->Inc(); }

int f() {
        A a(1);
        std::thread th(ThreadFuncParamsByPointer, &a);
        …
        // модификация a???
        //проблемы???
}
```

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 f() {
        A a(1);
        std::thread th(ThreadFuncParamsByRef, a); //в tuple запаковывается копия, 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 а???
        //проблемы???

        …
}
```

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

Альтернатива - использование std::forward_as_tuple()

```
void fThreadTuple(std::tuple<int&, std::string& > t){
        std::get<std::string&>(t)[0]++;
        std::get<int&>(t)++;
}
int main(){
        int x=1;
        std::string str("abc");
        std::thread th(fThreadTuple,
                        std::forward_as_tuple(x, str));
        …
}
```

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));
   …
}
```

Важно!

- Возвращаемое потоковой функцией значение игнорируется
- Если потоковая функция генерирует исключение (и не обрабатывает его сама), то вызывается std::terminate()

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

int main(){//если одно вычислительное ядро?
        std::thread t1(Draw, 'A', 10);
        std::thread t2(Draw, 'B', 20);
        std::thread t3(Draw, 'C', 30);
        t1.join();
        t2.join();
        t3.join();
        std::cout<<“main”; //????
}
```

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); }
        for(auto t:v) //???
        {  t.join(); }
}
```

Специфика join:

- При вызове обнуляются данные объекта thread => с системным 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

- если объект ассоциирован с потоком – true
- 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();}
```

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();
         std::cout <<“main”;
        …
}
```

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…

Когда можно создавать анонимные объекты thread?

```
void f1(int n){…}
void f2(std::thread th){ th.join(); }

int main(){
   std::thread{ f1 , 1}; //???
   f2(std::thread{ f1 , 2 }); //???
   std::vector<std::thread> v = { std::thread{f1 ,2}, std::thread{f1 ,3} };
                                                                  //???
         …
}
```

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

А так?

```
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
        …
}
```

Разница?

```
void f(const std::string& s){}

int main(){
        std::string s1(“abc”), s2(“123”);
//1.
        std::thread t1(f, s1);
        t1.detach();    //???
//2.
        std::thread t2(f, std::ref(s1));
        t2.detach();    //???
}
```

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();
        …
}
```

Продолжение - 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!)

<table>
<tr><td colspan="2">void f(const std::string&);</td></tr>
<tr><td colspan="2">Разница??? Проблемы???</td></tr>
<tr><td>char str[] = “abc”;
std::thread th(f,str);
th. detach();</td><td>std::string str(“abc”);
std::thread th(f,str);
th. detach();</td></tr>
</table>

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 или завершения потока)
  - 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 (если системный поток успешно запущен)
- возвращается методом thread::get_id()

Специфика:

- может быть не ассоциирован с потоком (если default конструктор thread или после отключения потока от объекта – данные обнуляются)
- может быть использован как ключ для хранения объектов в ассоциативных контейнерах

Использование идентификатора 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
```

C++20

STD::JTHREAD
STD::STOP_TOKEN

М.Полубенцева

Класс std::jthread <jthread>

- обладает дополнительной возможностью - корректно завершить поток «извне»
  - дополнительное private данное - std::stop_source
  - конструктор jthread может принимать параметр (он должен быть первым) типа std::interrupt_token
  - дополнительный public метод – interrupt()
- если на момент вызова деструктора поток является joinable(), вызывается request_stop() и join()

функциональность std::interrupt_token

методы	
valid()	true – если можно остановить можно
is_interrupted()	true, если interrupt_token==true Или interrupt_token==false и была вызвана interrupt()
interrupt()	if(!valid() || is_interrupted()) – ничего не делает В остальных случаях устанавливает interrupt_token => is_interrupted() == true

Последовательность действий:

- Создать std::stop_source
- Получить std::stop_token из std::stop_source
- Передать std::stop_token запускаемому потоку в качестве параметра
- Для завершения потока source.request_stop()
- Периодический вызов token.stop_requested() для выявления требования остановки
  Важно! если не вызвать token.stop_requested(), ничего не происходит

Пример – jthread без остановки потока

```
int main(){
    std::jthread nonInterruptable([]{                    // (1)
        int counter{0};
        while (counter < 10){
            std::this_thread::sleep_for(0.2s);
            std::cerr << "nonInterruptable: " << counter << std::endl;
            ++counter;
        }
    });
    std::this_thread::sleep_for(1s);
    nonInterruptable.interrupt();
}
```

Пример – jthread с остановкой потока

```
int main(){
 std::jthread interruptable([](std::interrupt_token itoken){        // (2)
       int counter{0};
       while (counter < 10){
          std::this_thread::sleep_for(0.2s);
          if (itoken.is_interrupted()) return;                      // (3)
          std::cerr << "interruptable: " << counter << std::endl;
          ++counter;
       }
   });
  std::this_thread::sleep_for(1s);
   interruptable.interrupt();
}
```

#include <thread>

ПРОСТРАНСТВО ИМЕН THIS_THREAD

М.Полубенцева

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 (поток не исключается из планирования, а переводится в состояние готовности)
- зависит от конкретной реализации (ОС), например, планировщик может поставить данный поток в конец очереди потоков с данным приоритетом и 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 (shared_ptr?) 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");}

void threadFunction(){
    try
    {
        throw_function();
    }
    catch (...)     { g_exceptions.push_back(std::current_exception());
            //здесь возможна гонка => нужно «защищать» этот код, чтобы он
                        выполнялся только одним потоком!
            }
}
```

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

М.Полубенцева

Concurrency != Parallelism

Concurrency	Parallelism
Цель: эффективная организация программы через взаимодействие отдельных компонент	**Цель:** эффективное использование «железа» и масштабируемая производительность
Способ: взаимодействующие потоки, синхронизация, ожидание и обработка событий	**Способ:** независимые задачи, которые можно исполнять одновременно в разных потоках

Аналогия:

Синхронизация осуществляется посредством:

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

Дж. Рихтер: «СИНХРОНИЗАЦИЯ ПОТОКОВ. ХУДШЕЕ, ЧТО МОЖНО СДЕЛАТЬ»

в системе с одним вычислительным ядром

```
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?

???

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

data race
(**неопределённость параллелизма**)

- Под состоянием **гонки** (конкуренции) понимается любая ситуация, исход которой зависит от относительного порядка выполнения операций двумя и более потоками (конкуренция за право выполнить операцию первым)
- Попытка модификации **разделяемых** данных может привести к неопределенному поведению
- Важно! состояние гонки — «**плавающая**» ошибка
  => проявляется не стабильно (скорее очень редко!)
  => печальное следствие для программиста: отладка (трудно выявляемый bug)!!!

Возможные последствия (если программист допустил при проектировании появление гонки)

- утечки ресурсов
- порча данных (например, частичная модификация или попытка получения уже недействительных данных)
- взаимные блокировки (dead lock)
- …

Иллюстрация data race:

std::stack<int> s ; //глобальный
//формирование значений	

//поток 1
 	
if(!s.empty())
{
std::cout<< s.top();
 	
s.pop();
}
	//поток 2
	
	if(!s.empty())
	{
	std::cout<< s.top();
	s.pop();
	}

мьютекс – взаимоисключающая 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

Важно!

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

Рекомендация:

- защищаемая секция должна быть как можно меньше

Инициализация мьютекса:

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

// mutex(mutex&&) – не определен!

```
std::mutex m; //ОК
std::mutex m1 = std::move(m); //???
```

Замечание:

- mutex& operator=(const mutex&) **= delete**;

Важно!

- мьютекс должен гарантированно существовать, пока хотя бы один поток его использует, иначе «behavior of a program is undefined»
- 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 (переводит мьютекс в занятое состояние) и продолжает выполнение. Операция захвата выполняется **атомарно**!
- если мьютекс занят (другой поток успел вызвать для него lock()), то поток блокируется

Замечание: если владелец такого мьютекса еще раз вызовет lock(), поведение не определено (вплоть до deadlock-а)!

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, иначе «the behavior is undefined»

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

Важно!

std::mutex m; //свободен

m.lock(); //захват мьютекса

…

bool b = **m.try_lock();** //в том же потоке => “undefined behavior”

m.unlock();

Получение доступа к системному примитиву

native_handle_type mutex::**native_handle**();

Специфика:

- наличие функции зависит от конкретной реализации
- Concurrency::critical_section *

Без мьютекса

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

int main(){
        thread t1 (Draw, 'A', 20);
        thread t2(Draw, 'B', 20);
        thread t3 (Draw, 'C', 20);

        t1.join();  t2.join();  t3.join();

        cout<<"main";

} //??? Порядок вывода?
```

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', m); //???
        std::thread t2(DrawMutex, 'B', m);
        std::thread t3(DrawMutex, 'C', m);
        t1.join();        t2.join();        t3.join();
        cout<<“main”;
}
```

Корректный 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();
        cout<<“main”;
} //есть ли 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();
}
```

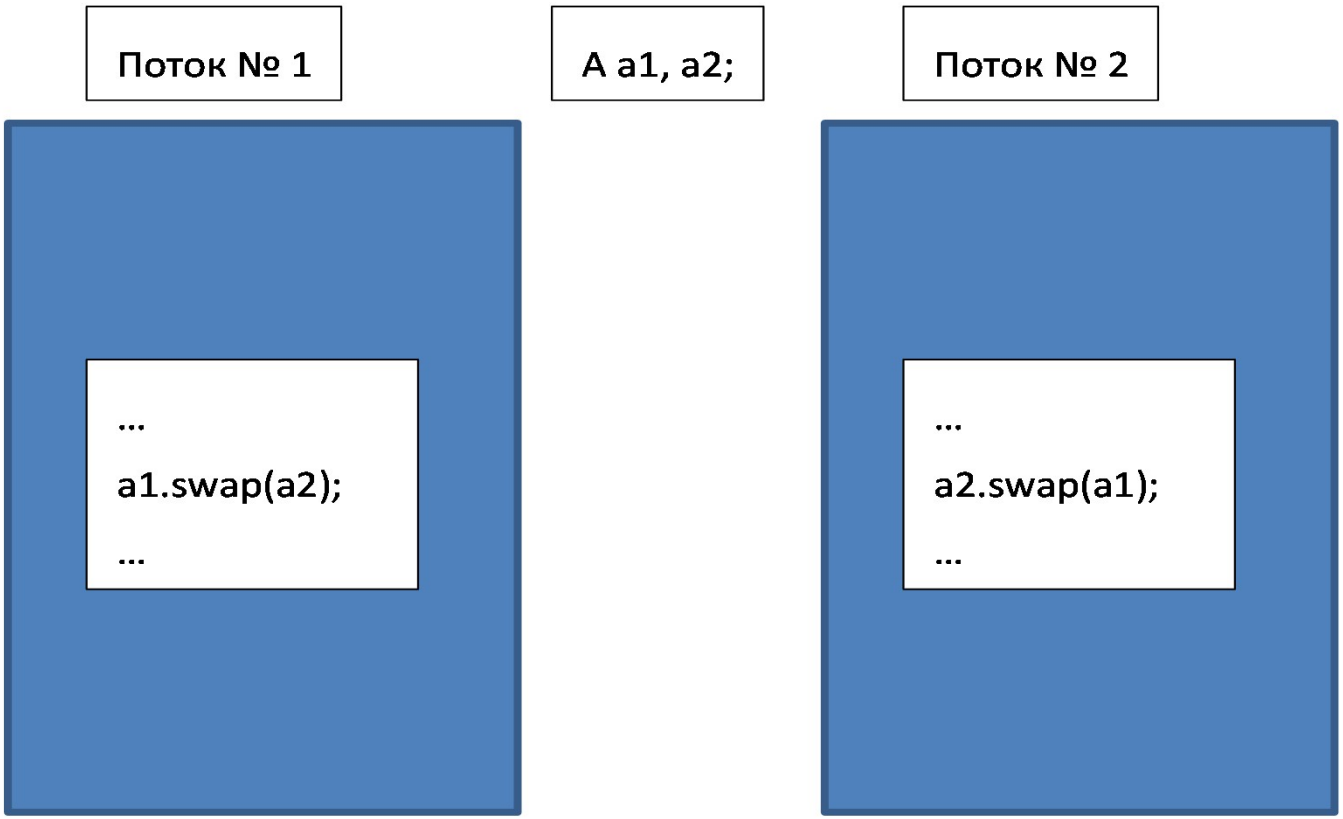
Пример использования lock(), unlock(), try_lock()

```
std::mutex m;
if (m.try_lock()) {std::cout << “Mutex was free. I’ve captured mutex" << std::endl;}
else {std::cout << “Somebody else already captured mutex…" << std::endl;}
//1.
        m.lock(); //???
//2.
        m.unlock();
        m.lock(); //???
```

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 обменивающихся данными объекта!
};
```

Проблема!



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
        }
};
```

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();
  }
}
```

Неблокирующая попытка захвата: std::try_lock()

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

//возвращаемое значение

- -1 – OK
- 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 в деструкторе только 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 {};

constexpr std::try_to_lock_t try_to_lock {};

constexpr std::adopt_lock_t adopt_lock {};

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); – m.**lock**() =>
  если мьютекс уже захвачен???
  lock_guard(mutex_type& m, std::adopt_lock_t t); -
  “оборачивает” мьютекс без вызова m.lock()
- в деструкторе – **unlock**()

Специфика:

- lock_guard(const lock_guard&) = delete;

- lock_guard&
  operator=(const lock_guard&) = delete;

Зачем нужны обертки?

- для неаккуратных программистов, которые забывают вызвать unlock()
- если сгенерировано исключение

Когда аккуратность программиста не спасает:

//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?
}
```

???

```
{
        std::mutex m;
        m.lock();
        std::lock_guard<std::mutex>  l(m);
}
{
        std::mutex m;
        std::lock_guard<std::mutex>  l(m);
        m.unlock();
}
```

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 со свободным или захваченным мьютексом???
        std::lock_guard l(m, std::adopt_lock);
        //работаем под защитой
} // ???
```

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

???

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::mutex m;
  m.lock();
  std::lock_guard<std::mutex>  l(m, std::adopt_lock );
}
{
  std::mutex m;
  m.lock();
  std::lock_guard<std::mutex>  l(m , std::adopt_lock );
  m.unlock();
}
```

STD::UNIQUE_LOCK

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)

=> чуть 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) noexcept; - 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
}
```

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
} ???
```

Отличие ???

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

Проблемы ???

```
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(this->m, other.m);
            //обмен
} ???
```

Решение:

```
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:

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**()

Важно!

std::timed_mutex **tm**; //захват мьютекса

bool b = tm.try_lock(); // “the behavior is undefined”

Пример

```
void threadFunc(std::timed_mutex& m)
{
   bool b = m.try_lock_for(std::chrono::seconds (1));
   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::timed_mutex m;
std::lock_guard<std::timed_mutex> l1(m); //ОК, возможное применение –
        в разных потоках используются разные обертки для одного и того же мьютекса
std::lock_guard<std::timed_mutex> l2(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
}   // ???
```

Важно!

std::**unique_lock**<std::timed_mutex> ul2(**tm**);

//захват мьютекса

bool b = ul2.try_lock(); // “the behavior is undefined”

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); //ОК
//ul1.try_lock_for(1s); //ошибка - у mutex нет try_lock_for

std::timed_mutex tm;
std::unique_lock<std::timed_mutex> ul2(tm, std:: std::defer_lock );
bool b = ul2.try_lock_for(1s);
```

STD::RECURSIVE_MUTEX

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>

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!

=> **srw - 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; //читатели!!!
                m.unlock_shared();
                return a;
        }
        void Set(int a) {//только один поток может модифицировать
                m.lock(); //???
                m_a=a;
                m.unlock();
        }
};
```

Проблема rw мьютекса – «голодание писателя»

"Optimized C++" Гантерота:

*По моему опыту, мьютексы “читатель/писатель” ведут к **голоданию потока писателя**, если только чтение не выполняется достаточно редко; но в этом случае величина оптимизации чтения/записи оказывается незначительной. Как и в случае с рекурсивными мьютексами, разработчики должны иметь **веские** основания для использования этого более сложного мьютекса и в общем случае выбирать более простой и более предсказуемый мьютекс.*

Продолжение простого примера:

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

Продолжение простого примера:

```
class Shared {
        int m_a=0;
        mutable std::shared_mutex m;
public:
 Shared(const Shared& other){
                other.m.lock_shared(); //если мьютекс заблокирован
                                       для записи, то засыпаем
                m_a=other.m_a;
                other.m.unlock_shared();

         }
};
```

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, m.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&);

Реализация operator=

```
Shared& Shared::operator= (const Shared& other)
{
if(this != &other){
//this - эксклюзивные права на запись
  std::unique_lock< std::shared_timed_mutex >
                                  this_ul(this->m, std::defer_lock);
//остальным (other) – разделяемые права на чтение
  std::shared_lock< std::shared_timed_mutex >
                                  other_sl(other.m, std::defer_lock);
  std::lock(this_ul, other_sl);

//выполняем присваивание:
  this->sharedData = other.sharedData;
}
  return *this;
} //???
```

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);

???

Переписываем
void A::swap(A& other);

```
void A::swap(A& other){
   std::scoped_lock lock{this->m, other.m};

    //обмен

} ???
```

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).

C++20

STD::SEMAPHORE

М.Полубенцева

Специфика:

- семафор ведет подсчет потоков, имеющих право одновременно выполнять защищаемый код => внутренний счетчик
- при попытке занять семафор – aquire():
  - если счетчик == 0, поток блокируется
  - если счетчик > 0, счетчик декрементируется – поток продолжает выполнение
- при освобождении семафора release() счетчик инкрементируется
- в отличие от мьютекса acquire() может вызвать один поток, а release() другой!

<semaphore>

Варианты:

- counting semaphore

```
template<std::ptrdiff_t LeastMaxValue = /* implementation-defined */>
        class counting_semaphore;
```

- binary semaphore

```
using binary_semaphore =
        std::counting_semaphore<1>;
```

Инициализация семафора:

- constexpr explicit **counting_semaphore**(std::ptrdiff_t desired);
  ,где desired – это начальное значение внутреннего счетчика

- counting_semaphore(const counting_semaphore&) = delete;

Замечание: operator=, move-операции deleted!

Захват семафора:

- void acquire();
  - если счетчик >0, счетчик-- , поток продолжает выполнение
  - если счетчик ==0, поток блокируется
- bool try_acquire() noexcept; - неблокирующая попытка захвата
- try_acquire_for(<duration>) - неблокирующая попытка с тайм-аутом
- try_acquire_until(<time_point>)

Освобождение семафора:

void release(std::ptrdiff_t update = 1);

Увеличение внутреннего счетчика на значение update.

Пример:

```
std::counting_semaphore<10> sema(3);

void f(char c, size_t n) {
        sema.aquire();
        for(size_t i=0; i<n; i++){std::cout<<c;}
        sema.release();
}

int main(){
        std::vector<thread> threads;
        for(char c = ‘a’; c<‘z’; c++) {threads.emplace_back(&f, c, 10);}
        for(auto& t:threads){t.join();}
}
```

Параллельные алгоритмы C++17

<std::algorithm>

М.Полубенцева

Параллельные алгоритмы STL

умеют преобразовать существующий последовательный код в параллельный с учетом:

- многопоточности (выполнение на нескольких ядрах)
- и векторизации (выполнение с задействованием векторных регистров процессора)

C++17 – политика выполнения <execution>

- - это набор C++ классов для задания типа параметра обобщенного алгоритма (для перегрузки)
- Для удобства программиста стандарт также определяет по одному объекту каждого такого класса, который может быть передан в качестве параметра при вызове алгоритмов.

Задание политики выполнения: <execution>

- **std::execution::sequenced_policy (seq)** – последовательно в вызывающем потоке
- **std::execution::parallel_policy (par)** – может выполняться параллельно. При этом пользовательские функции, вызываемые во время работы алгоритма, не должны порождать «гонку данных»
- **std::execution::parallel_unsequenced_policy (par_unseq)** – может выполняться параллельно (см. выше) и векторно
- **C++20 std::execution::unsequenced_policy** – в одном потоке с использованием векторных инструкций процессора

Замечание (важное!)

При параллельном запуске алгоритма нужно учитывать:

- факторы, зависящие от аппаратуры (эффекты кеша - перегрузка)
- «длина» распараллеливаемых данных
- количество запускаемых потоков
- ???

Иначе вместо выигрыша, можно получить проигрыш по сравнению с последовательным выполнением

Параллельные алгоритмы C++17 (специфика VS???)

- adjacent_difference() adjacent_find()
- all_of() any_of()
- count() count_if()
- equal()
- exclusive_scan()
- find() find_end() find_first_of() find_if()
- for_each() for_each_n()
- inclusive_scan()
- mismatch()
- none_of()
- reduce()
- remove() remove_if()
- search() search_n()
- sort() stable_sort()
- transform() transform_exclusive_scan() transform_inclusive_scan() transform_reduce()

Пример:

```
std::vector<int> v; //большой!!!
//сформировали значения
std::vector<int> res(v.size());
std::transform(std::execution::par_unseq, v.begin(), v.end(),
                    res.begin(), std::negate<int>());
```

Intel® Parallel Studio XE 2018

- Добавлены еще политики выполнения
- Parallel STL позволяет разработчикам сфокусироваться на выражении параллелизма в их приложениях, не заботясь о
  - низкоуровневом управлении потоками
  - и векторными регистрами.

М.Полубенцева