Стандарт C++ 1998 года не признавал существования потоков, поэтому результаты работы различных языковых конструкций описывались в терминах последовательной абстрактной машины. Более того, модель памяти не была формально определена, поэтому без поддержки со стороны расширений стандарта C++ 1998 года писать многопоточные приложения вообще было невозможно.

Разумеется, производители компиляторов вправе добавлять в язык любые расширения, а наличие различных API для поддержки многопоточности в языке C, например, в стандарте POSIX C Standard и в Microsoft Windows API, заставило многих производителей компиляторов C++ поддержать многопоточность с помощью платформенных расширений. Как правило, эта поддержка ограничивается разрешением использовать соответствующий платформе C API с гарантией, что библиотека времени исполнения C++ (в частности, механизм обработки исключений) будет корректно работать при наличии нескольких потоков. Хотя лишь очень немногие производители компиляторов предложили формальную модель памяти с поддержкой многопоточности, практическое поведение компиляторов и процессоров оказалось достаточно приемлемым для создания большого числа многопоточных программ на C++.

Не удовлетворившись использованием платформенно-зависимых C API для работы с многопоточностью, программисты на C++ пожелали, чтобы в используемых ими библиотеках классов были реализованы объектно-ориентированные средства для написания многопоточных программ.

Все изменилось с выходом стандарта C++11. Мало того, что в нем определена совершенно новая модель памяти с поддержкой многопоточности, так еще и в стандартную библиотеку C++ включены классы для управления потоками, защиты разделяемых данных, синхронизации операций между потоками и низкоуровневых атомарных операций.

В основу новой библиотеки многопоточности для C++ положен опыт, накопленный за время использования вышеупомянутых библиотек классов.

Поддержка параллелизма – лишь одна из новаций в стандарте C++. В сам язык тоже внесено много изменений, призванных упростить жизнь программистам. Хотя, вообще говоря, сами по себе они не являются предметом нашего обсуждения, некоторые оказывают прямое влияние на библиотеку многопоточности и способы ее использования. Прямая языковая поддержка атомарных операций позволяет писать эффективный код с четко определенной семантикой, не прибегая к языку ассемблера для конкретной платформы. Это крайне удобно для тех, кто пытается создавать эффективный и переносимый код, – мало того, что компилятор берет на себя заботу об особенностях платформы, так еще и оптимизатор можно написать так, что он будет учитывать семантику операций и, стало быть, лучше оптимизировать программу в целом.

Одна из проблем, с которыми сталкиваются разработчики высокопроизводительных приложений при использовании языка C++ вообще и классов, обертывающих низкоуровневые средства, типа тех, что включены в стандартную библиотеку C++ Thread Library, в частности, – это эффективность. Если вас интересует достижение максимальной производительности, то необходимо понимать, что использование любых высокоуровневых механизмов вместо обертываемых ими низкоуровневых средств влечет за собой некоторые издержки. Эти издержки называются платой за абстрагирование.

Комитет по стандартизации C++ прекрасно понимал это, когда проектировал стандартную библиотеку C++ вообще и стандартную библиотеку многопоточности для C++ в частности. Среди целей проектирования была и такая: выигрыш от использования низкоуровневых средств по сравнению с высокоуровневой оберткой (если такая предоставляется) должен быть ничтожен или отсутствовать вовсе. Поэтому библиотека спроектирована так, чтобы ее можно было эффективно реализовать (с очень небольшой платой за абстрагирование) на большинстве популярных платформ.
Комитет по стандартизации C++ поставил и другую цель – обеспечить достаточное количество низкоуровневых средств для желающих работать на уровне «железа», чтобы выдавить из него все, что возможно. Поэтому наряду с новой моделью памяти включена полная библиотека атомарных операций для прямого управления на уровне битов и байтов, а также средства межпоточной синхронизации и обеспечения видимости любых изменений. Атомарные типы и соответствующие операции теперь можно использовать во многих местах, где раньше разработчики были вынуждены опускаться на уровень языка ассемблера для конкретной платформы. Таким образом, код с применением новых стандартных типов и операций получается более переносимым и удобным для сопровождения.
Стандартная библиотека C++ также предлагает высокоуровневые абстракции и средства, позволяющие писать многопоточный код проще и с меньшим количеством ошибок. Некоторые из них несколько снижают производительность из-за необходимости выполнять дополнительный код. Однако эти накладные расходы не обязательно означают высокую плату за абстрагирование: в общем случае цена не выше, чем пришлось бы заплатить при написании эквивалентной функциональности вручную, и к тому же компилятор вполне может встроить значительную часть дополнительного кода.
В некоторых случаях высокоуровневые средства обеспечивают большую функциональность, чем необходимо для конкретной задачи. Как правило, это не страшно: вы не платите за то, чем не пользуетесь. Редко, но бывает, что избыточная функциональность негативно сказывается на производительности других частей программы. Если ее стоимость слишком высока, а производительность имеет первостепенное значение, то, быть может, имеет смысл вручную запрограммировать необходимую функциональность, пользуясь низкоуровневыми средствами. Но в подавляющем большинстве случаев дополнительная сложность и возможность внести ошибки намного перевешивают небольшой выигрыш в производительности. Даже если профилирование показывает, что средства стандартной библиотеки C++ действительно являются узким местом, не исключено, что проблема в неудачном дизайне приложения, а не в плохой реализации библиотеки. Например, когда слишком много потоков конкурируют за один мьютекс, производительность упадет – и сильно. Но лучше не пытаться чуть-чуть ускорить операции с мьютексами, а изменить структуру приложения, так чтобы снизить конкуренцию.
В тех крайне редких случаях, когда стандартная библиотека не обеспечивает необходимой производительности или поведения, может возникнуть необходимость в использовании платформенно-зависимых средств.
Хотя библиотека многопоточности для C++ содержит достаточно полный набор средств для создания многопоточных программ, на любой платформе имеются специальные средства, помимо включенных в библиотеку. Чтобы можно было получить доступ к этим средствам, не отказываясь от использования стандартной библиотеки, типы,

имеющиеся в библиотеки многопоточности, иногда содержат функцию-член native_handle() , которая позволяет работать на уровне платформенного API. По природе своей любые операции, выполняемые с помощью функции native_handle(), зависят от платформы и потому ни нами, ни в самой стандартной библиотеке C++ не рассматриваются.

Разумеется, перед тем задумываться о применении платформенно-зависимых средств, стоит как следует разобраться в том, что предлагает стандартная библиотека, поэтому начнем с примера.

```
#include <iostream>
#include <thread>
void hello()
{
      std::cout<<”Hello, parallel world!\n”;
}
int main()
{
      std::thread t(hello);
      t.join();
}
```

Код вывода сообщения перемещен в отдельную функцию. Это объясняется тем, что в каждом потоке должна быть начальная функция, в которой начинается исполнение потока. Для первого потока в приложении таковой является main(), а для всех остальных задаётся в конструкторе объекта std::thread. В данном случае в качестве начальной функции объекта типа std::thread, названного t, выступает функция hello().

Есть и еще одно отличие – вместо того чтобы сразу писать на стандартный вывод или вызывать hello() из main(), эта программа запускает новый поток, так что теперь общее число потоков равно двум: главный, с начальной функцией main(), и дополнительный, начинающий работу в функции hello().

После запуска нового потока начальный поток продолжает работать. Если бы он не ждал завершения нового потока, то просто дошел бы до конца main(), после чего исполнение программы закончилась бы – быть может, еще до того, как у нового потока появился шанс начать работу. Чтобы предотвратить такое развитие событие, мы добавили обращение к функции join(); это заставляет вызывающий поток (main()) ждать завершения потока, ассоциированного с объектом std::thread, – в данном случае t.

Если вам показалось, что для элементарного вывода сообщения на стандартный вывод работы слишком много, то так оно и есть, – обычно для решения такой простой задачи не имеет смысла создавать несколько потоков, особенно если главному потоку в это время нечего делать.

Для запуска потока следует сконструировать объект std::thread , который определяет, какая задача будет исполняться в потоке. В простейшем случае задача представляет собой обычную функцию без параметров, возвращающую void. Эта функция работает в своем потоке, пока не вернет управление, и в этом момент поток завершается. С другой стороны, в роли задачи может выступать объект-функция, который принимает дополнительные параметры и выполняет ряд независимых операций, информацию о которых получает во время работы от той или иной системы передачи сообщений. И останавливается такой поток, когда получит соответствующий сигнал, опять же с помощью системы передачи сообщений. Вне зависимости от того, что поток будет делать и откуда он запускается, сам запуск потока в стандартном C++ всегда сводится к конструированию объекта std::thread:

```
void do_some_work();
std::thread my_thread(do_some_work);
```
Как видите, все просто. Разумеется, как и во многих других случаях в стандартной библиотеке C++, класс std::thread работает с любым типом, допускающим вызов (Callable), поэтому конструктору std::thread можно передать экземпляр класса, в котором определен оператор вызова:
```
class background_task
{
public:
      void operator()() const
      {
            do_something();
            do_something_else();
      }
};
background_task f;
std::thread my_thread(f);
```
После запуска потока необходимо явно решить, ждать его завершения (присоединившись к нему) или предоставить собственной судьбе (отсоединив его). Если это решение не будет принято к моменту уничтожения объекта std::thread, то программа завершится (деструктор std::thread вызовет функцию std::terminate()). Поэтому вы обязаны гарантировать, что поток корректно присоединен либо отсоединен, даже если возможны исключения. Это решение следует принять именно до уничтожения объекта std::thread, к самому потоку оно не имеет отношения. Поток вполне может завершиться задолго до того, как программа присоединится к нему или отсоединит его. А отсоединенный поток может продолжать работу и после уничтожения объекта std::thread.

Если вы не хотите дожидаться завершения потока, то должны гарантировать, что данные, к которым поток обращается, остаются действительными до тех пор, пока они могут ему понадобиться. Эта проблема не нова – даже в однопоточной программа доступ к уже уничтоженному объекту считается неопределенным поведением, но при использовании потоков есть больше шансов столкнуться с проблемами, обусловленными временем жизни. Например, такая проблема возникает, если функция потока хранит указатели или ссылки на локальные переменные, и поток еще не завершился, когда произошел выход из области видимости, где эти переменные определены.
```
struct func
{
      int& i;
      func(int& i_) : i(i_){}
      void operator()()
      {
            for(unsigned j=0;j<1000000;++j)
            {
                  do_something(i);
            }
      }
};
void oops()
{
      int some_local_state=0;
      func my_func(some_local_state);
      std::thread my_thread(my_func);
      my_thread.detach();
}
```

В данном случае вполне возможно, что новый поток, ассоциированный с объектом my_thread, будет еще работать, когда функция oops вернет управление, поскольку мы явно решили не дожидаться его завершения, вызвав detach(). А если поток действительно работает, то при следующем вызове do_something(i) произойдет обращение к уже уничтоженной переменной. Точно так же происходит в обычном однопоточном коде – сохранять указатель или ссылку на локальную переменную после выхода из функции всегда плохо, – но в многопоточном коде такую ошибку сделать проще, потому что не сразу видно, что произошло.

Один из распространенных способов разрешить такую ситуацию – сделать функцию потока замкнутой, то есть *копировать* в поток данные, а не разделять их. Если функция потока реализована в виде вызываемого объекта, то сам этот объект копируется в поток, поэтому исходный объект можно сразу же уничтожить. Однако по-прежнему необходимо следить за тем, чтобы объект не содержал ссылок или указателей. В частности, не стоит создавать внутри функции поток, имеющий доступ к локальным переменным этой функции, если нет гарантии, что поток завершится до выхода из функции.

Есть и другой способ – явно гарантировать, что поток завершит исполнение до выхода из функции, *присоединившись* к нему.

Чтобы дождаться завершения потока, следует вызвать функцию join() ассоциированного объекта std::thread. В листинге мы можем заменить вызов my_thread.detach() перед закрывающей скобкой тела функции вызовом my_thread.join(), и тем самым гарантировать, что поток завершится до выхода из функции, то есть раньше, чем будут уничтожены локальные переменные. В данном случае это означает, что запускать функцию в отдельном потоке не имело смысла, так как первый поток в это время ничего не делает, но в реальной программе исходный поток мог бы либо сам делать что-то полезное, либо запустить несколько потоков параллельно, а потом дождаться их всех.

Функция join() дает очень простую и прямолинейную альтернативу – либо мы ждем завершения потока, либо нет. Если необходим более точный контроль над ожиданием потока, например, если необходимо проверить, завершился ли поток, или ждать только ограниченное время, то следует прибегнуть к другим механизмам, таким, как условные переменные и будущие результаты. Кроме того, при вызове join() очищается вся ассоциированная с потоком память, так что объект std::thread более не связан с завершившимся потоком – он вообще не связан ни с каким потоком. Это значит, что для каждого потока вызвать функцию join() можно только один раз; после первого вызова объект std::thread уже не допускает присоединения, и функция joinable() возвращает false.

Мы отметили, что функцию join() или detach() необходимо вызвать до уничтожения объекта std::thread. Если вы хотите отсоединить поток, то обычно достаточно вызвать detach() сразу после его запуска, так что здесь проблемы не возникает. Но если вы собираетесь дождаться завершения потока, то надо тщательно выбирать место, куда поместить вызов join(). Важно, чтобы из-за исключения, произошедшего между запуском потока и вызовом join(), не оказалось, что обращение к join() вообще окажется пропущенным.

Чтобы приложение не завершилось аварийно при возникновении исключения, необходимо решить, что делать в этом случае. Вообще говоря, если вы намеревались

вызвать функцию join() при нормальном выполнении программы, то следует вызывать ее и в случае исключения, чтобы избежать проблем, связанных с временем жизни.

```
struct func;

void f()
{
      int some_local_state=0;
      func my_func(some_local_state)
      std::thread t(my_func);
      try
      {
            do_something_in_current_thread();
      }
      catch(...)
      {
            t.join();
            throw;
      }
      t.join();
}
```

Блок try/catch используется для того, чтобы поток, имеющий доступ к локальному состоянию, гарантированно завершился до выхода из функции вне зависимости от того, происходит выход нормально или вследствие исключения. Записывать блоки try/catch очень долго и при этом легко допустить ошибку, поэтому такой способ не идеален. Если необходимо гарантировать, что поток завершается до выхода из функции – потому ли, что он хранит ссылки на локальные переменные, или по какой-то иной причине – то важно обеспечить это на всех возможных путях выхода, как нормальных, так и в результате исключения, и хотелось бы иметь для этого простой и лаконичный механизм.

Один из способов решить эту задачу – воспользоваться стандартной идиомой *захват ресурса есть инициализация* (RAII) и написать класс, который вызывает join() в деструкторе. Обратите внимание, насколько проще стала функция f().

```
class thread_guard
{
      std::thread& t;
public:
      explicit thread_guard(std::thread& t_) : t(t_)
      {}
      ~thread_guard()
      {
            if(t.joinable())
            {
                  t.join();
            }
      }
      thread_guard(thread_guard const&)=delete;
      thread_guard& operator=(thread_guard const&)=delete;
};
struct func;
void f()
{
      int some_local_state;
      std::thread t(func(some_local_state));
      thread_guard g(t);
      do_something_in_current_thread();
}
```

Когда текущий поток доходит до конца f, локальные объекты уничтожаются в порядке, обратном тому, в котором были сконструированы. Следовательно, сначала уничтожается объект g типа thread_guard, и в его деструкторе происходит присоединение к потоку. Это справедливо даже в том случае, когда выход из функции f произошел в результате исключения внутри функции do_something_in_current_thread.
Деструктор класса thread_guard сначала проверяет, что объект std::thread находится в состоянии joinable() и, лишь если это так, вызывает join(). Это существенно, потому что функцию join() можно вызывать только один раз для данного потока, так что если он уже присоединился, то делать это вторично было бы ошибкой.
Копирующий конструктор и копирующий оператор присваивания помечены признаком =delete, чтобы компилятор не генерировал их автоматически: копирование или присваивание такого объекта таит в себе опасность, поскольку время жизни копии может оказаться дольше, чем время жизни присоединяемого потока. Но раз эти функции объявлены как «удаленные», то любая попытка скопировать объект типа thread_guard приведет к ошибке компиляции.
Если ждать завершения потока не требуется, то от проблемы безопасности относительно исключений можно вообще уйти, отсоединив поток. Тем самым связь потока с объектом std::thread разрывается, и при уничтожении объекта std::thread функция std::terminate() не будет вызвана. Но отсоединенный поток по-прежнему работает – в фоновом режиме.
Вызов функции-члена detach() объекта std::thread оставляет поток работать в фоновом режиме, без прямых способов коммуникации с ним. Теперь ждать завершения потока не получится – после того, как поток отсоединен, уже невозможно получить ссылающийся на него объект std::thread, для которого можно было бы вызвать join(). Отсоединенные потоки действительно работают в фоне: отныне ими владеет и управляет библиотека времени выполнения C++, которая обеспечит корректное освобождение связанных с потоком ресурсов при его завершении.
Отсоединенные потоки часто называют потоками-демонами по аналогии с процессами-демонами в UNIX, то есть с процессами, работающими в фоновом режиме и не имеющими явного интерфейса с пользователем. Обычно такие потоки работают в течение длительного времени, в том числе на протяжении всего времени жизни приложения. Они, например, могут следить за состоянием файловой системы, удалять неиспользуемые записи из кэша или оптимизировать структуры данных. С другой стороны, иногда отсоединенный поток применяется, когда существует какой-то другой способ узнать о его завершении или в случае, когда нужно запустить задачу и «забыть» о ней.
Мы уже видели, что для отсоединения потока следует вызвать функцию-член detach() объекта std::thread. После возврата из этой функции объект std::thread уже не связан ни с каким потоком, и потому присоединиться к нему невозможно.

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

Разумеется, чтобы отсоединить поток от объекта std::thread, поток должен существовать: нельзя вызвать detach() для объекта std::thread, с которым не связан никакой поток. Это то же самое требование, которое предъявляется к функции join(), поэтому и проверяется оно точно так же – вызывать t.detach() для объекта t типа std::thread можно только тогда, когда t.joinable() возвращает true.

Возьмем в качестве примера текстовый редактор, который умеет редактировать сразу несколько документов. Реализовать его можно разными способами – как на уровне пользовательского интерфейса, так и с точки зрения внутренней организации. В настоящее время все чаще для этой цели используют несколько окон верхнего уровня, по одному для каждого редактируемого документа. Хотя эти окна выглядят совершенно независимыми, в частности, у каждого есть свое меню и все прочее, на самом деле они существуют внутри единственного экземпляра приложения. Один из подходов к внутренней организации программы заключается в том, чтобы запускать каждое окно в отдельном потоке: каждый такой поток исполняет один и тот же код, но с разными данными, описывающими редактируемый документ и соответствующее ему окно. Таким образом, чтобы открыть еще один документ, необходимо создать новый поток. Потоку, обрабатывающему запрос, нет дела до того, когда созданный им поток завершится, потому что он работает над другим, независимым документом. Вот типичная ситуация, когда имеет смысл запускать отсоединенный поток.

```
void edit_document(std::string const& filename)
{
      open_document_and_display_gui(filename);
      while(!done_editing())
      {
            user_command cmd=get_user_input();
            if(cmd.type==open_new_document)
            {
                  std::string const new_name= get_filename_from_user();
                  std::thread t(edit_document,new_name);
                  t.detach();
            }
            else
            {
            process_user_input(cmd);
            }
      }
}
```

Когда пользователь открывает новый документ, мы спрашиваем, какой документ открыть, затем запускаем поток, в котором этот документ открывается, и отсоединяем его. Поскольку новый поток делает то же самое, что текущий, только с другим файлом, то мы можем использовать ту же функцию (edit_document), передав ей в качестве аргумента имя только что выбранного файла.

Этот пример демонстрирует также, почему бывает полезно передавать аргументы функции потока: мы передаем конструктору объекта std::thread не только имя функции, но и ее параметр – имя файла. Существуют другие способы добиться той же цели, но библиотека предлагает и такой простой механизм.

По существу передача аргументов вызываемому объекту или функции сводится просто к передаче дополнительных аргументов конструктору std::thread. Однако важно иметь в виду, что по умолчанию эти аргументы копируются в память объекта, где они доступны вновь созданному потоку, причем так происходит даже в том случае, когда функция ожидает на месте соответствующего параметра ссылку.

```
void f(int i,std::string const& s);
std::thread t(f,3,”hello”);
```

Здесь создается новый ассоциированный с объектом t поток, в котором вызывается функция f(3,”hello”). Отметим, что функция f принимает в качестве второго параметра объект типа std::string, но мы передаем строковый литерал char const*, который преобразуется к типу std::string уже в контексте нового потока.

При передаче потокам указателей на локальные переменные и отсоединении их сразу после создания всегда есть шанс того, что выход из основного потока произойдет до того, чем аргумент будет преобразован к типу std::string в новом потоке, и в таком случае поведение будет неопределенным. Чтобы избежать подобного, преобразование необходимо производить до передачи аргумента конструктору std::thread.
Возможен и обратный сценарий: копируется весь объект, а вы хотели бы получить ссылку. Такое бывает, когда поток обновляет структуру данных, переданную по ссылке. Функция потока может ожидать, что её параметр будет передан по ссылке, но конструктор std::thread не знает об этом: он не в курсе того, каковы типы аргументов, ожидаемых функцией, и просто слепо копирует переданные значения. Поэтому функции, обновляющей структуру, будет передана ссылка на её внутреннюю копию, а не на сам объект. Следовательно, по завершении потока от обновлений ничего не останется, так как внутренние копии переданных аргументов уничтожаются, и функция конечной обработки получит не обновленные данные, а исходный объект data.
Решение очевидно для знакомых с механизмом std::bind: нужно обернуть аргументы, которые должны быть ссылками, объектом std::ref . В данном случае, если мы напишем std::thread t(update_data_for_widget,w,**std::ref(data)**);
то функции обновления структуры будет правильно передана ссылка на неё, а не её копия.
Предположим, что требуется написать функцию для создания потока, который должен работать в фоновом режиме, но при этом мы не хотим ждать его завершения, а хотим, чтобы владение новым потоком было передано вызывающей функции. Или требуется сделать обратное – создать поток и передать владение им некоторой функции, которая будет ждать его завершения. В обоих случаях требуется передать владение из одного места в другое.
Именно здесь и оказывается полезной поддержка классом std::thread семантики перемещения. В стандартной библиотеке C++ есть много типов, владеющих ресурсами, например std::ifstream и std::unique_ptr, которые являются перемещаемыми, но не копируемыми, и один из них – std::thread. Это означает, что владение потоком можно передавать от одного экземпляра std::thread другому.

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```

Поддержка операции перемещения в классе std::thread означает, что владение можно легко передать при возврате из функции.

```
std::thread f()
{
          void some_function();
return std::thread(some_function);
}
std::thread g()
{
     void some_other_function(int);
     std::thread t(some_other_function,42);
     return t;
}
```

Аналогично, если требуется передать владение внутрь функции, то достаточно, чтобы она принимала экземпляр std::thread по значению в качестве одного из параметров, например:

```
void f(std::thread t);
void g()
{
      void some_function();
      f(std::thread(some_function));
      std::thread t(some_function);
      f(std::move(t));
}
```

Задание количества потоков во время выполнения: в стандартной библиотеке C++ есть функция std::thread::hardware_concurrency(), которая поможет нам решить эту задачу. Она возвращает число потоков, которые могут работать по-настоящему параллельно. В многоядерной системе это может быть, например, количество процессорных ядер. Возвращаемое значение – всего лишь оценка; более того, функция может возвращать 0, если получить требуемую информацию невозможно. Однако эту оценку можно с пользой применить для разбиения задачи на несколько потоков.

Идентификатор потока имеет тип std::thread::id, и получить его можно двумя способами. Во-первых, идентификатор потока, связанного с объектом std::thread, возвращает функция-член get_id()этого объекта. Если с объектом std::thread не связан никакой поток, то get_id() возвращает сконструированный по умолчанию объект типа std::thread::id, что следует интерпретировать как «не поток». Идентификатор текущего потока можно получить также, обратившись к функции std::this_thread::get_id(), которая также определена в заголовке <thread>.

Объекты типа std::thread::id можно без ограничений копировать и сравнивать, в противном случае они вряд ли могли бы играть роль идентификаторов. Если два объекта типа std::thread::id равны, то либо они представляют один и тот же поток, либо оба содержат значение «не поток». Если же два таких объекта не равны, то либо они представляют разные потоки, либо один представляет поток, а другой содержит значение «не поток».

Библиотека Thread Library не ограничивается сравнением идентификаторов потоков на равенство, для объектов типа std::thread::id определен полный спектр операторов сравнения, то есть на множестве идентификаторов потоков задан полный порядок. Это позволяет использовать их в качестве ключей ассоциативных контейнеров, сортировать и сравнивать любым интересующим программиста способом. Поскольку операторы сравнения определяют полную упорядоченность различных значений типа std::thread::id, то их поведение интуитивно очевидно: если a<b и b<c, то a<c и так далее. В стандартной библиотеке имеется также класс std::hash<std::thread::id>, поэтому значения типа std::thread::id можно использовать и в качестве ключей новых неупорядоченных ассоциативных контейнеров.

Объекты std::thread::id часто применяются для того, чтобы проверить, должен ли поток выполнить некоторую операцию. Например, если потоки используются для разбиения задач, то начальный поток, который запускал все остальные, может вести себя несколько иначе, чем прочие. В таком случае этот поток мог бы сохранить значение std::this_thread::get_id() перед тем, как запускать другие потоки, а затем в основной части алгоритма (общей для всех потоков) сравнить собственный идентификатор с сохраненным значением. Или можно было бы сохранить std::thread::id текущего потока

в некоторой структуре данных в ходе выполнения какой-то операции. В дальнейшем при операциях с той же структурой данных можно было сравнить сохраненный идентификатор с идентификатором потока, выполняющего операцию, и решить, какие операции разрешены или необходимы.

Одно из основных достоинств применения потоков для реализации параллелизма – возможность легко и беспрепятственно разделять между ними данные, поэтому, уже зная, как создавать потоки и управлять ими, мы обратимся к вопросам, связанным с разделением данных. Если потоки разделяют какие-то данные, то необходимы правила, регулирующие, какой поток в какой момент к каким данным может обращаться и как сообщить об изменениях другим потокам, использующим те же данные. Легкость, с которой можно разделять данные между потоками в одном процессе, может обернуться не только благословением, но проклятием. Некорректное использование разделяемых данных – одна из основных причин ошибок, связанных с параллелизмом.

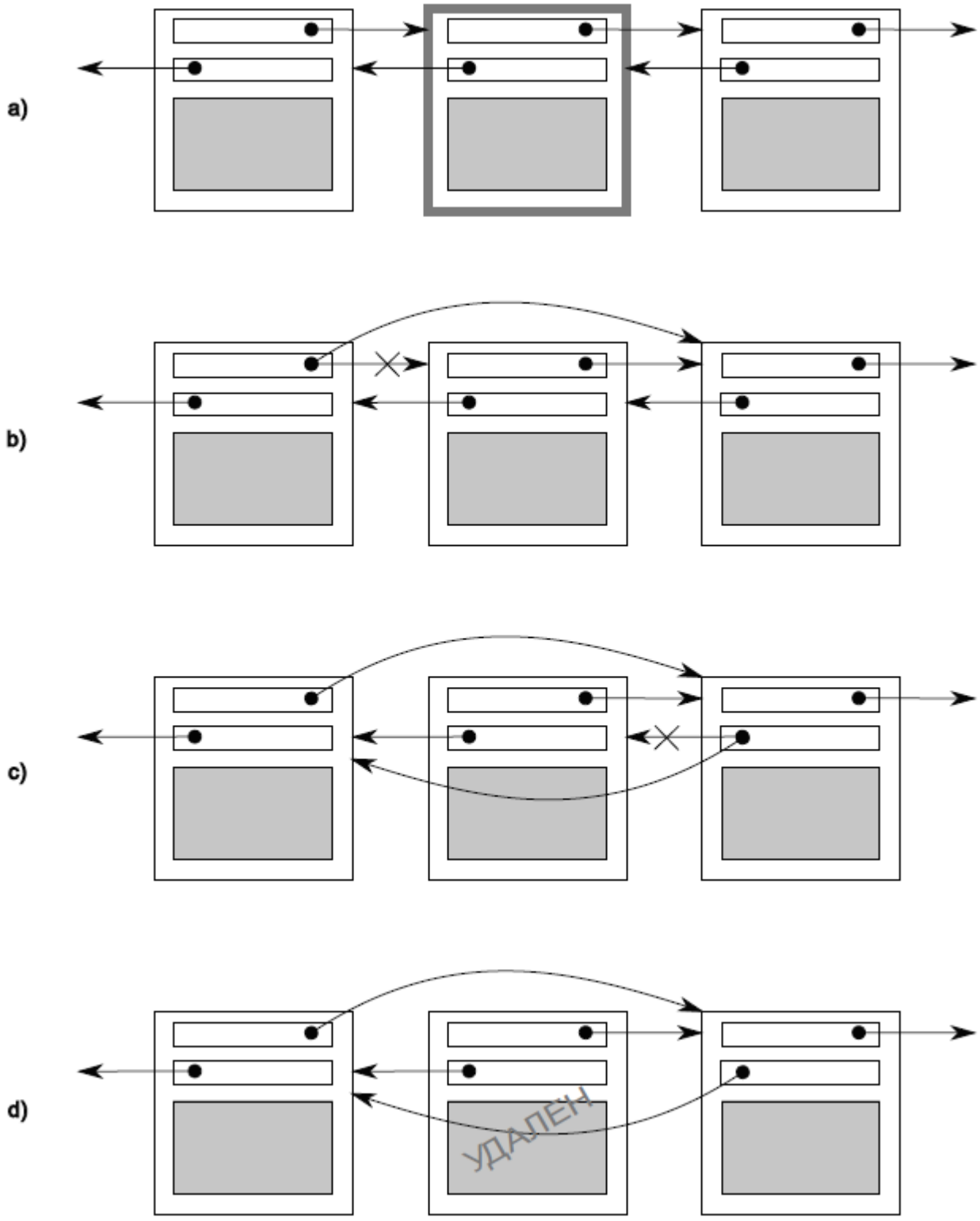
Все проблемы разделения данных между потоками связаны с последствиями модификации данных. Если разделяемые данные только читаются, то никаких сложностей не возникает, поскольку любой поток может читать данные независимо от того, читают их в то же самое время другие потоки или нет. Но стоит одному или нескольким потокам начать модифицировать разделяемые данные, как могут возникнуть неприятности. В таком случае ответственность за правильную работу ложится на программиста.

При рассуждениях о поведении программы часто помогает понятие инварианта – утверждения о структуре данных, которое всегда должно быть истинным, например, «значение этой переменной равно числу элементов в списке». В процессе обновления инварианты часто нарушаются, особенно если структура данных сложна или обновление затрагивает несколько значений.

Рассмотрим двусвязный список, в котором каждый узел содержит указатели на следующий и предыдущий узел. Один из инвариантов формулируется так: если «указатель на следующий» в узле A указывает на узел B, то «указатель на предыдущий» в узле B указывает на узел A. Чтобы удалить узел из списка, необходимо обновить узлы по обе стороны от него, так чтобы они указывали друг на друга. После обновления одного узла инвариант оказывается нарушен и остается таковым, пока не будет обновлен узел по другую сторону. После того как обновление завершено, инвариант снова выполняется. Таким образом, шаги удаления узла из списка:

1. Найти подлежащий удалению узел (N).
2. Изменить «указатель на следующий» в узле, предшествующем N, так чтобы он указывал на узел, следующий за N.
3. Изменить «указатель на предыдущий» в узле, следующем за N, так чтобы он указывал на узел, предшествующий N.
4. Удалить узел N.

Как видите, между шагами 2 и 3 с указатели в одном направлении не согласуются с указателями в другом направлении, и инвариант нарушается.



Простейшая проблема, которая может возникнуть при модификации данных, разделяемых несколькими потоками, – нарушение инварианта. Если не предпринимать никаких мер, то в случае, когда один поток читает двусвязный список, а другой в это же время удаляет из списка узел, вполне может случиться, что читающий поток увидит список, из которого узел удален лишь частично (потому что изменен только один указатель, как на шаге 2), так что инвариант нарушен. Последствия могут быть разными – если поток читает список слева направо, то он просто пропустит удаляемый узел. Но если другой поток пытается удалить самый правый узел, показанный на рисунке, то он может навсегда повредить структуру данных, и в конце концов это приведет к аварийному завершению программы. Как бы то ни было, этот пример иллюстрирует

одну из наиболее распространенных причин ошибок в параллельном коде: состояние гонки (race condition).

Предположим, вы покупаете билеты в кино. Если кинотеатр большой, то в нем может быть несколько касс, так что в каждый момент времени билеты могут покупать несколько человек. Если кто-то покупает билет на тот же фильм, что и вы, но в другой кассе, то какие места вам достанутся, зависит от того, кто был первым. Если осталось всего несколько мест, то разница может оказаться решающей: за последние билеты возникает гонка в самом буквальном смысле. Это и есть пример состояния гонки: какие места вам достанутся (да и достанутся ли вообще), зависит от относительного порядка двух покупок.

В параллельном программировании под состоянием гонки понимается любая ситуация, исход которой зависит от относительного порядка выполнения операций в двух или более потоках – потоки конкурируют за право выполнить операции первыми. Как правило, ничего плохого в этом нет, потому что все исходы приемлемы, даже если их взаимный порядок может меняться. Например, если два потока добавляют элементы в очередь для обработки, то вообще говоря неважно, какой элемент будет добавлен первым, лишь бы не нарушались инварианты системы. Проблема возникает, когда гонка приводит к нарушению инвариантов, как в приведенном выше примере удаления из двусвязного списка. В контексте параллельного программирования состоянием гонки обычно называют именно такую проблематичную гонку – безобидные гонки не так интересны и к ошибкам не приводят. В стандарте C++ определен также термин гонка за данными (data race), означающий ситуацию, когда гонка возникает из-за одновременной модификации одного объекта; гонки за данными приводят к неопределенному поведению.

Проблематичные состояния гонки обычно возникают, когда для завершения операции необходимо модифицировать два или более элементов данных, например два связующих указателя в примере выше. Поскольку элементов несколько, то их модификация производится разными командами, и может случиться, что другой поток обратится к структуре данных в момент, когда завершилась только одна команда. Зачастую состояние гонки очень трудно обнаружить и воспроизвести, поскольку она происходит в очень коротком интервале времени, – если модификации производятся последовательными командами процессора, то вероятность возникновения проблемы при конкретном прогоне очень мала, даже если к структуре данных одновременно обращается другой поток. По мере увеличения нагрузки на систему и количества выполнений операции вероятность проблематичной последовательности выполнения возрастает. И, разумеется, почти всегда такие ошибки проявляются в самый неподходящий момент. Поскольку состояние гонки так чувствительно ко времени, оно может вообще не возникнуть при запуске приложения под отладчиком, так как отладчик влияет на хронометраж программ, пусть и незначительно.

При написании многопоточных программ гонки могут изрядно отравить жизнь – своей сложностью параллельные программы в немалой степени обязаны стараниям избежать проблематичных гонок.

Существует несколько способов борьбы с проблематичными гонками. Простейший из них – снабдить структуру данных неким защитным механизмом, который гарантирует, что только поток, выполняющий модификацию, может видеть промежуточные состояния, в которых инварианты нарушены; с точки зрения всех остальных потоков,

обращающихся к той же структуре данных, модификация либо еще не началась, либо уже завершилась.
Другой вариант – изменить дизайн структуры данных и ее инварианты, так чтобы модификация представляла собой последовательность неделимых изменений, каждое из которых сохраняет инварианты. Этот подход обычно называют программированием без блокировок (lock-free programming) и реализовать его правильно очень трудно; если вы работаете на этом уровне, то приходится учитывать нюансы модели памяти и разбираться, какие потоки потенциально могут увидеть те или иные наборы значений.
Еще один способ справиться с гонками – рассматривать изменения структуры данных как транзакцию, то есть так, как обрабатываются обновления базы данных внутри транзакции. Требуемая последовательность изменений и чтений данных сохраняется в журнале транзакций, а затем атомарно фиксируется. Если фиксация невозможна, потому что структуру данных в это время модифицирует другой поток, то транзакция перезапускается. Это решение называется программной транзакционной памятью (Software Transactional Memory – STM), но мы не будем его рассматривать, потому что в C++ нет для него поддержки.
Самый простой механизм защиты разделяемых данных из описанных в стандарте C++ – это мьютекс. Пусть у нас есть разделяемая структура данных, например, связанный список из предыдущего примера, и мы хотим защитить его от гонки и нарушения инвариантов, к которым она приводит. Было бы здорово, если бы мы могли пометить участки кода, в которых производятся обращения к этой структуре данных, взаимно исключающими, так что если один поток начинает выполнять такой участок, то все остальные потоки должны ждать, пока первый не завершит обработку данных. Тогда ни один поток, кроме выполняющего модификацию, не смог бы увидеть нарушенный инвариант.
Именно такое поведение вы получаете при использовании примитива синхронизации, который называется мьютекс (слово mutex происходит от mutual exclusion – взаимное исключение). Перед тем как обратиться к структуре данных, программа захватывает (lock) мьютекс, а по завершении операций с ней освобождает (unlock) его. Библиотека Thread Library гарантирует, что если один поток захватил некоторый мьютекс, то все остальные потоки, пытающиеся захватить тот же мьютекс, будут вынуждены ждать, пока захвативший не освободит его. В результате все потоки видят согласованное представление разделяемых данных, без нарушенных инвариантов.
Мьютексы – наиболее общий механизм защиты данных в C++, но панацеей они не являются; важно структурировать код так, чтобы защитить нужные данные, и избегать состояний гонки, внутренне присущих интерфейсам. С мьютексами связаны и собственные проблемы, а именно: взаимоблокировки (deadlock), а также защита слишком большого или слишком малого количества данных.
В C++ для создания мьютекса следует сконструировать объект типа std::mutex , для захвата мьютекса служит функция-член lock(), а для освобождения – функция-член unlock(). Можно вызывать эти функции напрямую, но в этом случае необходимо помнить о вызове unlock() на каждом пути выхода из функции, в том числе и вследствие исключений. Однако в стандартной библиотеке также имеется шаблон класса std::lock_guard, который реализует идиому RAII – захватывает мьютекс в конструкторе и освобождает в деструкторе, – гарантируя тем самым, что захваченный мьютекс обязательно будет освобожден.

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value)
{
      std::lock_guard<std::mutex> guard(some_mutex);
      some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{
      std::lock_guard<std::mutex> guard(some_mutex);
      return std::find(some_list.begin(),some_list.end(),value_to_find)
            != some_list.end();
}
```

Хотя иногда такое использование глобальных переменных уместно, в большинстве случаев мьютекс и защищаемые им данные помещают в один класс, а не в глобальные переменные. Это не что иное, как стандартное применение правил объектно-ориентированного проектирования; помещая обе сущности в класс, вы четко даете понять, что они взаимосвязаны, а, кроме того, обеспечиваете инкапсулирование функциональности и ограничение доступа. В данном случае функции add_to_list и list_contains следует сделать функциями-членами класса, а мьютекс и защищаемые им данные – закрытыми переменными-членами класса. Так будет гораздо проще понять, какой код имеет доступ к этим данным и, следовательно, в каких участках программы необходимо захватывать мьютекс. Если все функции-члены класса захватывают мьютекс перед обращением к каким-то другим даннымчленам и освобождают по завершении действий, то данные оказываются надежно защищены от любопытствующих.

С другой стороны, мы понимаем, что это не совсем верно. Если какая-нибудь функция-член возвращает указатель или ссылку на защищенные данные, то уже неважно, правильно функции-члены управляют мьютексом или нет, ибо вы проделали огромную брешь в защите. Любой код, имеющий доступ к этому указателю или ссылке, может прочитать (и, возможно, модифицировать) защищенные данные, не захватывая мьютекс. Таким образом, для защиты данных с помощью мьютекса требуется тщательно проектировать интерфейс, гарантировать, что перед любым доступом к защищенным данным производится захват мьютекса, и не оставлять черных ходов. Для защиты данных с помощью мьютекса недостаточно просто «воткнуть» объект std::lock_guard в каждую функцию-член: один-единственный «отбившийся» указатель или ссылка сводит всю защиту на нет. На некотором уровне проверить наличие таких отбившихся указателей легко – если ни одна функция-член не передает вызывающей программе указатель или ссылку на защищенные данные в виде возвращаемого значения или выходного параметра, то данные в безопасности. Но стоит копнуть чуть глубже, как выясняется, что всё не так просто. Недостаточно проверить, что функции-члены не возвращают указатели и ссылки вызывающей программе, нужно еще убедиться, что такие указатели и ссылки не передаются в виде входных параметров вызываемым ими функциям, которые вы не контролируете. Это ничуть не менее опасно – что, если такая функция сохранит где-то указатель или ссылку, а потом какой-то другой код обратится к данным, не захватив предварительно мьютекс?

Здесь фундаментальная проблема заключается в том, что нельзя забывать помечать все участки кода, в которых имеется доступ к структуре данных, как взаимно исключающие. К сожалению, в этом стандартная библиотека C++ нам помочь не в силах: именно программист должен позаботиться о том, чтобы защитить данные мьютексом. Не передавайте указатели и ссылки на защищенные данные за пределы области видимости блокировки никаким способом, будь то возврат из функции, сохранение в видимой извне памяти или передача в виде аргумента пользовательской функции.

Однако и после обеспечения безопасности отдельных операций наши неприятности еще не закончились – гонки все еще возможны, даже для самого простого интерфейса. Рассмотрим структуру данных для реализации стека. Помимо конструкторов, имеется еще пять операций со стеком: push() заталкивает в стек новый элемент, pop() выталкивает элемент из стека, top() возвращает элемент, находящийся на вершине стека, empty() проверяет, пуст ли стек, и size() возвращает размер стека. Даже такой интерфейс уязвим для гонки. Проблема не в реализации на основе мьютексов, она присуща самому интерфейсу, то есть гонка может возникать даже в реализации без блокировок. Проблема в том, что на результаты, возвращенные функциями empty() и size(), нельзя полагаться – хотя в момент вызова они, возможно, и были правильны, но после возврата из функции любой другой поток может обратиться к стеку и затолкнуть в него новые элементы, либо вытолкнуть существующие, причем это может произойти до того, как у потока, вызвавшего empty() или size(), появится шанс воспользоваться полученной информацией.

Если экземпляр stack не является разделяемым, то нет ничего страшного в том, чтобы проверить, пуст ли стек с помощью empty(), а затем, если стек не пуст, вызвать top() для доступа к элементу на вершине стека. Но если объект stack является разде-
ляемым, то такая последовательность операций уже не безопасна, так как между вызовами empty() и top() другой поток мог вызвать pop() и удалить из стека последний элемент. Таким образом, мы имеем классическую гонку, и использование внутреннего мьютекса для защиты содержимого стека ее не предотвращает. Это следствие дизайна интерфейса. В простейшем случае мы могли бы просто декларировать, что top() возбуждает исключение, если в момент вызова в стеке нет ни одного элемента. Формально это решает проблему, но затрудняет программирование, поскольку теперь мы должны быть готовы к перехвату исключения, даже если вызов empty() вернул false. По сути дела, вызов empty() вообще оказывается ненужным.

Поток A	Поток B

`if(!s.empty())`
 	
`  int const value=s.top();`
 	
`  s.pop();`
`  do_something(value);`
	
	`(!s.empty())`
	
	`  int const value=s.top();`
	
	`  s.pop();`
	`  do_something(value);`

Ещё одно состояние гонки. Как видите, если работают только эти два потока, то между двумя обращениями к top() никто не может модифицировать стек, так что оба потока увидят одно и то же значение. Однако беда в том, что между обращениями к pop() нет обращений к top(). Следовательно, одно из двух хранившихся в стеке значений никто

даже не прочитает, оно будет просто отброшено, тогда как другое будет обработано дважды. Это еще одно состояние гонки, и куда более коварное, чем неопределенное поведение в случае гонки между empty() и top(), – на первый взгляд, ничего страшного не произошло, а последствия ошибки проявятся, скорее всего, далеко от места возникновения, хотя, конечно, всё зависит от того, что именно делает функция do_something().

Перейдем к проблеме взаимоблокировки. Допустим, есть два потока, и для выполнения некоторой операции они должны захватить два мьютекса, но сложилось так, что каждый поток захватил только один мьютекс и ждет другого. Ни один поток не может продолжить, так как каждый ждет, пока другой освободит нужный ему мьютекс. Такая ситуация называется взаимоблокировкой; это самая трудная из проблем, возникающих, когда для выполнения операции требуется захватить более одного мьютекса.

Общая рекомендация, как избежать взаимоблокировок, заключается в том, чтобы всегда захватывать мьютексы в одном и том же порядке, – если мьютекс A всегда захватывается раньше мьютекса B, то взаимоблокировка не возникнет. Иногда это просто, потому что мьютексы служат разным целям, а иногда совсем не просто, например, если каждый мьютекс защищает отдельный объект одного и того же класса. Рассмотрим, к примеру, операцию сравнения двух объектов одного класса. Чтобы сравнению не мешала одновременная модификация, необходимо захватить мьютексы для обоих объектов. Однако, если выбрать какой-то определенный порядок (например, сначала захватывается мьютекс для объекта, переданного в первом параметре, а потом – для объекта, переданного во втором параметре), то легко можно получить результат, обратный желаемому: стоит двум потокам вызвать функцию сравнения, передав ей одни и те же объекты в разном порядке, как мы получим взаимоблокировку.

В стандартной библиотеке есть на этот случай решение в виде функции std::lock, которая умеет захватывать сразу два и более мьютексов без риска получить взаимоблокировку.

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
      some_big_object some_detail;
      std::mutex m;
public:
      X(some_big_object const& sd):some_detail(sd){}
      friend void swap(X& lhs, X& rhs)
      {
            if(&lhs==&rhs)
                  return;
            std::lock(lhs.m,rhs.m);
            std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);
            std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);
            swap(lhs.some_detail,rhs.some_detail);
      }
};
```

Сначала проверяется, что в аргументах переданы разные экземпляры, поскольку попытка захватить std::mutex, когда он уже захвачен, приводит к неопределенному поведению. Затем мы вызываем std::lock(), чтобы захватить оба мьютекса, и конструируем два экземпляра std::lock_guard, – по одному для каждого мьютекса. Помимо самого мьютекса, конструктору передается параметр std::adopt_lock , сообщающий объектам std::lock_guard, что мьютексы уже захвачены, и им нужно лишь

принять владение существующей блокировкой, а не пытаться еще раз захватить мьютекс в конструкторе.
Дополнительные рекомендации по избежанию взаимоблокировок:

- Избегайте вложенных блокировок: не захватывайте мьютекс, когда захватили какой-то другой.
- Не вызывайте пользовательский код, удерживая мьютекс.
- Захватывайте мьютексы в фиксированном порядке.
- Пользуйтесь иерархией блокировок: Идея в том, чтобы разбить приложение на отдельные слои и выявить все мьютексы, которые могут быть захвачены в каждом слое. Программе будет отказано в попытке захватить мьютекс, если она уже удерживает какой-то мьютекс из нижележащего слоя. Чтобы проверить это во время выполнения, следует приписать каждому мьютексу номер слоя и вести учет мьютексам, захваченным каждым потоком.

Шаблон std::unique_lock обладает большей гибкостью, чем std::lock_guard, потому что несколько ослабляет инварианты – экземпляр std::unique_lock не обязан владеть ассоциированным с ним мьютексом. Прежде всего, в качестве второго аргумента конструктору можно передавать не только объект std::adopt_lock, заставляющий объект управлять захватом мьютекса, но и объект std::defer_lock, означающий, что в момент конструирования мьютекс не должен захватываться. Захватить его можно будет позже, вызвав функцию-член lock() объекта std::unique_lock (а *не* самого мьютекса) или передав функции std::lock() сам объект std::unique_lock. При простой замене lock_guard на unique_lock без надобности std::unique_lock потребляет больше памяти и выполняется чуть дольше, чем std::lock_guard. Та гибкость, которую мы получаем, разрешая экземпляру std::unique_lock *не* владеть мьютексом, обходится не бесплатно – дополнительную информацию надо где-то хранить и обновлять.

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);
class X
{
private:
      some_big_object some_detail;
      std::mutex m;
public:
      X(some_big_object const& sd):some_detail(sd){}
      friend void swap(X& lhs, X& rhs)
      {
            if(&lhs==&rhs)
            return;
            std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock);
            std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock);
            std::lock(lock_a,lock_b);□ □ Мьютексы захватываются
            swap(lhs.some_detail,rhs.some_detail);
      }
};
```

Объекты std::unique_lock можно передавать функции std::lock(), потому что в классе std::unique_lock имеются функции-члены lock(), try_lock() и unlock(). Для выполнения реальной работы они вызывают одноименные функции контролируемого мьютекса, а сами только поднимают в экземпляре std::unique_lock флаг, показывающий, что в данный момент этот экземпляр владеет мьютексом. Флаг необходим для того, чтобы деструктор знал, вызывать ли функцию unlock(). Если экземпляр *действительно* владеет мьютексом, то деструктор *должен* вызвать unlock(), в противном случае – *не*

должен. Опросить состояние флага позволяет функция-член owns_lock(). Естественно, этот флаг необходимо где-то хранить. Поэтому размер объекта std::unique_lock обычно больше, чем объекта std::lock_guard, и работает std::unique_lock чуть медленнее std::lock_guard, потому что флаг нужно проверять и обновлять. Если класс std::lock_guard отвечает вашим нуждам, то я рекомендую использовать его. Тем не менее, существуют ситуации, когда std::unique_lock лучше отвечает поставленной задаче, так как без свойственной ему дополнительной гибкости не обойтись.

Мы рассмотрели различные способы защиты данных, разделяемых между потоками. Но иногда требуется не только защитить данные, но и синхронизировать действия, выполняемые в разных потоках. Например, возможно, что одному потоку перед тем как продолжить работу, нужно дождаться, пока другой поток завершит какую-то операцию. В общем случае, часто возникает ситуация, когда поток должен ожидать какого-то события или истинности некоторого условия. Конечно, это можно сделать, периодически проверяя разделяемый флаг «задача завершена» или что-то в этом роде, но такое решение далеко от идеала. Необходимость в синхронизации операций – настолько распространенный сценарий, что в стандартную библиотеку C++ включены специальные механизмы для этой цели – условные переменные и будущие результаты (future).

Если один поток хочет дождаться, когда другой завершит некую операцию, то может поступить несколькими способами. Во-первых, он может просто проверять разделяемый флаг (защищенный мьютексом), полагая, что второй поток поднимет этот флаг, когда завершит свою операцию. Это расточительно по двум причинам: на опрос флага уходит процессорное время, и мьютекс, захваченный ожидающим потоком, не может быть захвачен никаким другим потоком. То и другое работает против ожидающего потока, поскольку ограничивает ресурсы, доступные потоку, которого он так ждет, и даже не дает ему возможность поднять флаг, когда работа будет завершена. Ожидающий поток потребляет ресурсы, которые пригодились бы другим потокам, в результате чего ждет дольше, чем необходимо.

Второй вариант – заставить ожидающий поток спать между проверками с помощью функции std::this_thread::sleep_for().Это уже лучше, потому что во время сна поток не расходует процессорное время. Но трудно выбрать подходящий промежуток времени. Если он слишком короткий, то поток все равно впустую тратит время на проверку; если слишком длинный – то поток будет спать и после того, как ожидание завершилось, то есть появляется ненужная задержка. Редко бывает так, что слишком длительный сон прямо влияет на работу программу, но в динамичной игре это может привести к пропуску кадров, а в приложении реального времени – к исчерпанию выделенного временного кванта.

Третий – и наиболее предпочтительный – способ состоит в том, чтобы воспользоваться средствами из стандартной библиотеки C++, которые позволяют потоку ждать события. Самый простой механизм ожидания события, возникающего в другом потоке (например, появления нового задания в конвейере), дают условные переменные. Концептуально условная переменная ассоциирована с каким-то событием или иным условием, причем один или несколько потоков могут ждать, когда это условие окажется выполненным. Если некоторый поток решит, что условие выполнено, он может известить об этом один или несколько потоков, ожидающих условную переменную, в результате чего они возобновят работу.

Стандартная библиотека C++ предоставляет не одну, а две реализации условных переменных: std::condition_variable и std::condition_variable_any. Оба класса объявлены в заголовке <condition_variable> . В обоих случаях для обеспечения синхронизации необходимо взаимодействие с мьютексом; первый класс может работать только с std::mutex, второй – с любым классом, который отвечает минимальным требованиям к «мьютексоподобию», отсюда и суффикс _any. Поскольку класс std::condition_variable_any более общий, то его использование может обойтись дороже с точки зрения объема потребляемой памяти, производительности и ресурсов операционной системы. Поэтому, если дополнительная гибкость не требуется, то лучше ограничиться классом std::condition_variable.

Допустим, мы имеем очередь, которая служит для передачи данных между двумя потоками. Когда данные будут готовы, поток, отвечающий за их подготовку, помещает данные в очередь, предварительно захватив защищающий ее мьютекс с помощью мьютекса. Затем он вызывает функцию-член notify_one() объекта std::condition_variable, чтобы известить ожидающий поток (если таковой существует).

По другую сторону находится поток, обрабатывающий данные. Он в самом начале захватывает мьютекс. Затем поток вызывает функцию-член wait() объекта std::condition_variable, передавая ей объект-блокировку и лямбда-функцию, выражающую ожидаемое условие. Лямбда-функции – это нововведение в C++11, они позволяют записать анонимную функцию как часть выражения и идеально подходят для задания предикатов для таких стандартных библиотечных функций, как wait().

В данном случае простая лямбда-функция []{return !data_queue.empty();} проверяет, что очередь не пуста (вызывая ее метод empty()), то есть что в ней имеются данные для обработки.

Затем функция wait() проверяет условие (вызывая переданную лямбда-функцию) и возвращает управление, если оно выполнено (то есть лямбда-функция вернула true). Если условие не выполнено (лямбда-функция вернула false), то wait() освобождает мьютекс и переводит поток в состояние ожидания. Когда условная переменная получит извещение, отправленное потоком подготовки данных с помощью notify_one(), поток обработки пробудится, вновь захватит мьютекс и еще раз проверит условие. Если условие выполнено, то wait() вернет управление, причем мьютекс в этот момент будет захвачен. Если же условие не выполнено, то поток снова освобождает мьютекс и возобновляет ожидание. Для этого необходимо использовать обёртку над мьютексом std::unique_lock – ожидающий поток должен освобождать мьютекс, когда находится в состоянии ожидания, и захватывать его по выходе из этого состояния, и стандартные функции захвата и освобождения мьютекса, а также std::lock_guard такой гибкостью не обладает. Если бы мьютекс оставался захваченным в то время, когда поток обработки спит, поток подготовки данных не смог бы захватить его, чтобы поместить новые данные в очередь, а, значит, ожидаемое условие никогда не было бы выполнено.

Механизм будущих результатов. Если поток должен ждать некоего одноразового события, то он каким-то образом получает представляющий его объект-будущее. Затем поток может периодически в течение очень короткого времени ожидать этот объект-будущее, проверяя, произошло ли событие, а между проверками заниматься другим делом. Можно поступить и иначе – выполнять другую работу до тех пор, пока не наступит момент, когда без наступления ожидаемого события двигаться дальше невозможно, и вот тогда ждать готовности будущего результата. С будущим результатом могут быть ассоциированы какие-то данные, но это необязательно. После

того как событие произошло (то есть будущий результат готов), сбросить объект-будущее в исходное состояние уже невозможно.
В стандартной библиотеке C++ есть две разновидности будущих результатов, реализованные в форме двух шаблонов классов, которые объявлены в заголовке <future>: уникальные будущие результаты (std::future<>) и разделяемые будущие результаты (std::shared_future<>). Эти классы устроены по образцу std::unique_ptr и std::shared_ptr. На одно событие может ссылаться только один экземпляр std::future , но несколько экземпляров std::shared_future. В последнем случае все экземпляры оказываются готовы одновременно и могут обращаться к ассоциированным с событием данным. Именно из-за ассоциированных данных будущие результаты представлены шаблонами, а не обычными классами; точно так же шаблоны std::unique_ptr и std::shared_ptr параметризованы типом ассоциированных данных. Если ассоциированных данных нет, то следует использовать специализации шаблонов std::future<void> и std::shared_future<void>. Хотя будущие результаты используются как механизм межпоточной коммуникации, сами по себе они не обеспечивают синхронизацию доступа. Если несколько потоков обращаются к единственному объекту-будущему, то они должны защитить доступ с помощью мьютекса или какого-либо другого механизма синхронизации. Однако, каждый из нескольких потоков может работать с собственной копией std::shared_future<> безо всякой синхронизации, даже если все они ссылаются на один и тот же асинхронно получаемый результат.
Самое простое одноразовое событие – это результат вычисления, выполненного в фоновом режиме. Мы видели, что класс std::thread не предоставляет средств для возврата вычисленного значения, поэтому посмотрим, как это можно сделать.
Допустим, вы начали какое-то длительное вычисление, которое в конечном итоге должно дать полезный результат, но пока без него можно обойтись. Для вычисления можно запустить новый поток, но придется самостоятельно позаботиться о передаче в основную программу результата, потому что в классе std::thread такой механизм не предусмотрен. Тут-то и приходит на помощь шаблон функции std::async (также объявленный в заголовке <future>).
Функция std::async позволяет запустить асинхронную задачу, результат которой прямо сейчас не нужен. Но вместо объекта std::thread она возвращает объект std::future, который будет содержать возвращенное значение, когда оно станет доступно. Когда программе понадобится значение, она вызовет функцию-член get() объекта-будущего, и тогда поток будет приостановлен до готовности будущего результата, после чего вернет значение.

```
#include <future>
#include <iostream>
int find_the_answer();
void do_other_stuff();
int main()
{
      std::future<int> the_answer=std::async(find_the_answer);
      do_other_stuff();
      std::cout<<”Ответ равен “<<the_answer.get()<<std::endl;
}
```

Шаблон std::async позволяет передать функции дополнительные параметры, точно так же, как std::thread.
Все блокирующие вызовы, рассмотренные до сих пор, приостанавливали выполнение потока на неопределенно долгое время – до тех пор, пока не произойдет ожидаемое событие. Часто это вполне приемлемо, но в некоторых случаях время ожидания

желательно ограничить. Например, это может быть полезно, когда нужно отправить сообщение вида «Я еще жив» интерактивному пользователю или другому процессу, или, когда ожидание действительно необходимо прервать, потому что пользователь устал ждать и нажал Cancel. Можно задать таймаут одного из двух видов: интервальный, когда требуется ждать в течение определенного промежутка времени, или абсолютный , когда требуется ждать до наступления указанного момента. У большинства функций ожидания имеются оба варианта. Варианты, принимающие интервальный таймаут, оканчиваются словом _for, а принимающие абсолютный таймаут – словом _until.

Например, в классе std::condition_variable есть по два перегруженных варианта функций-членов wait_for() и wait_until(), соответствующие двум вариантам wait() – первый ждет поступления сигнала или истечения таймаута или ложного пробуждения, второй проверяет при пробуждении переданный предикат и возвращает управление, только если предикат равен true (и условной переменной поступил сигнал) или истек таймаут.

Одна из самых важных особенностей стандарта C++11 – та, которую большинство программистов даже не замечают. Это не новые синтаксические конструкции и не новые библиотечные средства, а новая модель памяти, учитывающая многопоточность. Без модели памяти, которая точно определяет, как должны работать основополагающие строительные блоки, ни на одно из описанных выше средств нельзя было бы полагаться. Понятно, почему большинство программистов этого не замечают: если вы пользуетесь для защиты данных мьютексами, а для сигнализации о событиях – условными переменными или будущими результатам, то вопрос о том, почему они работают, не так уж важен. И лишь когда вы подбираетесь «ближе к железу», становятся существенны точные детали модели памяти.

C++ используется для решения разных задач, но одна из основных – системное программирование. Поэтому комитет по стандартизации в числе прочих целей ставил и такую: сделать так, чтобы в языке более низкого уровня, чем C++, не возникало необходимости. C++ должен обладать достаточной гибкостью, чтобы программист мог сделать то, что хочет, без помех со стороны языка, в том числе и работать «на уровне железа». Атомарные типы и операции – шаг именно в этом направлении, поскольку они предоставляют низкоуровневые механизмы синхронизации, которые обычно транслируются в одну-две машинные команды.

Под атомарными понимаются неделимые операции. Ни из одного потока в системе невозможно увидеть, что такая операция выполнена наполовину, – она либо выполнена целиком, либо не выполнена вовсе. Если операция загрузки, которая читает значение объекта, атомарна, и все операции модификации этого объекта также атомарны, то в результате загрузки будет получено либо начальное значение объекта, либо значение, сохраненное в нем после одной из модификаций.

И наоборот, если операция не атомарная, то другой поток может видеть, что она выполнена частично. Если это операция сохранения, то значение, наблюдаемое другим потоком, может не совпадать ни со значением до начала сохранения, ни с сохраненным значением. С другой стороны, операция загрузки может извлечь часть объекта, после чего значение будет модифицировано другим потоком, а затем операция прочитает оставшуюся часть объекта. В результате будет извлечено значение, которое объект не имел ни до, ни после модификации. Это простая проблематичная гонка, но на этом

уровне она может представлять собой гонку за данными и, стало быть, являться причиной неопределенного поведения.
В C++ для того чтобы операция была атомарной, обычно необходимы атомарные типы. Все стандартные атомарные типы определены в заголовке <atomic>. Любые операции над такими типами атомарны, и только операции над этими типами атомарны в смысле принятого в языке определения, хотя мьютексы позволяют реализовать кажущуюся атомарность других операций. На самом деле, и сами стандартные атомарные типы могут пользоваться такой эмуляцией: почти во всех имеется функция-член is_lock_free(), которая позволяет пользователю узнать, выполняются ли операции над данным типом с помощью действительно атомарных команд (x.is_lock_free() возвращает true) или с применением некоторой внутренней для компилятора и библиотеки блокировки (x.is_lock_free() возвращает false).
Единственный тип, в котором функция-член is_lock_free() отсутствует, – это std::atomic_fl ag. В действительности это по-настоящему простой булевский флаг, а операции над этим типом обязаны быть свободными от блокировок; если имеется простой свободный от блокировок булевский флаг, то на его основе можно реализовать простую блокировку и, значит, все остальные атомарные типы. Говоря по-настоящему простой, я именно это и имел в виду: после инициализации объект типа std::atomic_flag сброшен, и для него определены всего две операции: проверить и установить (функция-член test_and_set()) и очистить (функция-член clear()). Это всё – нет ни присваивания, ни копирующего конструктора, ни операции «проверить и очистить», вообще ничего больше.
Доступ ко всем остальным атомарным типам производится с помощью специализаций шаблона класса std::atomic<>; их функциональность несколько богаче, но они необязательно свободны от блокировок (как было объяснено выше). На самых распространенных платформах можно ожидать, что атомарные варианты всех встроенных типов (например, std::atomic<int> и std::atomic<void*>) действительно будут свободны от блокировок, но такого требования не предъявляется. Как мы скоро увидим, интерфейс каждой специализации отражает свойства типа; например, поразрядные операции, например &=, не определены для простых указателей, поэтому они не определены и для атомарных указателей.
Стандартные атомарные типы не допускают копирования и присваивания в обычном смысле, то есть не имеют копирующих конструкторов и операторов присваивания. Однако им все же можно присваивать значения соответствующих встроенных типов, и они поддерживают неявные преобразования в соответствующие встроенные типы. Кроме того, в них определены функции-члены load(), store(), exchange(), compare_exchange_weak() и compare_exchange_strong(). Поддерживаются также составные операторы присваивания (там, где это имеет смысл) +=, -=, *=, |= и т. д., а для целочисленных типов и специализаций std::atomic<> для указателей – еще и операторы ++ и --. Этим операторам соответствуют также именованные функции-члены с идентичной функциональностью: fetch_add(), fetch_or() и т. д. Операторы присваивания возвращают сохраненное значение, а именованные функции-члены – значение, которое объект имел до начала операции. Это позволяет избежать потенциальных проблем, связанных с тем, что обычно операторы присваивания возвращают ссылку на объект в левой части. Чтобы получить из такой ссылки сохраненное значение, программа должна была бы выполнить еще одну операцию чтения, но тогда между присваиванием и чтением другой поток мог бы модифицировать значение, открывая дорогу гонке.

Но шаблон класса std::atomic<> – не просто набор специализаций. В нем есть основной шаблон, который можно использовать для создания атомарного варианта пользовательского типа. Поскольку это обобщенный шаблон класса, определены только операции load(), store() (а также присваивание значения пользовательского типа и преобразования в пользовательский тип), exchange(), compare_exchange_weak() и compare_exchange_strong().

Рассмотрим, какие операции можно производить над каждым из стандартных атомарных типов.

Простейший стандартный атомарный тип std::atomic_flag представляет булевский флаг. Объекты этого типа могут находиться в одном из двух состояний: установлен или сброшен. Этот тип намеренно сделан максимально простым, рассчитанным только на применение в качестве строительного блока. Поэтому увидеть его в реальной программе можно лишь в очень специфических обстоятельствах. Тем не менее, он послужит нам отправной точкой для обсуждения других атомарных типов, потому что на его примере отчетливо видны общие относящиеся к ним стратегии.

Объект типа std::atomic_fl ag должен быть инициализирован значением ATOMIC_FLAG_INIT. При этом флаг оказывается в состоянии сброшен. Никакого выбора тут не предоставляется – флаг всегда должен начинать существование в сброшенном состоянии:

```
std::atomic_flag f=ATOMIC_FLAG_INIT;
```

Требование применяется вне зависимости от того, где и в какой области видимости объект объявляется. Это единственный атомарный тип, к инициализации которого предъявляется столь специфическое требование, зато при этом он является также единственным типом, гарантированно свободным от блокировок. Если у объекта std::atomic_flag статический класс памяти, то он гарантированно инициализируется статически, и, значит, никаких проблем с порядком инициализации не будет – объект всегда оказывается инициализированным к моменту первой операции над флагом.

После инициализации с флагом можно проделать только три вещи: уничтожить, очистить или установить, одновременно получив предыдущее значение. Им соответствуют деструктор, функция-член clear() и функция-член test_and_set().

Объект std::atomic_flag нельзя сконструировать копированием из другого объекта, не разрешается также присваивать один std::atomic_flag другому. Это не особенность типа std::atomic_flag, а свойство, общее для всех атомарных типов. Любые операции над атомарным типом должны быть атомарными, а для присваивания и конструирования копированием нужны два объекта. Никакая операция над двумя разными объектами не может быть атомарной. В случае копирования и присваивания необходимо сначала прочитать значение первого объекта, а потом записать его во второй. Это две отдельные операции над двумя различными объектами, и их комбинация не может быть атомарной. Поэтому такие операции запрещены.

Такая ограниченность функциональности делает тип std::atomic_flag идеальным средством для реализации мьютексов-спинлоков. Первоначально флаг сброшен и мьютекс свободен. Чтобы захватить мьютекс, нужно в цикле вызывать функцию test_and_set(), пока она не вернет прежнее значение false, означающее, что теперь в этом потоке установлено значение флага true. Для освобождения мьютекса нужно просто сбросить флаг.

```
class spinlock_mutex
{
      std::atomic_flag flag;
```

```
public:
      spinlock_mutex():
      flag(ATOMIC_FLAG_INIT)
      {}
      void lock()
      {
            while(flag.test_and_set(std::memory_order_acquire));
      }
      void unlock()
      {
            flag.clear(std::memory_order_release);
      }
};
```

Это очень примитивный мьютекс, но даже его уже достаточно. По своей природе, он активно ожидает в функции-члене lock(), поэтому не стоит использовать его, если предполагается хоть какая-то конкуренция, однако задачу взаимного исключения он решает.

Тип std::atomic_flag настолько ограничен, что его даже нельзя использовать в качестве обычного булевского флага, так как он не допускает проверки без изменения значения. На эту роль больше подходит тип std::atomic<bool>.

Из атомарных целочисленных типов простейшим является std::atomic<bool>. Как и следовало ожидать, его функциональность в качестве булевского флага богаче, чем у std::atomic_flag. Хотя копирующий конструктор и оператор присваивания по-прежнему не определены, но можно сконструировать объект из неатомарного bool, поэтому в начальном состоянии он может быть равен как true, так и false. Разрешено также присваивать объектам типа std::atomic<bool> значения неатомарного типа bool:

```
std::atomic<bool> b(true);
b=false;
```

Что касается оператора присваивания с неатомарным bool в правой части, нужно еще отметить отход от общепринятого соглашения о возврате ссылки на объект в левой части – этот оператор возвращает присвоенное значение типа bool. Такая практика обычна для атомарных типов: все поддерживаемые ими операторы присваивания возвращают значения (соответствующего неатомарного типа), а не ссылки. Если бы возвращалась ссылка на атомарную переменную, то программа, которой нужен результат присваивания, должна была бы явно загрузить значение, открывая возможность для модификации результата другим потоком в промежутке между присваиванием и чтением. Получая же результат присваивания в виде неатомарного значения, мы обходимся без дополнительной операции загрузки и можем быть уверены, что получено именно то значение, которое было сохранено.

Запись (любого значения: true или false) производится не чрезмерно ограничительной функцией clear() из класса std::atomic_flag, а путем вызова функции-члена store(), хотя семантику упорядочения доступа к памяти по-прежнему можно задать. Аналогично вместо test_and_set() используется более общая функция-член exchange(), которая позволяет атомарно заменить ранее сохраненное значение новым и вернуть прежнее значение. Тип std::atomic<bool> поддерживает также проверку значения без модификации посредством неявного преобразования к типу bool или явного обращения к функции load(). Как нетрудно догадаться, store() – это операция сохранения, load() – операция загрузки, а exchange() – операция чтения-модификации-записи:

```
std::atomic<bool> b;
bool x=b.load(std::memory_order_acquire);
b.store(true);
x=b.exchange(false,std::memory_order_acq_rel);
```

Функция exchange() – не единственная операция чтения-модификации-записи, которую поддерживает тип std::atomic<bool>; в нем также определена операция сохранения нового значения, если текущее совпадает с ожидаемым.
Сохранение (или несохранение) нового значения в зависимости от текущего
Новая операция называется «сравнить и обменять» и реализована в виде функций-членов compare_exchange_weak() и compare_exchange_strong(). Эта операция – краеугольный камень программирования с использованием атомарных типов; она сравнивает значение атомарной переменной с указанным ожидаемым значением и, если они совпадают, то сохраняет указанное новое значение. Если же значения не совпадают, то ожидаемое значение заменяется фактическим значением атомарной переменной. Функции сравнения и обмена возвращают значение типа bool, равное true, если сохранение было произведено, и false – в противном случае.
В случае compare_exchange_weak() сохранение может не произойти, даже если текущее значение совпадает с ожидаемым. В таком случае значение переменной не изменится, а функция вернет false. Такое возможно на машинах, не имеющих аппаратной команды сравнить-и-обменять, если процессор не может гарантировать атомарности операции – например, потому что поток, в котором операция выполнялась, был переключен в середине требуемой последовательности команд и замещен другим потоком (когда потоков больше, чем процессоров). Эта ситуация называется ложным отказом, потому что причиной отказа являются не значения переменных, а хронометраж выполнения функции.
Поскольку compare_exchange_weak() может стать жертвой ложного отказа, обычно ее вызывают в цикле:

```
bool expected=false;
extern atomic<bool> b; // установлена где-то в другом месте
while(!b.compare_exchange_weak(expected,true) && !expected);
```

Этот цикл продолжается, пока expected равно false, что указывает на ложный отказ compare_exchange_weak().
С другой стороны, compare_exchange_strong() гарантированно возвращает false только в том случае, когда текущее значение не было равно ожидаемому (expected). Это устраняет необходимость в показанном выше цикле, когда нужно только узнать, удалось ли нам изменить переменную или другой поток добрался до нее раньше.
Если мы хотим изменить переменную, каким бы ни было ее текущее значение (при этом новое значение может зависеть от текущего), то обновление expected оказывается полезной штукой; на каждой итерации цикла expected перезагружается, так что если другой поток не модифицирует значение в промежутке, то вызов compare_exchange_weak() или compare_exchange_strong() должен оказаться успешным на следующей итерации. Если новое сохраняемое значение вычисляется просто, то выгоднее использовать compare_exchange_weak(), чтобы избежать двойного цикла на платформах, где compare_exchange_weak() может давать ложный отказ (и, следовательно, compare_exchange_strong() содержит цикл). С другой стороны, если вычисление нового значения занимает длительное время, то имеет смысл использовать compare_exchange_strong(), чтобы не вычислять значение заново, когда expected не изменилась. Для типа std::atomic<bool> это не столь существенно – в конце концов, есть всего два возможных значения – но для более широких атомарных типов различие может оказаться заметным.

Еще одно отличие std::atomic<bool> от std::atomic_flag заключается в том, что тип std::atomic<bool> не обязательно свободен от блокировок; для обеспечения атомарности реализация библиотеки может захватывать внутренний мьютекс. В тех редких случаях, когда это важно, можно с помощью функции-члена is_lock_free() узнать, являются ли операции над std::atomic<bool> свободными от блокировок. Это еще одна особенность, присущая всем атомарным типам, кроме std::atomic_flag.

Следующими по простоте являются атомарные специализации указателей std::atomic<T*>.

Атомарная форма указателя на тип T – std::atomic<T*> – выглядит так же, как атомарная форма bool (std::atomic<bool>). Интерфейс по существу такой же, только операции применяются к указателям на значения соответствующего типа, а не к значениям типа bool. Как и в случае std::atomic<bool>, копирующие конструктор и оператор присваивания не определены, но разрешено конструирование и присваивание на основе подходящих указателей. Помимо обязательной функции is_lock_free(), тип std::atomic<T*> располагает также функциями load(), store(), exchange(), compare_exchange_weak() и compare_exchange_strong() с такой же семантикой, как std::atomic<bool>, но принимаются и возвращаются значения типа T*, а не bool.

Новыми в типе std::atomic<T*> являются арифметические операции над указателями. Базовые операции предоставляются функциями-членами fetch_add() и fetch_sub(), которые прибавляют и вычитают целое число из сохраненного адреса, а также операторы +=, -=, ++ и -- (последние в обеих формах – пред и пост), представляющие собой удобные обертки вокруг этих функций. Операторы работают так же, как для встроенных типов: если x – указатель std::atomic<Foo*> на первый элемент массив объектов типа Foo, то после выполнения оператора x+=3*x будет указывать на четвертый элемент и при этом возвращается простой указатель Foo*, который также указывает на четвертый элемент. Функции fetch_add() и fetch_sub() отличаются от операторов тем, что возвращают старое значение (то есть x.fetch_add(3) изменит x, так что оно будет указывать на четвертый элемент, но вернет указатель на первый элемент массива). Эту операцию еще называют обменять-и-прибавить, она относится к категории атомарных операций чтения-модификации-записи, наряду с exchange(), compare_exchange_weak() и compare_exchange_strong(). Как и другие операции такого рода, fetch_add() возвращает простой указатель T*, а не ссылку на объект std::atomic<T*>, поэтому вызывающая программа может выполнять действия над прежним значением.

Все прочие атомарные типы по существу одинаковы: это атомарные целочисленные типы с общим интерфейсом, различаются они только ассоциированными встроенными типами. Поэтому опишем их все сразу.

Помимо обычного набора операций (load(), store(), exchange(), compare_exchange_weak() и compare_exchange_strong()), атомарные целочисленные типы такие, как std::atomic<int> или std::atomic<unsigned long long>) обладают целым рядом дополнительных операций: fetch_add(), fetch_sub(), fetch_and(), fetch_or(), fetch_xor(), их вариантами в виде составных операторов присваивания (+=, -=, &=, |=, ^=) и операторами пред- и постинкремента и декремента (++x, x++, --x, x--). Это не весь набор составных операторов присваивания, имеющихся у обычного целочисленного типа, но близко к тому – отсутствуют лишь операторы умножения, деления и сдвига. Поскольку атомарные целочисленные значения обычно используются в качестве счетчиков или битовых масок, потеря не слишком велика, а в случае необходимости

недостающие операции можно реализовать с помощью вызова функции compare_exchange_weak() в цикле.
Семантика операций близка к семантике функций fetch_add() и fetch_sub() в типе std::atomic<T*>; именованные функции выполняют свои операции атомарно и возвращают старое значение, а составные операторы присваивания возвращают новое значение.
Операторы пред- и постинкремента и декремента работают как обычно: ++x увеличивает значение переменной на единицу и возвращает новое значение, а x++ увеличивает значение переменной на единицу и возвращает старое значение. Как вы теперь уже понимаете, результатом в обоих случаях является значение ассоциированного целочисленного типа.
Мы рассмотрели все простые атомарные типы; остался только основной обобщенный шаблон класса std::atomic<> без специализации.