Субъективный объективизм (http://www.scrutator.me/)

(https://twitter.com/ixsci) (http://www.scrutator.me/syndication.axd)

← Конструирование в C++11. Часть 1: Стирая границы (/post/2012/11/16/new-ctors-p1.aspx)
Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный → (/post/2012/06/03/parallel-world-p2.aspx)

Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

28. августа 2012 ixSci (http://www.scrutator.me/author/Adminaspx) разработка (/category/разработка.aspx) (4) (/post/2012/08/28/parallel-world-p3.aspx#comment)

Данная статья является третьей в цикле статей(часть 1 (/post/2012/04/04/parallel-world-p1.aspx) и часть 2 (/post/2012/06/03/parallel-world-p2.aspx)) о многозадачности в C++11. По сути, данная статья должна бы стоять первой в этом цикле, но, т.к. данный материал является более сложным, чем материал предыдущих частей, я решил несколько нарушить логический порядок. В статье речь пойдёт о самом низком уровне, появившейся в C++, многозадачности, а именно об изменениях в модели C++, появление атомарных типов и операций, а также об упущенном в прошлых статьях, локальном, по отношению к потоку, хранению объектов. В данной статье рассматриваются простейшие случаи использования атомарных объектов и операций. Этого должно быть достаточно большинству тех, кто всё таки решил воспользоваться низкоуровневыми примитивами. Более детальному рассмотрению будет посвящена последняя статья в цикле. А начнем мы с

Базовая модель

Прежде чем говорить о новшествах, давайте освежим в памяти то, что же мы имели раньше.

К примеру, если мы имели структуру следующего содержания:

```
struct Test
{
    int a;
    int b;
};
```

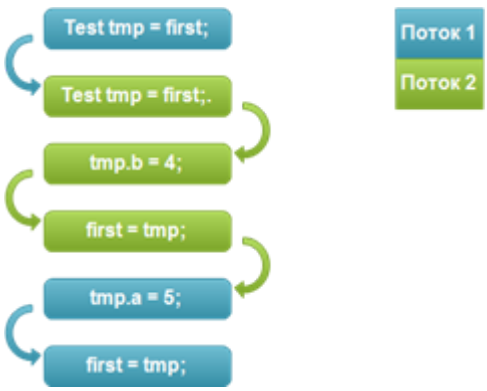
и глобальный объект этой структуры, который может быть модифицирован различными потоками:

```
Test first;//Глобальная облась видимости
    ...
first.a = 5;//Поток 1
    ...
first.b = 4;//Поток 2
```

Казалось бы всё здесь хорошо, разные потоки модифицируют разные части структура, но как бы не так! В C++03 нет многозадачности, а значит компилятор имеет полное право преобразовать код выше в следующий код:

```
Test first;//Глобальная область видимости
...
Test tmp = first;
tmp.a = 5;
first = tmp;//Поток 1
...
Test tmp = first;
tmp.b = 4;
first = tmp;//Поток 2
```

Что в результате будет в **first**? Не известно, и следующая картинка показывает возможный сценарий исполнения кода выше:



(/image.axd?picture=thread_1_1.png)

Итак, вместо ожидаемого пользователем **first{a=5, b=4}** мы получим **first{a=5, b=??}**. И это абсолютно легально в C++03 т.к. старая модель не подразумевает никакой многозадачности! Тоже самое относится и к другим структурам, таким как *битовые поля*, обновление одной части структуры провоцирует гонки, даже если другая часть структуры не задевается в этом потоке исполнения. Пометим это как случай I.

Другим недостатком модели C++03 является не возможность задания порядка исполнения выражения в одном потоке, относительно выражения в другом потоке. Оно и понятно, C++03 ничего не знает о потоках. Поэтому следующий код дает неопределенное поведение:

```
1  int shared = 0;//Глобальная область видимости
2  ...
3  ++shared;//Поток 1
4  ...
5  ++shared;//Поток 2
```

А неопределенное, оно в силу того, что возможен следующий сценарий:

(/image.axd?picture=thread_2_1.png)

При этом сценарии мы получим **shared=1**, возможен и сценарий, при котором мы получим **shared=2**. Еще одним препятствием на пути получения актуального значения **shared** является когерентность кэша (http://ru.wikipedia.org/wiki/Когерентность_кэша), т.к. нет гарантии, что один поток(корректнее говорить о ядрах, но я пропущу это для простоты) успеет “уведомить” другие потоки об обновлении **shared**. Т.е. даже при следующем сценарии, у нас нет гарантии, что мы получим **shared=2**:

(/image.axd?picture=thread_3_1.png)

Невозможность предоставления гарантии порядка исполнения среди разных потоков пометим как случай II

Единственным механизмом старой модели, привносящим порядок в исполнение кода, является наличие отношения *следует за(sequenced before).* Этот принцип довольно прост и интуитивно понятен: если выражение **B** *следует*(в коде) *за* выражением **A**, значит выражение **A** будет выполнено до выражения **B**, принцип транзитивен. Вот такой простой и незамысловатый принцип, который делает предсказуемым весь поток исполнения в C++03, когда отношение *следует за* может быть установлено. К сожалению оно не может быть уставлено между выражениями в двух различных потоками исполнения, ведь потоков не существует, вы помните? 😄

Кратко пробежав по недостаткам предыдущей модели переходим к

Атомарные типы и операции над ними

Появление многопоточности, помимо прочего, вносит сложность с загрузкой и сохранением данных. Ведь далеко не любая структура данных может быть загружена\сохранена в память одной операцией. Более того, то, что может быть загружено\сохранено в память одной операцией на одной архитектуре, не может быть на другой. А если такой операции нет, тогда появляется шанс, что пока один поток загрузил часть данных, второй поток прервал выполнение первого и изменил структуру, потом первый вернул управление и считал оставшуюся часть. Но это уже часть изменившейся структуры, а следовательно первый поток содержит некорректные данные!

Дабы извести подобные ситуации, на корню, и были введены атомарные объекты. Атомарный объект – это такой объект операции над которым можно считать неделимыми, т.е. такими, которые не могут быть прерваны или результат которых не может быть получен, до окончания операции. Таким образом исключается сама возможность получения некорректных данных в результате наблюдения половины записанных данных, к примеру.

Используя материал предыдущих статей, можно написать такой код:

```
1  int shared = 0;
2  ...
3  std::mutex mutex;
4  mutex.lock();
5  shared = 1;
6  mutex.unlock();
```

Здесь обновление **shared** можно считать атомарным, т.к. никто не может попасть в блок огражденный mutex'ми. В более общем случае всё, что ограждено mutex'ами можно считать атомарной операцией. Кто-то, возможно, спросит: "зачем тогда городить некие атомарные объекты, когда можно всё мьютексами оградить?". Причин тут, как минимум, 2: эффективность и безопасность. Мьютексы крайне не эффективный метод, и может стать причиной падения производительности, тогда как атомарные объекты *могут* быть гораздо эффективнее за счёт использования особой, процессорной, магии 😊. Безопасность же заключается в том, что вы запросто можете забыть оградить **shared** в каком-то месте программы и ваша атомарность на этом закончится. Атомарные объекты, напротив, не позволят вам забыться. Но, к сожалению, за всё приходится платить и в данном случае платой за эффективность будет сложность их использования. Ничего удивительного, ведь это самый низкий уровень многопоточных примитивов доступных в C++.

> Надо понимать, что мьютексы могут быть медленнее операций над атомарными объектами. Но не надо воспринимать это как мьютексы плохие, а атомарные объекты – наше всё. Это не так, они просто созданы для разных ситуаций и вам, как правило, не нужно использовать атомарные объекты пока вы не увидите очевидной выгоды от этого.

Внимательный читатель должен был заметить, что я сказал, что "атомарные объекты *могут* быть гораздо эффективнее". Я не сказал, что они *будут*, только *могут.* И для этого есть причина: атомарный доступ может понадобится к структуре любой сложности и размера, но настоящей атомарной операцией над данными можно считать лишь ту, которую процессор, определенной архитектуры, может выполнить одной командой. Вы прекрасно понимаете, что процессор не имеет таких команд для структур данных любой сложности. Поэтому, чтобы оставить возможность атомарного доступа, и не скатиться лишь к базовому набору типов, атомарность должна эмулироваться для всех типов, которые процессор не может обрабатывать атомарно! Возможно, за счёт тех же самых мьютексов. Именно поэтому мьютексы могут не проиграть атомарному доступу в некоторых случаях, теоретически.

К счастью, стандарт предоставляет возможность разработчику точно знать, является атомарный объект требуемого типа *свободным от блокировок(lock-free)* или нет. В связи с этим все атомарные объекты в C++11 можно разделить на две группы: *свободные от блокировок* и *не свободные от блокировок*. То, как определить к какой группе принадлежит тот или иной объект, и какие ещё гарантии даёт нам стандарт мы рассмотрим далее.

Атомарные объекты, представленные в C++11, также, можно разделить на 2 группы, по другому признаку:

- Объекты типом которых является *std::atomic<T>,* где *T* это тип данных, атомарный доступ к объекту которого требуется.
- Другие объекты.

К *другим объектам*, относится только один тип – *std::atomic_flag*. Причина, по которой этот тип стоит особняком, – проста, это тип является простейшим атомарным объектом и представляет собой булев флаг. Он содержит свой, уникальный набор операций и, самое главное, он единственный **гарантированно** является *свободным от блокировок*! Т.е. по стандарту все операции над объектом типа *std::atomic_flag* являются “чисто” атомарными, без каких либо условностей. Исходя из вышесказанного можно предположить, что остальные атомарные типы, для которых не существует *свободной от блокировок* версии на той или иной архитектуре, будут реализованы посредством *atomic_flag.*

atomic_flag содержит всего две операции: *test_and_set* и *clear,* чего, собственно говоря, вполне достаточно для флага, ведь он может быть либо поднятым, либо опущенным. Но отсутствует операция проверки значения флага, без модификации оного, что сильно ограничивает сферы его использования. Так же есть набор свободных функций, которые могут оперировать флагом:

- *std::atomic_flag_test_and_set*
- *atomic_flag_test_and_set_explicit*
- *atomic_flag_clear*
- *atomic_flag_clear_explicit*

Здесь, **_explicit* версии позволяют явно задать *порядок [появления] данных*(*memory ordering).*

> Порядок данных мы рассмотрим чуть позже, сейчас достаточно того, что таковой существует. Любая операция над содержимым атомарного объекта принимает одним из параметров порядок данных, по умолчанию это всегда std::*memory_order_seq_cst.*

Еще одним важным свойством *atomic_flag,* которое необходимо упомянуть, является его неопределенность при создании. Т.е. стандарт не оговаривает в каком состоянии находится флаг, ежели он не инициализирован. Поэтому, для получения предсказуемого результата есть смысл всегда инициализировать флаг; для этих целей существует специальный макрос *ATOMIC_FLAG_INIT.* Для инициализации флага, просто присвойте этот макрос вашему флагу, и, тогда, флаг инициализируется и гарантированно становится сброшенным:

```
1 std::atomic_flag flag = ATOMIC_FLAG_INIT;
```

Для иллюстрации работы флага напишем свой мьютекс, который является абсолютно неэффективным и простым, но, тем не менее, иллюстрирующим работу с флагом:

```
class CustomMutex
{
public:
    void lock()
    {
        while(m_Locked.test_and_set());
    }
    void unlock()
    {
        m_Locked.clear();
    }
private:
    std::atomic_flag m_Locked = ATOMIC_FLAG_INIT;
};
```

Теперь перейдем к более сложным структурам и рассмотрим класс

std::atomic<T>

*std::atomic<T>, я*вляясь базовым шаблоном для других атомарных типов, предоставляет нам следующие базовые операции:

- *is_lock_free* – то, о чем мы говорил выше. Предоставляет нам информации о свободности данного типа от блокировок.
- *store* – Кладет новое значение в объект.
- *load* – Извлекает значение из объекта.
- *exchange* – Заменяет значение в объекте на новое и возвращает старое.
- *compare_exchange_*(object, expected, desired, success, failure)* – Если **object** равен **expected**, тогда **desired** помещается в **object**. В противном случае **object** помещается в **expected**.
- *compare_exchange_weak* – вариант *compare_exchange* который может вернуть false, даже в случае если **object** равен **expected.** Это происходит благодаря пресловутой *фальшивой ошибке*(*spurious failure).* Поэтому, по аналогии с тем как мы боролись с этим в части 1 (/post/2012/04/04/Добро-пожаловать-в-параллельныи-мир-Часть-1-Мир-многопоточныи.aspx), *compare_exchange_weak* лучше использовать в цикле.
- *compare_exchange_strong* – гарантированно возвращает верный результат и не зависит от *фальшивой ошибки.*

Также существует *operator=()* и *operator T(),* которые эквивалентны *store* и *load* соответственно.

Вышеописанные операции являются общими и могут применятся к любому типу, но, помимо них в стандарте существуют и специализированные версии, которые расширяют возможности в зависимости от типа использованного в них. Давайте рассмотрим эти, специализированные версии:

std::atomic<integral>

Этот тип включает в себя все интегральные типы существующие в C++: *char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, char16_t, char32_t, wchar_t.*

Отличительной особенностью этой версии является наличие дополнительных операций, которые можно осуществлять с атомарным интегральным объектом:

- *fetch_add(object, value)* – атомарно помещает сумму (object **+** value) в object.
- *fetch_sub(object, value)* – атомарно помещает (object **–** value) в object.
- *fetch_and(object, value)* – атомарно помещает (object **&** value) в object.
- *fetch_or(object, value)* – атомарно помещает (object **|** value) в object.
- *fetch_xor(object, value)* – атомарно помещает (object **^** value) в object.

Для удобства использования также представлены соответствующие операторы ++, –, &= и т.д. которые являются обёрткой над соответствующими *fetch_** аналогами.

ВАЖНО помнить, что каждая из этих операций является атомарной, но их комбинация атомарной **НЕ ЯВЛЯЕТСЯ**. К примеру:

```
1  std::atomic<int>integer(0);
2  std::atomic<int> otherInteger(0);
3  integer++;//Атомарно
4  otherInteger += integer++;//Не атомарно!
```

На каждый интегральный тип существует свой *typedef*; так вы можете использовать *std::atomic_int* вместо *std::atomic<int>*(В MSVC2012 это разные типы, с какой-то стати). И так для каждого типа. Я не буду приводить весь список, если интересно посмотрите в заголовке *<atomic>* или же в стандарте.

Перейдем к еще одному типу, конкретизирующему *std::atomic* и добавляющему новые операции:

std::atomic<T*>

Этот тип используется для всех указателей, работа с которыми должна быть атомарна. Специализация для указателей схожа с оной для интегральных типов, за той лишь разницей что тут отсутствуют *fetch_or, fetch_xor* и *fetch_and*. Что и понятно, они лишены смысла, в контексте указателей. Также, очевидным отличием является использование типа *ptrdiff_t* в *fetch_sub* и *fetch_add*, а также наличие оператора разыменовывания указателя; оператор –>, к слову, отсутствует.

Помимо того, что все вышеперечисленные операции присущи соответствующему классу они, также, доступны как свободные функции с префиксом *atomic_**(*atomic_fetch_add* например).

В качестве примера, можно рассмотреть неэффективный мьютекс, из секции описывающей *std::atomic_flag*, с применением *atomic_bool*:

```
class CustomMutex
{
public:
    void lock()
    {
        while(m_Locked.exchange(true));
    }
    void unlock()
    {
        m_Locked.store(false);
    }
private:
    std::atomic_bool m_Locked = false;
};
```

Теперь рассмотрим пример из случая II, но изменим его, так, чтобы использовать атомарный объект:

```
std::atomic_int shared = 0;//Глобальная область видимости
...
++shared;//Поток 1
...
++shared;//Поток 2
```

Мы по прежнему не знаем, какая строчка выполнится раньше, в потоке 1 или 2. Но мы можем сказать, с полной уверенностью, что если обе строчки были выполнены то *shared == 2*. Таким образом случай II не является больше проблемой.

Итак, мы рассмотрели какие атомарные типы предоставляет нам стандарт, и какие операции мы можем выполнять над ними. Теперь пора бы рассмотреть, что это за *порядок данных* и “с чем его едят”, но сначала обратимся к истокам и рассмотрим

Обновленная модель

Новая модель, а точнее модель C++11, уже не отказывает потокам в существовании, а всецело пытается привнести порядок и предоставить некий глобальный поток исполнения, который может быть предсказуемым и последовательным.

> Под глобальным потоком исполнения я подразумеваю некий единый конвейер, в котором выполняются команды или выражения. Исполнение команд является четко определенным и исполняется в том порядке, в каком они попали в конвейер при исполнении программы.

Первым важным нововведением, которое мы рассмотрим, является появление нотации *ячейки памяти(memory location)*, определение которой звучать так: это объект, интегральный тип или последовательность смежных битовых полей не нулевого размера.

```
1  struct Test
2  {
3      int a;
4      int b;
5      int c:3;
6      int d:4;
7      int :0;
8      int f:8;
9  };
```

a, b, f,(c,d) – являются раздельными *ячейками памяти*. При этом **c** и **d** формируют отдельную ячейку памяти, но **c** не отделим от **d**, так как они оба являются частью одной *ячейки*.

Введя эту нотацию, комитет по стандартизации, также, предоставил и пояснение поведения потоков при обновлении *ячейки памяти*. Так, различные потоки могут обновлять различные *ячейки памяти* безопасно! А для нас это значит, что ситуация описанная в случае Iбольше не является не определенной. Т.к. **a, b, f,(c,d)** являются различными *ячейками памяти*, то их раздельно обновление не может воздействовать друг на друга! Заметьте, также, что части битового поля **(c,d)** не могут быть безопасно обновлены, т.е. не дается никакой гарантии, что при обновлении **c** не будет затронут **d** и наоборот. Следовательно случай I является решенным в C++11. Недостаток исправлен. Итак, оба случая, которые давали неопределенное поведение в C++03 с успехом решены с использованием C++11.

Следующими нововведением является обновленная нотация порядка исполнения; если раньше всё решалось посредством отношения *следует за*, то сейчас всё стало несколько сложнее. С появлением потоков и атомарных объектов, появилась необходимость в возможности упорядочить их исполнение. Так, для обеспечения порядка взаимодействия атомарных объектов вводится новый порядок, – *порядок изменения(ПИ).* Этот порядок учитывает наличие потоков, и является глобальным по отношению к ним, т.е. *ПИ* выстраивает порядок не относительно какого-либо потока а является

общим для всех потоков, в котором участвуют атомарные объекты данного *ПИ*. *ПИ* имеет отношение лишь к атомарным объектам, следовательно он влияет только на порядок выражений, в которых вовлечены атомарные объекты. Сейчас это может быть несколько непонятно; дальше мы рассмотрим *ПИ* подробнее.

> Кстати, глобальный порядок, о котором я писал выше составляется из ПИ и отношения “следует за”

ПИ, в свою очередь, регулируется отношением *происходит до(happens before)*: Если выражения **A** и **B** модифицируют некий атомарный объект **M** и **A** *происходит до* **B**, тогда **A** будет выполнено до **B** в *ПИ* по отношению к **M**. Исходя из этого можно заметить, что *ПИ* формируется для каждого атомарного объекта в отдельности, т.е. у каждого атомарного объекта свой *порядок изменения*(что не добавляет простоты понимания, надо сказать). Не отчаивайтесь если на данном этапе мало что понятно, всё это немного прояснится дальше.

Отношение *происходит до* формируется за счёт применения двух других отношений: уже известного нам *следует за* и *меж-поточно происходит до. Меж-поточно происходит до,* в свою очередь, регулируется отношением *синхронизируется с.*

> На самом деле это наглая ложь, так как отношений, регулирующих “м*еж-поточно происходит до*”, существует больше. Но мы не будем их рассматривать в рамках данной статьи так как они **действительно** тяжелы для понимания. Более того при использовании параметров по умолчанию для аргументов задающих **порядок данных** вышеприведенное предложение будет всегда верно. Более подробно это вопрос, со всем имеющимися отношениями, будет рассмотрен в следующей статье

Отношение *синхронизируется с* применяются когда существуют 2 и более операций над неким атомарным объектом в различных потоках. А если быть точным, то операции записи(store) *синхронизируются с* операциями загрузки(load) в различных потоках. Интуитивно это итак должно быть понятно, если в одном потоке, что-то записали, то в другом потоке при загрузке получим все то, что было записано. Вот что-значит *синхронизируется с*.

```
1  //Поток 1
2  g_nonAtomic = true;
3  flag.store(true);
4  ...
5  //Поток 2
6  if(flag.load())
7      assert(g_nonAtomic);
```

Здесь *store* в потоке 1 *синхронизируется с load* в потоке 2, а это в свою очередь значит, что если *store* был выполнен, тогда **g_nonAtomic** точно будет равен *true*. Откуда берётся эта гарантия объяснено ниже.

Если операция A, над атомарным объектом M, *синхронизируется с* операцией B. Тогда операция A *меж-поточно происходит до* операции B и, следовательно, *происходит до* операции B. Отношение *меж-поточно происходит до* транзитивно, т.е. если операция A *меж-поточно происходит до* операции B, а операция B *меж-поточно происходит до* операции C, значит операция A *меж-поточно происходит до* операции C.

Так как отношение *происходит до,* формируется за счёт отношений *следует за* и *меж-поточно происходит до,* можно выстроить следующую цепочку: Если операция **A**, над атомарным объектом **M**, *следует за* НЕ атомарной операцией **NA** и операция **A** *меж-поточно происходит до* операции **B**, значит операция **NA** *происходит до* операции B! Или кодом:

```
1  //Глобальная область видимости
2  std::atomic_bool flag = false;
3  int someValue = 0;
4  ...
5  //Поток 1
6  someValue = 10;//#1
7  flag.store(true);//#2
8  ...
9  //Поток 2
10 while(!flag.load())//#3
11     ;
12 assert(someValue == 10);//#4
```

Т.к. поток 2 ждёт появления true во флаге, а true может там появится только тогда, когда поток выполнит **#2**. **#2**, в свою очередь, с*ледует за **#1**,* следовательно выражение в *assert* всегда будет true! В этих отношениях заложена вся мощь синхронизационного потенциала атомарных объектов. Тут можно использовать простое мнемоническое правило: операции записи не могут “перепрыгивать” *store* и операции чтения не могут “выпрыгивать” из *load.* Т.е. все, что было записано до *store* будет видеться всеми процессорами, если *load* вернул значение записанное предыдущим *store*. И все, что в отношении *следует за*, должно быть загружено после *load* будет загружено после него. Главное понимать, что синхронизация происходит между парой *store/load* и поодиночке они никаких гарантий не предоставляют. Именно благодаря этому правилу у нас есть гарантия, что если true было помещено в *store*, то **someValue** будет равен 10. Запись в someValue не может “перепрыгнуть” *store*, и чтение **someValue** не может “выпрыгнуть” за *load*.

Порядок данных

В предыдущих параграфах я уже упоминал *порядок данных* и *std::memory_order_seq_cst.*

std::memory_order_seq_cst является наиболее строгим порядком, при его использование все, что написано про отношения в предыдущем параграфе будет выполнятся. При использование этого порядка существует, как-бы, единый порядок модификации атомарного объекта. Т.е. всегда существует некий порядок, в котором все потоки будут видеть изменения определенного объекта. Этот порядок может быть разным, в зависимости от того, какой поток добрался до атомарного объекта первым, но этот порядок всегда определен. Т.е. вы никогда не получите *неопределенное поведение*(*undefined behavior)* при использовании *std::memory_order_seq_cst.*

Например:

```
1  //Глобальная область видимости
2  size_t nonAtomic = 0;
3  std::atomic_bool flagA = false;
4  std::atomic_bool flagB = false;
5  ...
6  //Поток 1
7  nonAtomic++;
8  flagA.store(true);
9  ...
10 //Поток 2
11 nonAtomic++;
12 flagB.store(true);
13
14 //Поток 3
15 while(!flagA.load())
16     ;
17 if(flagB.load())
18     assert(nonAtomic == 2);
19 else
20     assert(nonAtomic != 0);
21
22 //Поток 4
23 while(!flagB.load())
24     ;
25 if(flagA.load())
26     assert(nonAtomic == 2);
27 else
28     assert(nonAtomic != 0);
```

имеет 3 возможных сценария:

- Флаг A будет установлен и прочитан до установки B
- Флаг B будет установлен и прочитан до установки A

- На стадии чтения оба флага A и B будут установлены

Всё зависит от того, какой поток получит управление раньше. Таким образом в данном примере мы имеем 3 разных сценария, каждый из которых жестко определен. “Все равно неопределенность!”, можете сказать вы, но это не совсем так. Т.к. все пути определены, не определён лишь какой из путей будет “выбран” в результате. В ваших руках находится возможность задания только одного пути, если пожелаете, или же использование каждого пути в своих целях.

Таким образом, *std::memory_order_seq_cst* позволяет строго соблюдать отношение *происходит до.*

> Может показаться, что именно такого поведения вы и ожидали, и что оно интуитивно понятно. И вы будете абсолютно правы, при использовании memory_order_seq_cst вы получаете вполне логичное и интуитивно понятное поведение. Именно поэтому этот порядок является порядком по умолчанию, ведь с помощью него реализуется последовательная согласованность (http://ru.wikipedia.org/wiki/Последовательная_консистентность). С остальными вариантами порядка данных, всё далеко не так просто. Именно поэтому они не рассмотрены здесь, и будут рассмотрены в отдельной статье.

Локальное хранилище

Еще одним нововведением C++11, касающимся потоков, является появление локального, по отношению к потоку, хранилища. Тогда как любые объекты, которые создаются внутри функции априори являются частью того потока, в котором функция вызывается, все глобальные объекты хранятся в единой области памяти и являются едиными для всех потоков.

> Здесь, под глобальными объектами понимаются объекты, которые хранятся в “глобальной памяти”, и не ограничены лишь объектами с глобальной область видимости. Т.е. такие объекты как: глобальный объект(без и с модификатором static) в общем пространстве имен, глобальный объект(без и с модификатором static) в пользовательском пространстве имен, статический член класса(объявленный с модификатором static), статический локальный объект функции(объявленный с модификатором static)

Локальное хранилище(thread local storage или TLS), призвано локализовать глобальный объекты для потоков, создав копию любого глобального объекта, имеющего специальную маркировку, для каждого потока. *Специальной маркировкой* является явное указание *thread_local* по отношению к некому глобальной объекту. При этом *thread_local* может сосуществовать с двумя другими модификаторами

static и *extern.* Таким образом, при использовании модификатора *thread_local*, каждый поток будем иметь свою, локальную копию объекта и не будет мешать другим потокам. Более того, каждая из этих копий будет создаваться при старте потока, и уничтожаться при его окончании:

```
1  thread_local int g_SomeGlobal;
2
3  class SomeClass
4  {
5  public:
6      thread_local static int m_ClassGlobal;
7  };
8
9  void foo()
10 {
11     thread_local static int i = 0;
12     i++;
13 }
```

Каждая из выше представленных переменных(**i**, **m_ClassGlobal** и **g_SomeGlobal**) является локальной для каждого потока. К примеру, если выполнить **foo**, в 10-и потоках **i** будет равен 1 в каждом из них, а не 10 во всех.

Остальные статьи цикла:

Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Часть 4: Порядки и отношения (/post/2015/08/14/parallel-world-p4.aspx)

Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)

Метки : C++ (http://www.scrutator.me/?tag=C%2b%2b), atomic (http://www.scrutator.me/?tag=atomic), thread_local (http://www.scrutator.me/?tag=thread_local), memory model (http://www.scrutator.me/?tag=memory+model)

Текущий рейтинг: 5.0 (4 голосов)

Похожие записи

Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Данная статья является третьей в цикле статей(часть 1 и часть 2) о многозадачности в C++11. По сути,

Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Продолжая тему начатую в части 1, перейдём от простейшего способа использования потоков, к следующем

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Ну вот мы и дождались, C++, наконец, перестал игнорировать ситуацию за окном и признал, что “исполня

Последние записи

Нет оценок

Вся правда об указателях. Часть 2: Памятная (/post/2015/12/30/pointers_demystified_p2.aspx)
Рейтинг: 5 / 2

Обзор книги Applied Cryptography (/post/2015/12/25/review_applied_cryptography.aspx)
Рейтинг: 5 / 1

Вся правда об указателях. Часть 1: Вводная (/post/2015/11/26/pointers_demystified_p1.aspx)
Рейтинг: 5 / 3

Обзор книги C# 5.0 in a Nutshell: The Definitive Reference (/post/2015/11/24/review_csharp5_in_a_nutshell.aspx)
Нет оценок

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)
Рейтинг: 5 / 2

Список категорий

(/category/feed/книги.aspx) книги (15) (/category/книги.aspx)

(/category/feed/разработка.aspx) разработка (43) (/category/разработка.aspx)

Список записей по годам/месяцам

2011

2012

2013

2014

2015

2016

Март (/2016/03/default.aspx) (2)