Das C++-Speichermodell

Rainer Grimm
Schulungen, Coaching und Technologieberatung

Multithreading mit C++11

C++'s Antwort auf die Anforderungen der Multicore-Architekturen.

Ein definiertes Speichermodell

- Atomare Operationen
- Partielle Ordnung von Operationen
- Sichtbare Effekte von Operationen

Eine standardisierte Threading-Schnittstelle

- Threads und Tasks
- Schutz und sicheres Initialisieren der Daten
- Thread-lokale Daten
- Synchronisation der Threads

Das C++-Speichermodell

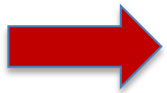
- Der Vertrag
- Atomare Datentypen
- Synchronisations- und Ordnungsbedingungen
- Singleton Pattern
- Sukzessive Optimierung

Das C++-Speichermodell

- **Der Vertrag**
- Atomare Datentypen
- Synchronisations- und Ordnungsbedingungen
- Singleton Pattern
- Sukzessive Optimierung

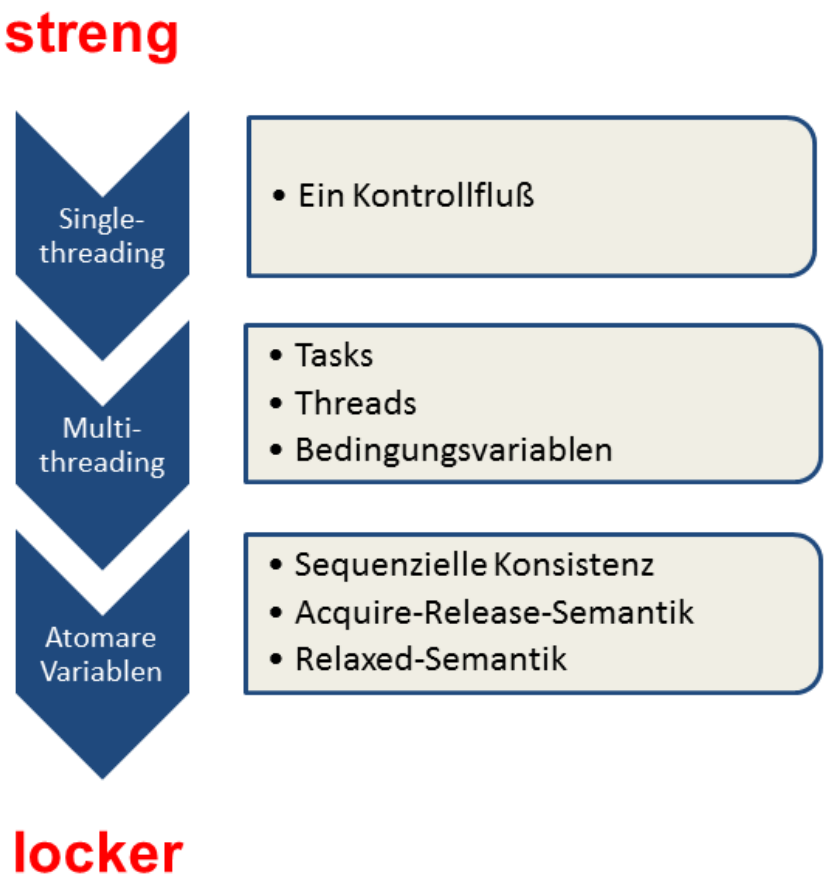
Der Vertrag

- Entwickler respektiert die Regeln
  - atomare Operationen
  - Partielle Ordnung von Operationen
  - Speichersichtbarkeit
- System besitzt die Freiheit zu optimieren
  - Compiler
  - Prozessor
  - Speicherebenen



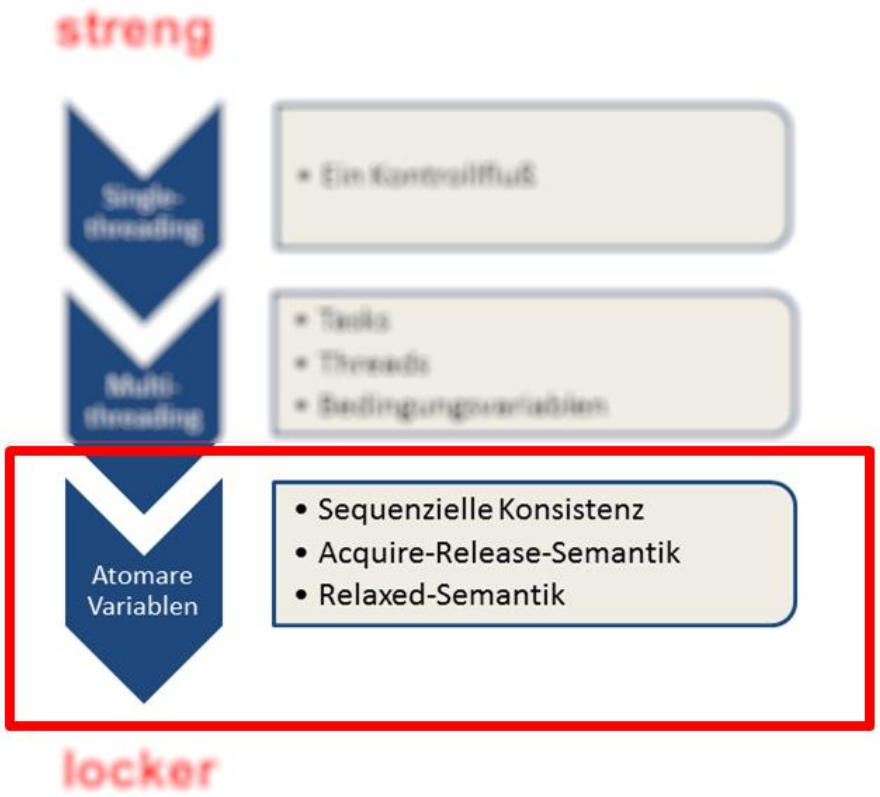
Hoch optimiertes Programm, das auf die Plattform zugeschnitten ist.

Der Vertrag



- Mehr Optimierungspotential für das System
- Anzahl der möglichen Kontrollflüsse steigt exponentiell
- Zunehmend ein ausschließliches Gebiet für Domänexperten
- Bruch der Intuition
- Feld für Mikrooptimierung

Der Vertrag



- Sequenzielle Konsistenz
  - *Strong Memory Model*
  - Globaler Zeitgeber
  - Speichermodell für Java und C#

Bruch der Sequenziellen Konsistenz

- Acquire-Release-Semantik
  - Synchronisation zwischen Threads mit atomaren Operationen
- Relaxed-Semantik
  - We*ak Memory Model*
  - Schwache Zusicherungen

Das C++-Speichermodell

Der Vertrag

Atomare Datentypen

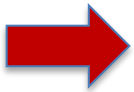
Synchronisations- und Ordnungsbedingungen

Singleton Pattern

Sukzessive Optimierung

Atomare Datentypen

Atomare Variablen sind die Grundlage für das C++-Speichermodell.



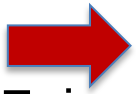
Atomare Operationen auf atomare Variablen definieren die Synchronisations- und Ordnungsbedingungen.

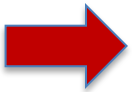
- Synchronisations- und Ordnungsbedingungen gelten für atomare Variablen und nicht atomare Variablen.
- Synchronisations- und Ordnungsbedingungen werden von höheren Abstraktionen verwendet.
  - Threads
  - Mutexe und Locks
  - Bedingungsvariablen
  - ...

Atomare Datentypen: std::atomic_flag

Das atomare Flag `std::atomic_flag`

- bietet ein sehr einfaches Interface an.
  - `clear` und `test_and_set`
- ist die einzige lockfreie Datenstruktur.

   Alle weiteren atomaren Datentypen für integrale Typen, Zeiger und eigene Datentypen können intern Locks verwenden.
- ist der elementarer Baustein für höhere Abstraktionen.

   Spinlock

Atomare Datentypen: std::atomic_flag

Spinlock

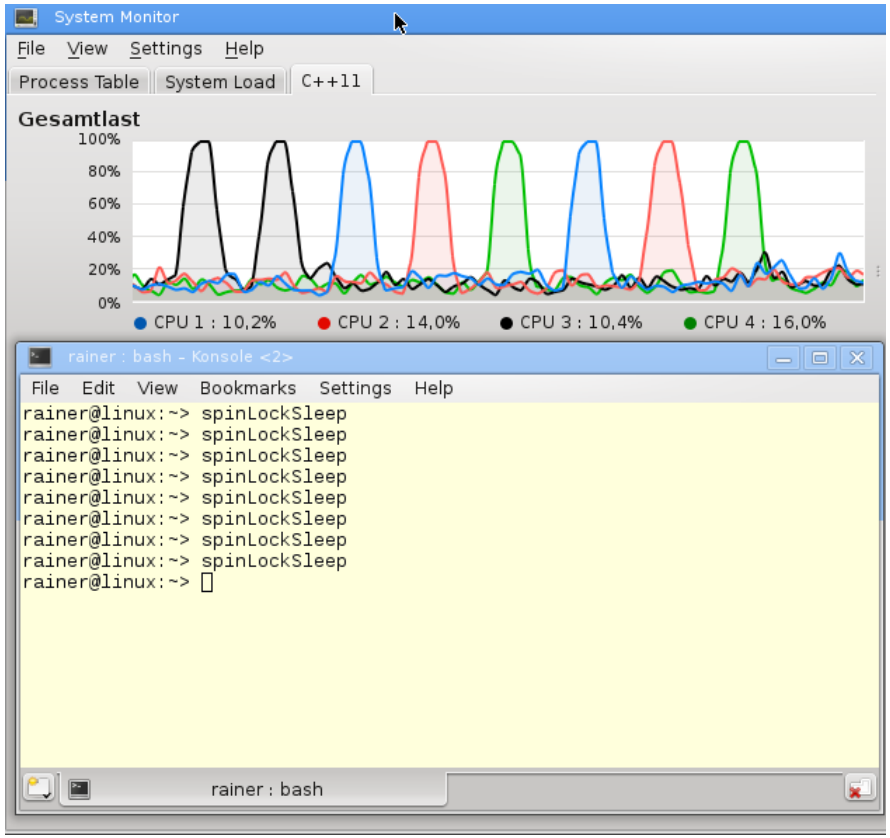
```
class Spinlock{
 std::atomic_flag  flag;
public:
  Spinlock():flag(ATOMIC_FLAG_INIT){}

  void lock(){
    while(flag.test_and_set());
  }

  void unlock(){
    flag.clear();
  }
};
```

Locken einer Ressource

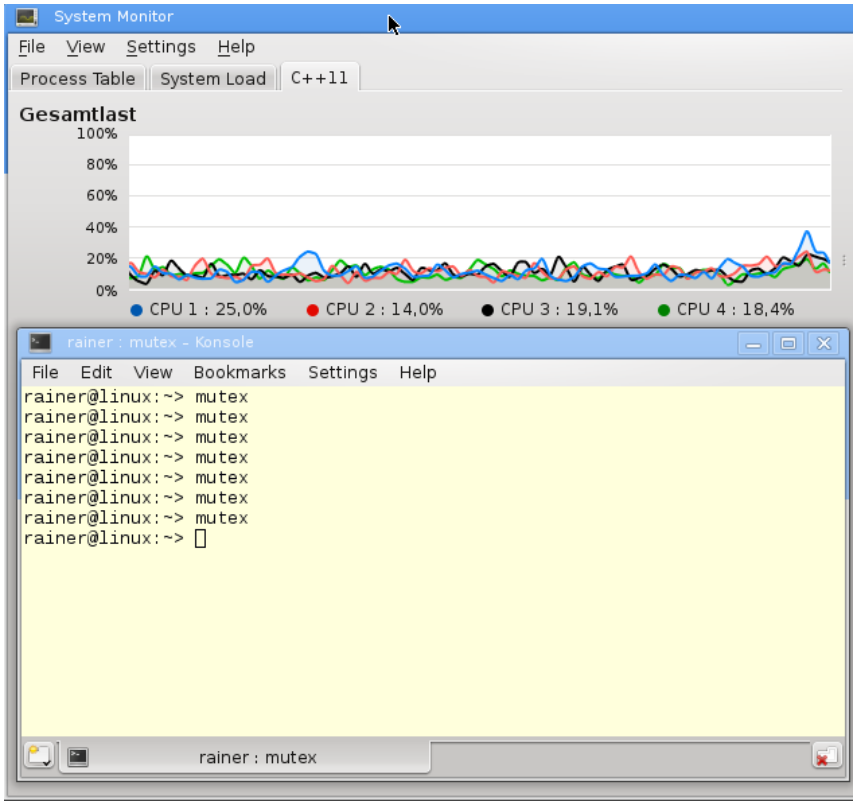
```
Spinlock spin;
// Mutex spin;
void workOnResource(){
  spin.lock();
  sleep_for(seconds(2));
  spin.unlock();
}
int main({
  thread t(workOnResource);
  thread t2(workOnResource);
  t.join();
  t2.join();
}
```

Atomare Datentypen: std::atomic_flag

Spinlock



Mutex



Atomare Datentypen: std::atomic<bool>

Der atomare Wahrheitswert `std::atomic<bool>`

- lässt sich explizit auf `true` oder `false` setzen.

- unterstützt die Funktion `compare_exchange_strong.`
  - Fundamentale Funktion für atomare Operationen.
  - Vergleicht und setzt einen Wert in einer atomaren Operation.
  - Syntax: `bool compare_exchange_strong(expected, desired)`
  - Strategie: `atom.compare_exchange_strong(exp,des)`

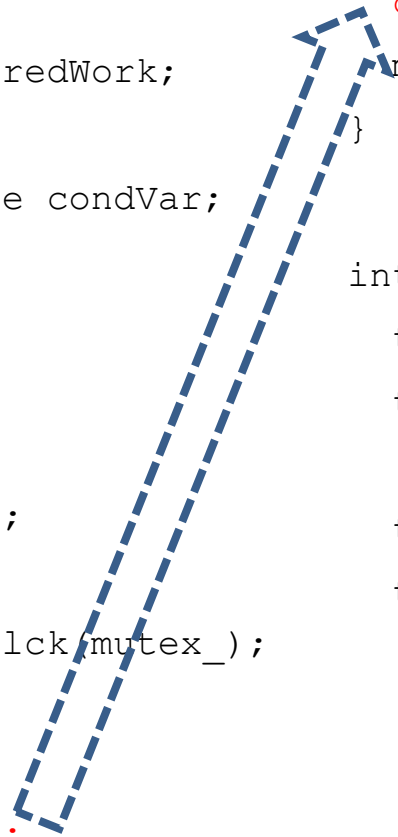
```
*atom == exp  ⇒  *atom= des
*atom != exp  ⇒   exp= *atom
```

- lässt sich als Bedingungsvariable verwenden.

Atomare Datentypen: Bedingungsvariable

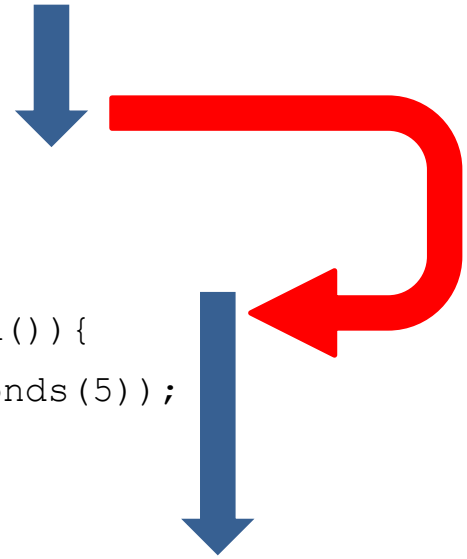
```
std::vector<int> mySharedWork;
std::mutex mutex_;
std::condition_variable condVar;

bool dataReady;

void setDataReady(){
  mySharedWork={1,0,3};
  {
    lock_guard<mutex> lck(mutex_);
    dataReady=true;
  }
  condVar.notify_one();
}
```

```
void waitingForWork(){
  unique_lock<mutex> lck(mutex_);
  condVar.wait(lck,[]{return dataReady;});
  mySharedWork[1]= 2;
}


int main(){
  thread t1(waitingForWork);
  thread t2(setDataReady);

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

   for (auto v: mySharedWork){
      std::cout << v << " ";
  } // 1 2 3
}
```



Atomare Datentypen: std::atomic<bool>

```
std::vector<int> mySharedWork;
std::atomic<bool> dataReady(false);

void setDataReady(){
  mySharedWork={1,0,3};
  dataReady= true;
}

void waitingForWork(){
  while (!dataReady.load()){
     sleep_for(milliseconds(5));
  }
  mySharedWork[1]= 2;
}
```



```
int main(){
  thread t1(waitingForWork);
  thread t2(setDataReady);
  t1.join();
  t2.join();
  for (auto v: mySharedWork){
    cout << v << " ";
  }
}  // 1 2 3
```

happens-before

synchronizes-with

Atomare Datentypen: std::atomic

Alle weiteren atomaren Datentypen sind teilweise oder vollständige Spezialisierungen von `std::atomic`.

```
std::atomic<T*>
std::atomic<Integraler Typ>
std::atomic<Eigener Typ>
```

- Für eigene Datentypen gelten Einschränkungen
  - Ihr Copy-Zuweisungsoperator und alle ihre Basisklassen müssen trivial sein.
  - Sie dürfen keine virtuellen Methoden und Basisklassen enthalten.
  - Sie müssen bitweise vergleichbar sein.

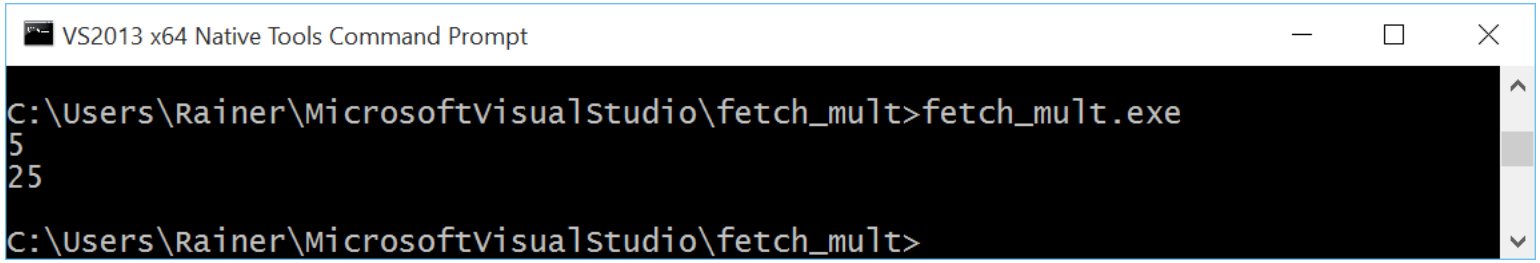
Atomare Datentypen: std::atomic

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<Integrale Typ>	atomic<Eigener Ty>
test_and_set	ja				
clear	ja				
is_lock_free		ja	ja	ja	ja
load		ja	ja	ja	ja
store		ja	ja	ja	ja
exchange		ja	ja	ja	ja
compare_exchange_weak compare_exchange_strong		ja	ja	ja	ja
fetch_add, += fetch_sub, -=			ja	ja	
fetch_or, |= fetch_and, &= fetch_xor, ^=				ja	
++ --			ja	ja	

! Es gibt keine Multiplikation oder Division.

Atomare Datentypen: std::atomic

```
template <typename T>
T fetch_mult(std::atomic<T>& shared, T mult){
  T oldValue= shared.load();
  shared.compare_exchange_strong(oldValue, oldValue * mult);
  return oldValue;
}
int main(){
  std::atomic<int> myInt{5};
  std::cout << myInt << std::endl;
  fetch_mult(myInt,5);
  std::cout << myInt << std::endl;
}
```



Das C++-Speichermodell

Der Vertrag

Atomare Datentypen

Synchronisations- und Ordnungsbedingungen

Singleton Pattern

Sukzessive Optimierung

Synchronisations- und Ordnungsbedingungen

C++ kennt sechs verschiedene Speichermodelle.

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

- Per Default gilt die Sequenzielle Konsistenz.
  - Das Speichermodell für C# und Java.
  - `memory_order_seq_cst`
  - Implizites Argument bei atomaren Operationen

```
std::atomic<int> shared;

shared.load() ≅ shared.load(std::memory_order_seq_cst);
```

Synchronisations- und Ordnungsbedingungen

Ordnung in das Speichermodell bringt die Beantwortung zweier Fragen.

1. Für welche atomaren Operationen sind die Speichermodelle konzipiert?

2. Welche Synchronisations- und Ordnungsbedingungen definieren die Speichermodelle?

Synchronisations- und Ordnungsbedingungen

1. Für welche atomaren Operationen sind die Speichermodelle konzipiert?

   - **read**-Operationen:
     `memory_order_acquire` und `memory_order_consume`
   - **write**-Operationen:
     `memory_order_release`
   - **read-modify-write**-Operationen:
     `memory_order_acq_rel` und `memory_order_seq_cst`

! `memory_order_relaxed` definiert keine Synchronisations- und Ordnungsbedingungen.

Synchronisations- und Ordnungsbedingungen

Operation	read Operationen	write Operationen	read-modify-write Operationen
test_and_set			√
clear		√	
is_lock_free	√		
load	√		
store		√	
exchange			√
compare_exchange_weak compare_exchange_strong			√
fetch_add, += fetch_sub, -=			√
fetch_or, |= fetch_and, &= fetch_xor, ^=			√
++ --			√

```
std::atomic<int> atom;
atom.load(std::memory_order_acq_rel) ≅ atom.load(std::memory_order_acquire)
atom.load(std::memory_order_release) ≅ atom.load(std::memory_order_relaxed)
```

Synchronisations- und Ordnungsbedingungen

2. Welche Synchronisations- und Ordnungsbedingungen definieren die Speichermodelle?

- **Sequenzielle Konsistenz**
  - Globale Ordnung auf allen Threads

    `memory_order_seq_cst`
- **Acquire-Release-Sematik**
  - Ordnung zwischen Lese- und Schreibeoperationen der gleichen atomaren Variablen auf verschiedenen Threads

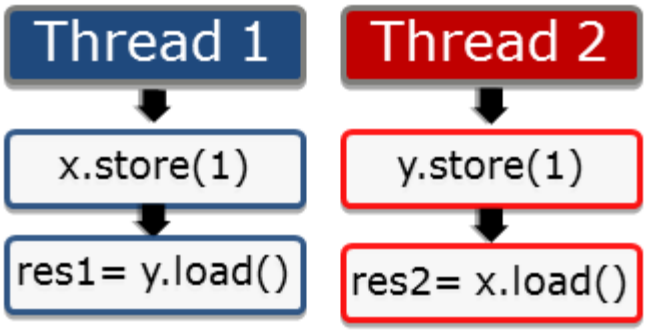
    `memory_order_consume`, `memory_order_acquire`, `memory_order_release` und `memory_order_acq_rel`
- **Relaxed-Semantik**
  - Keine Synchronisations- oder Ordnungsbedingungen

    `memory_order_relaxed`

Synchronisations- und Ordnungsbedingungen

Sequenzielle Konsistenz (Leslie Lamport 1979)

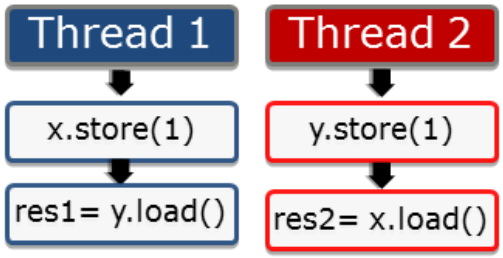
1. Die Anweisungen eines Programms werden in der Sourcecodereihenfolge ausgeführt.
2. Es gibt eine globale Reihenfolge aller Operationen auf allen Threads.



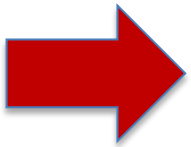
Sequenzielle Konsistenz ergibt

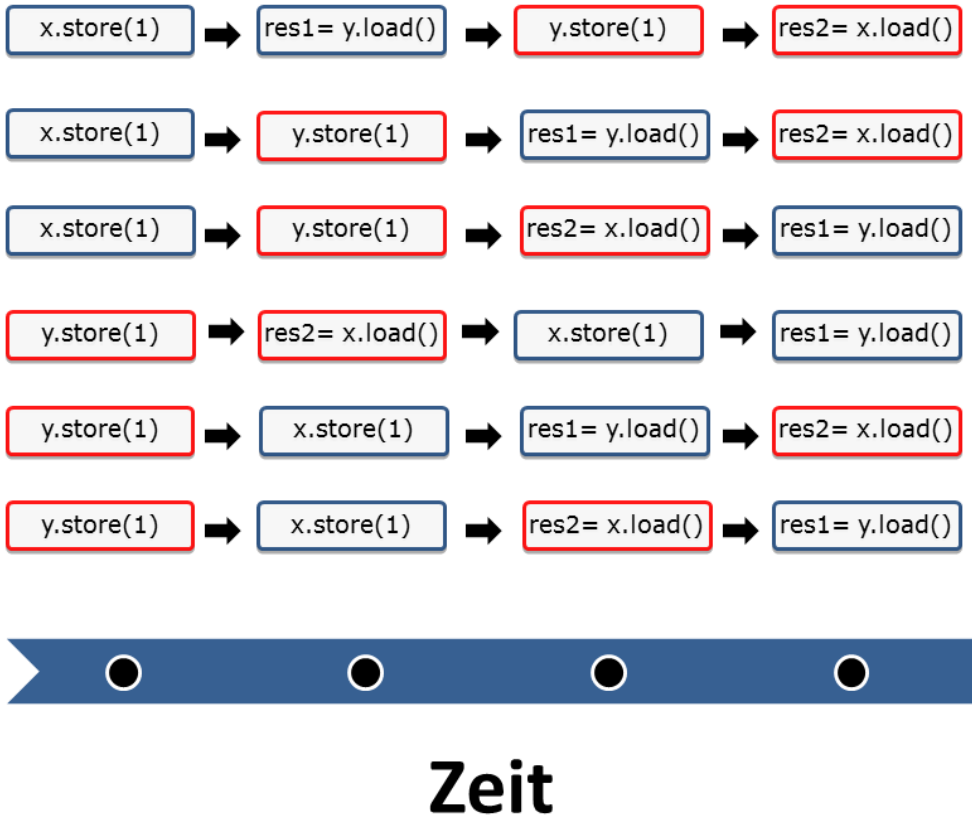
1. Die Befehle werde in der Reihenfolge ausgeführt, in der sie im Sourcecode stehen.
2. Jeder Thread sieht die Operationen jedes anderen Threads in der gleichen Reihenfolge (globaler Zeittakt).

Synchronisations- und Ordnungsbedingungen



mögliche Ausführungs-reihenfolgen





Synchronisations- und Ordnungsbedingungen

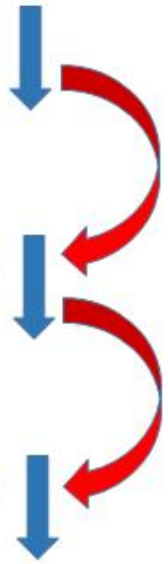
Acquire-Release-Semantik

- Eine release-Operation auf einer atomaren Variable synchronisiert sich mit einer acquire-Operation auf der gleichen atomaren Variablen und definiert eine Ordnungsbedingung.

- Acquire-Operation:
  - Lese-Operation (`load` oder auch `test_and_set`)
- Release-Operation:
  - Schreibe-Operation (`store` oder auch `clear`)
- Ordnungsbedingungen:
  - Lese- und Schreiboperationen können nicht **vor** eine acquire-Operation verschoben werden
  - Lese- und Schreiboperationen können nicht **hinter** eine release-Operation verschoben werden

Synchronisations- und Ordnungsbedingungen

```
void dataProducer(){
    mySharedWork={1,0,3};
    dataProduced.store(true, std::memory_order_release);
}

void deliveryBoy(){
    while( !dataProduced.load(std::memory_order_acquire) );
    dataConsumed.store(true,std::memory_order_release);
}

void dataConsumer(){
    while( !dataConsumed.load(std::memory_order_acquire) );
    mySharedWork[1]= 2;
}
```



Thread 1

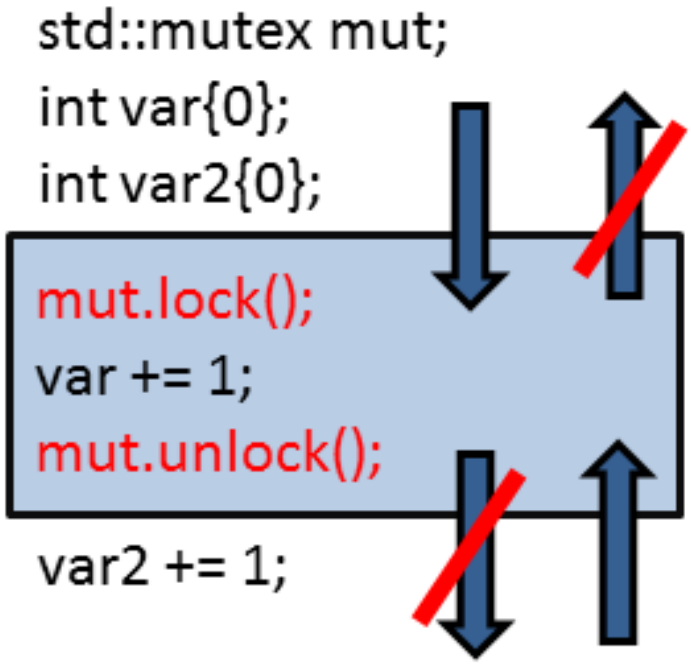
Thread 2

Thread 3

sequenced-before

synchronizes-with

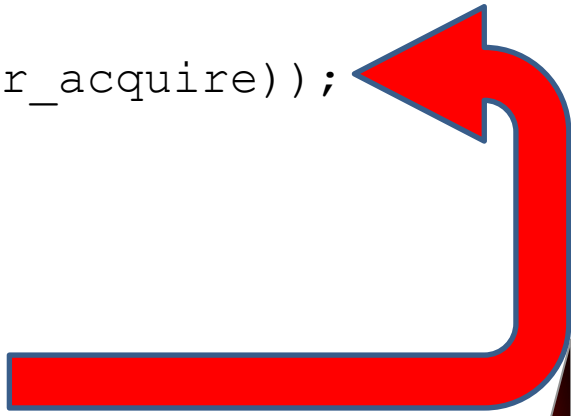
Synchronisations- und Ordnungsbedingungen



- Acquire-Operationen
  - Locken eines Mutex
  - Warten einer Bedingungsvariable
  - Starten eines Threads
- Release-Operationen
  - Unlocken eines Mutex
  - Benachrichtigung einer Bedingungsvariable
  - join-Aufruf auf einem Thread

Synchronisations- und Ordnungsbedingungen

```
class Spinlock{
  std::atomic_flag flag;
public:
  Spinlock(): flag(ATOMIC_FLAG_INIT) {}

  void lock(){
    while(flag.test_and_set(std::memory_order_acquire));
  }

  void unlock(){
    flag.clear(std::memory_order_release);
  }
};
```



Synchronisations- und Ordnungsbedingungen

Consume-Release-Semantik

- Consume-Release-Semantik ist die Acquire-Release-Semantik ohne Ordnungsbedingungen.

- Besitzt einen legendären Ruf
  - Sehr verständnisrersistent
  - `std::memory_order_consume` wird durch den Compiler auf `std::memory_order_acquire` abgebildet
  - Kein Compiler implementiert sie (*Ausnahme GCC*)

- Beschäftigt sich mit Datenabhängigkeiten
  - In einem Thread: *carries-a-dependency-to*
  - Zwischen Threads: *dependency-ordered-before*

Synchronisations- und Ordnungsbedingungen

```
atomic<string*> ptr;
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(14,memory_order_relaxed);
    ptr.store(p,memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_acquire)));
    cout << *p2 << " " << data;
    cout << atoData.load(memory_order_relaxed);
}
```

```
atomic<string*> ptr,
int data;
atomic<int> atoData;

void producer(){
    string* p  = new string("C++11");
    data = 2011;
    atoData.store(14,memory_order_relaxed);
    ptr.store(p, memory_order_release);
}

void consumer(){
    string* p2;
    while (!(p2 = ptr.load(memory_order_consume)));
    cout << *p2 << " " << data;
    cout << atoData.load(memory_order_relaxed);
}
```

Synchronisations- und Ordnungsbedingungen

```
std::atomic<std::string*> ptr;
int data;
std::atomic<int> atoData;

void producer(){
    std::string* p  = new std::string("C++11");
    data = 2011;
    atoData.store(2014,std::memory_order_relaxed);
    ptr.store(p, std::memory_order_release);
}

void consumer(){
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)));
    std::cout << "*p2: " << *p2 << std::endl;
    std::cout << "data: " << data << std::endl;
    std::cout << "atoData: " << atoData.load(std::memory_order_relaxed) << std::endl;
}
```

Synchronisations- und Ordnungsbedingungen

Relaxed Semantik

- Es gelten keine Synchronisations- und Ordnungsbedingungen. Nur die Atomizität der atomaren Operationen ist gewährleistet.

- Regeln
  - Atomare Operationen mit strengeren Speicherordnungen werden verwendet, um atomare Operationen mit Relaxed-Semantik und nicht atomare Operationen zu ordnen.
  - Operationen in einem Thread werden in Sourcecodeorder ausgeführt (*sequenced-before*).
  - Typische Anwendungsfälle  atomare Zähler (`shared_ptr`)

Threads können die Operationen in einem anderen Thread in anderer Reihenfolge wahrnehmen.

Synchronisations- und Ordnungsbedingungen

```
std::atomic<int> cnt = {0};
void f(){
    for (int n = 0; n < 1000; ++n) {
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
}
int main(){
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n){
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << cnt << '\n';
}
```

Das C++-Speichermodell

Der Vertrag

Atomare Datentypen

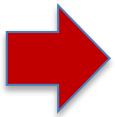
Synchronisations- und Ordnungsbedingungen

Singleton Pattern

Sukzessive Optimierung

Singleton

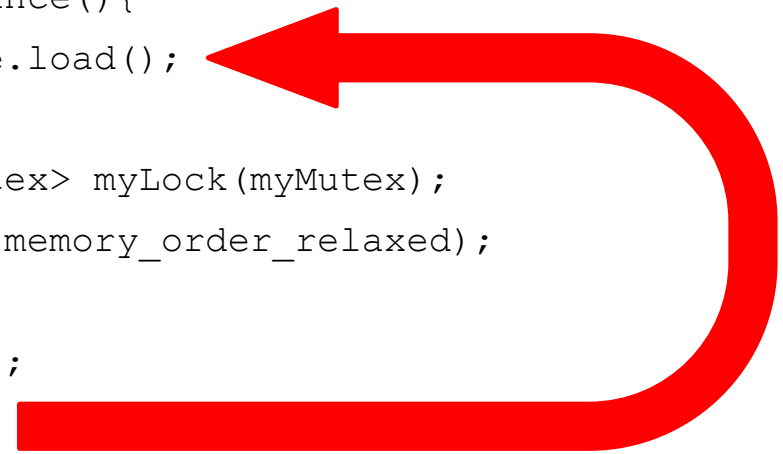
```
mutex myMutex;

class MySingleton{
public:
  static MySingleton& getInstance(){
    lock_guard<mutex> myLock(myMutex);
    if( !instance ) instance= new MySingleton();
    return *instance;
  }
private:
  MySingleton();
  ~MySingleton();
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
  static MySingleton* instance;
};
MySingleton::MySingleton()= default;
MySingleton::~MySingleton()= default;
MySingleton* MySingleton::instance= nullptr;
...
MySingleton::getInstance();
```



Performanzproblem

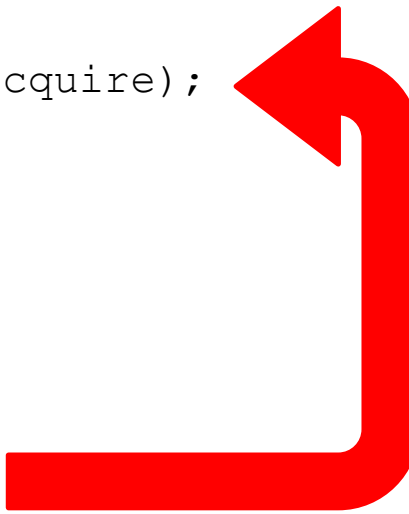
Singleton: Sequenzielle Konsistenz

```
class MySingleton{
public:
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load();
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load(std::memory_order_relaxed);
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin);
      }
    }
    return sin;
  }
private:
  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
  . . .
```



Singleton: Acquire-Release-Semantik

```
class MySingleton{
public:
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load(std::memory_order_acquire);
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load(std::memory_order_relaxed);
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin,std::memory_order_release);
      }
    }
    return sin;
  }
. . .
```



Singleton: Meyers Singleton

```
class MySingleton{
public:
  static MySingleton& getInstance(){
    static MySingleton instance;
    return instance;
  }
private:
  MySingleton()= default;
  ~MySingleton()= default;
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
};
```

Setzt Microsoft Visual Studio 2015 voraus.

Singleton

Performanzvergleich:

- Je vier Thread greifen 10'000'000 Mal auf ein Singleton zu
- Mein PC besitzt 4 Kerne
- Auf Linux mit maximaler Optimierung

std::lock_guard (Mutex)	Sequenzielle Konsistenz	Acquire-Release-Semantik	Meyers Singleton
12.47	0.09	0.07	0.04

- Die Details:
  - Vergleich Single-Threaded versus Multithreaded
  - Weitere Varianten
  - Windows (cl.exe) versus Linux (GCC)
  - Mit und ohne voller Optimierung

Threadsicheres Initialisieren eines Singleton

Das C++-Speichermodell

Der Vertrag

Atomare Datentypen

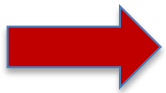
Synchronisations- und Ordnungsbedingungen

Singleton Pattern

Sukzessive Optimierung

Probleme?

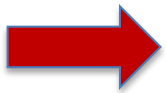
```
int x= 0;
int y= 0;

void writing(){
  x= 2000;
  y= 11;
}

void reading(){
  cout << "y: " << y << " ";
  cout << "x: " << x << endl;
}

int main(){
  thread thread1(writing);
  thread thread2(reading);
  thread1.join();
  thread2.join();
};
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

Mutex

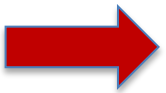
```
int x= 0;
int y= 0;
mutex mut;

void writing(){
  lock_guard<mutex> guard(mut);
  x= 2000;
  y= 11;
}

void reading(){
  lock_guard<mutex> guard(mut)
  cout << "y: " << y << " ";
  cout << "x: " << x << endl;
}
...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

Probleme?

```
volatile x= 0;
volatile y= 0;

void writing(){
  x= 2000;
  y= 11;
}

void reading(){
  cout << y << " ";
  cout << x << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

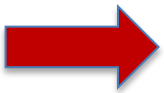
Java versus C++

Java volatile == C++ atomic

- `std::atomic`
  - Schutz der Daten vor gemeinsamen Zugriff mehrerer Threads
- `volatile`
  - Zugriff auf speziellen Speicher, auf dem Lesen und Schreiben nicht optimiert werden darf

Atomare Variablen

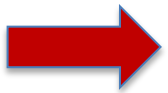
```
atomic<int> x= 0;
atomic<int> y= 0;

void writing(){
  x.store(2000);
  y.store(11);
}

void reading(){
  cout << y.load() << " ";
  cout << x.load() << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

Acquire-Release Semantik

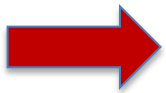
```
atomic<int> x= 0;
atomic<int> y= 0;

void writing(){
  x.store(2000,memory_order_relaxed);
  y.store(11, memory_order_release);
}

void reading(){
  cout << y.load(memory_order_acquire) << " ";
  cout << x.load(memory_order_relaxed) << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

Nicht atomare Variablen

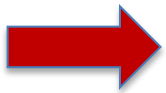
```
int x= 0;
atomic<int> y= 0;

void writing(){
  x= 2000;
  y.store(11, memory_order_release);
}

void reading(){
  cout << y.load(memory_order_acquire) << " ";
  cout << x << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

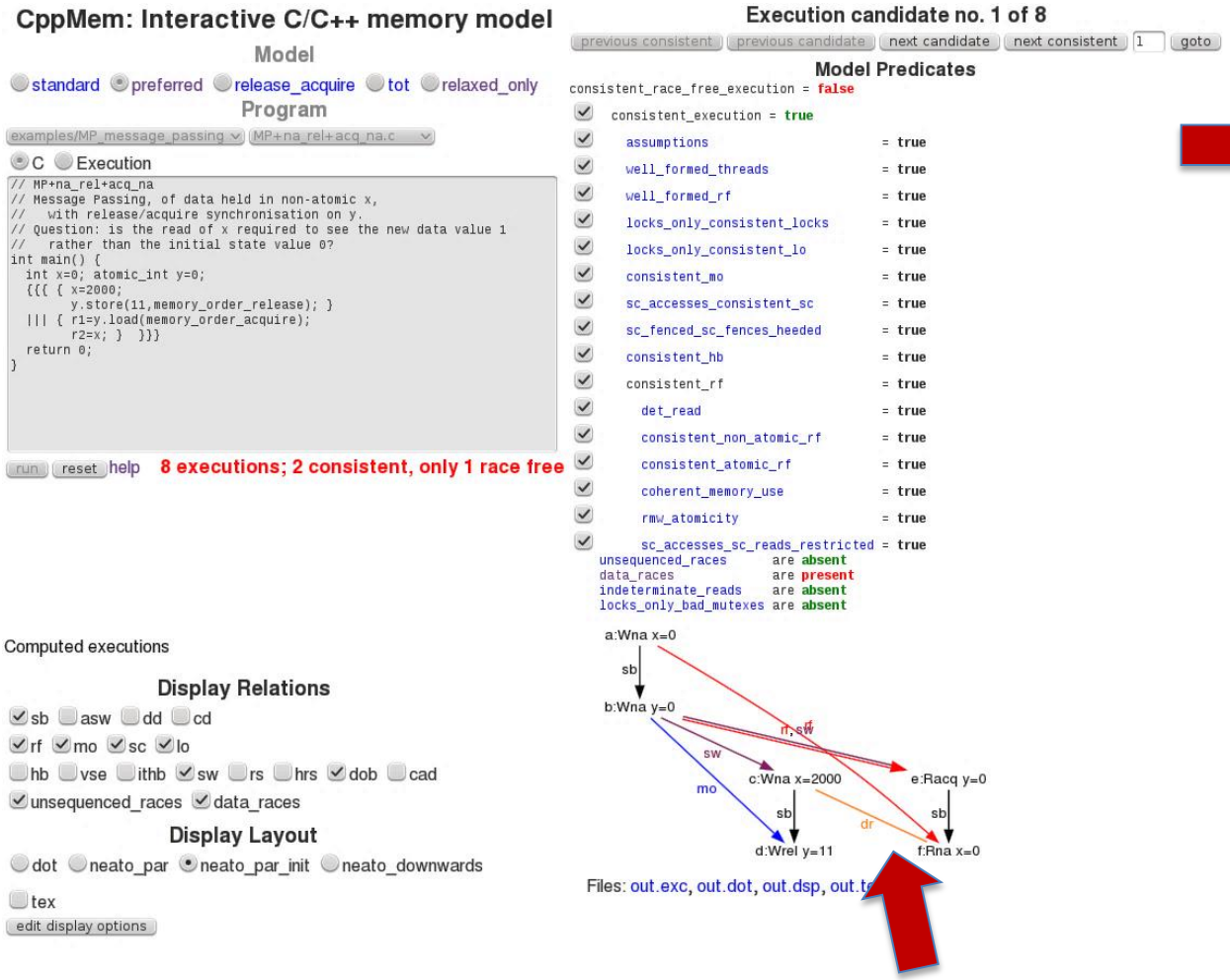
Relaxed Semantik

```
atomic<int> x= 0;
atomic<int> y= 0;

void writing(){
  x.store(2000,memory_order_relaxed);
  y.store(11, memory_order_relaxed);
}

void reading(){
  cout << y.load(memory_order_relaxed) << " ";
  cout << x.load(memory_order_relaxed) << endl;
}

...

thread thread1(writing);
thread thread2(reading);
```



y	x	Yes
0	0	
11	0	
0	2000	
11	2000	

Die Lösung: CppMem

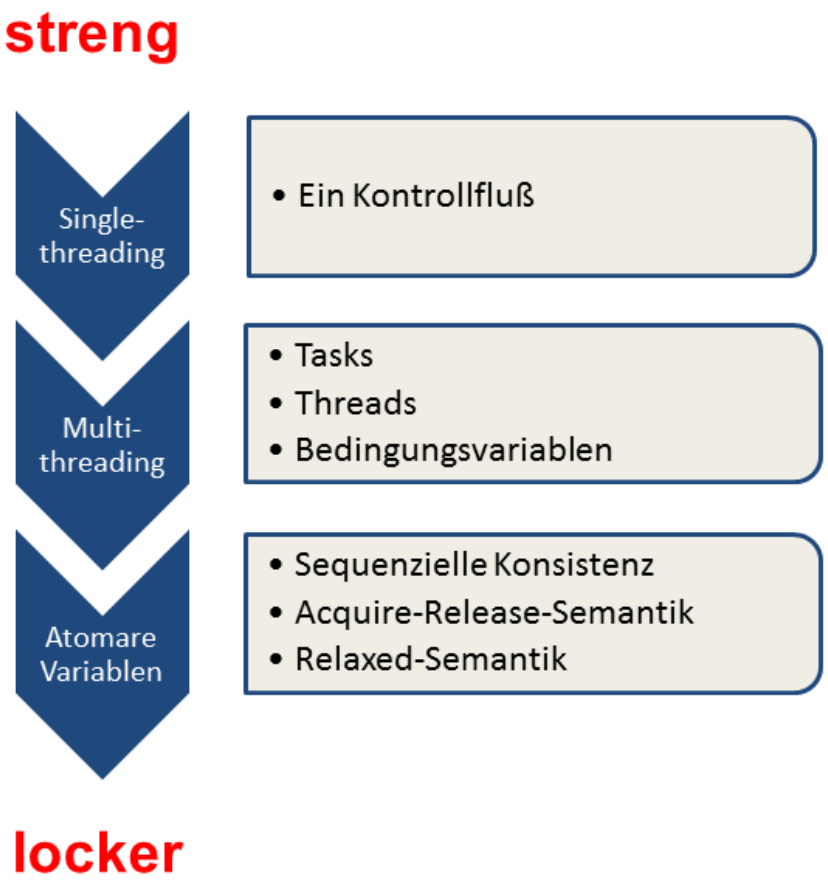


[CppMem]

Das C++-Speichermodell

- Der Vertrag
- Atomare Datentypen
- Synchronisations- und Ordnungsbedingungen
- Singleton Pattern
- Sukzessive Optimierung

Das C++-Speichermodell



- Mehr Optimierungspotential für das System
- Anzahl der möglichen Kontrollflüsse steigt exponentiell
- Zunehmend ein ausschließliches Gebiet für Domänexperten
- Bruch der natürlichen Intuition
- Feld für Mikrooptimierung

Weitere Informationen

- **Modernes C++:** Schulungen, Coaching und Technologieberatung durch Rainer Grimm
  - www.modernescpp.de
- Blog zu Modernem C++
  - www.grimm-jaud.de
- Kontakt
  - @rainer_grimm
  - schulungen@grimm-jaud.de


Rainer Grimm
Schulungen, Coaching und Technologieberatung