Den neuen Standard effektiv einsetzen

C++11

für Programmierer

O'REILLY®

Rainer Grimm

Rainer Grimm, C++11 für Programmierer, O´Reilly, ISBN 978-3-95561-391-4

Inhalt

1 Einführung . . . . . **IX**

Teil I: Tour de C++11 . . . . . **1**

2 Die Standardisierung **3**

3 Ziele von C++11 . . . . . **7**

4 Kernsprache **9**
Usability 9
Entwurf von Klassen 19
Rvalue-Referenzen 30
Generische Programmierung 33
Erweiterte Datenkonzepte und Literale 39
Weitere Aufräumarbeiten und Integration von C99 42

5 Multithreading **45**
Threads 47
Thread-lokale Daten 56
Synchronisation von Threads 57
Asynchrone Aufgaben 60

6 Die Standardbibliothek **65**
Neue Bibliotheken 66
Verbesserte Bibliotheken 77

Teil II: Kernsprache **95**

7 Usability **97**
Die Range-basierte For-Schleife 97
Automatische Typableitung 100
Lambda-Funktionen 109
Vereinheitlichte Initialisierung 120

8 Entwurf von Klassen **125**
Initialisierung von Objekten 125
Explizite Klassendefinitionen 135

9 Rvalue-Referenzen **151**
Lvalue- versus Rvalue-Referenzen 152
Move-Semantik 159
Perfect Forwarding 171

10 Generische Programmierung **177**
Variadic Templates 177
Zusicherungen zur Compile-Zeit 183
Aliase Templates 186

11 Erweiterte Datenkonzepte und Literale **189**
Konstante Ausdrücke 189
Plain Old Data (POD) 196
Unbeschränkte Unions 198
Streng typisierte Aufzählungstypen 201
Raw-String-Literale 205
Unicode-Unterstützung 207
Benutzerdefinierte Literale 210
nullptr 215

12 Removed und Deprecated **219**
Removed 219
Deprecated 219

Teil III: Multithreading **223**

13 Das C++11-Speichermodell **225**

14 Atomare Datentypen **229**

15 Threads **235**
Erzeugen von Threads 235
Lebenszeit der Daten 236
Operationen auf Threads 242

16 Gemeinsam von Threads genutzte Daten **247**
Schutz der Daten 247
Sichere Initialisierung der Daten 260

17 Thread-lokale Daten **269**

18 Synchronisation der Threads **273**

19 Asynchrone Aufgaben **283**
async 283
packaged_task 287
future und promise 292

Teil IV: Die Standardbibliothek **301**

20 Neue Bibliotheken **303**
Reguläre Ausdrücke 303
Type-Traits 338
Zufallszahlen 354
Zeitbibliothek 362
Referenz-Wrapper 372

21 Verbesserte Bibliotheken **379**
Smart Pointer 379
Neue Container 413
Neue Algorithmen 448
bind und function 456

Teil V: Ausblick **465**

22 Die nächsten C++-Standards **467**
C++14 467
C++17 470

Teil VI: Anhang **479**

A Build-Umgebung installieren **481**
Aktueller C++-Compiler 481
Boost-Bibliothek 483

B Funktionsobjekte **485**
Wie funktioniert ein Funktionsobjekt? 485
Welche Vorteile bietet ein Funktionsobjekt? 486

C Resource Acquisition Is Initialization **489**

D Implizit erzeugte Methoden und Operatoren **491**

E Promotion Trait **495**

F Funktionale Programmierung **499**
Programmieren mit mathematischen Funktionen 500
Charakteristiken funktionaler Programmierung 500

Literaturverzeichnis **519**

Index **525**

KAPITEL 4

Multithreading

In diesem Kapitel:

- Threads
- Thread-lokale Daten
- Synchronisation von Threads
- Asynchrone Aufgaben

Mehrkernprozessoren sind der Standard, wenn es um den Arbeitsplatzrechner, den heimischen PC oder den Laptop geht. Daher ist es von existenzieller Bedeutung für eine moderne Programmiersprache, auf die Anforderungen der modernen Rechnerarchitekturen adäquate Antworten zu geben – zumal funktionale Programmiersprachen wie Clojure (Clojure, 2011) oder auch Haskell (The Haskell Programming Language, 2011) die Messlatte bei der Unterstützung von Nebenläufigkeit sehr hoch gelegt haben. Sowohl Clojure als auch Haskell bieten Software Transactional Memory (STM) an.

Exkurs: Software Transactional Memory

Transaktionen sind eine bewährte Technik aus dem Datenbankumfeld, wenn es darum geht, konkurrierende Zugriffe auf Daten zu koordinieren. Die besondere Eigenschaft von Transaktionen ist, dass sie entweder vollständig oder gar nicht ausgeführt werden. Am Ende der Transaktion wird daher vom Transaktionssystem entschieden, ob die Transaktion veröffentlicht wird oder nicht. Ist die Veröffentlichung der Transaktion nicht möglich, wird sie in der Regel neu angestoßen.

ACID

Eine Transaktion zeichnet sich durch das Akronym ACID aus. ACID steht für:

◀ **Tabelle 4-1** Eigenschaften von Transaktionen – ACID

Akronym	Eigenschaft	Beschreibung
A	atomar	Transaktionen erfolgen in einem Schritt.
C	konsistent	Das System ist immer in einem konsistenten Zustand.
I	isoliert	Jede Transaktion verläuft in sich abgeschlossen.
D	dauerhaft	Das Ergebnis der erfolgreichen Transaktion wird automatisch gesichert.

→

Das ACID-Modell lässt sich, abgesehen von der Dauerhaftigkeit, auf den konkurrierenden Zugriff gemeinsam genutzter Daten verschiedener Threads übertragen. Im Gegensatz zum bekannten Locking, in dem die konkurrierende Ressource auf Verdacht gelockt wird, bevor ein Zugriff auf sie erfolgt, findet dieser beim optimistischen Ansatz von STM nicht statt. Beim STM werden die Daten verändert, und am Ende der Transaktion wird entschieden, ob das Ergebnis der Transaktion veröffentlicht wird.

Dieser optimistische Ansatz und ein weiteres Charakteristikum von STM, das darin besteht, dass es sehr einfach zu verstehen und anzuwenden ist, sind zwei große Vorteile von STM. Dagegen stehen der erhöhte Speicherverbrauch und die erhöhte CPU-Auslastung. Der Speicherverbrauch steigt beim STM, da jede Transaktion alle Daten, auf denen sie arbeitet, kopieren muss. Sind die Transaktionen sehr ungünstig strukturiert, kann das zu häufigen Wiederholungen einer Transaktion führen, bis sie erfolgreich durchgeführt wurde, und somit die CPU-Auslastung deutlich erhöhen.

Diese Abstraktion der Multithreading-Unterstützung erreicht C++11 noch nicht, aber es befindet sich auf dem richtigen Weg. Die neuen Features sind:

- eine standardisierte Threading-Schnittstelle, unabhängig von Betriebssystem und Compiler,
- ein definiertes Speichermodell und atomare Datentypen,
- mehrere Techniken zum Schutz der Daten vor konkurrierendem Zugriff,
- Bedingungsvariablen, um Threads durch Events zu synchronisieren,
- Threads, die lokale Daten halten,
- asynchrone Tasks in Form von Futures.

Die Darstellung des Speichermodells und der atomaren Datentypen wird erst in Teil III, *Multithreading*, auf Seite 223 Thema sein, da diese neuen Features deutlich das Niveau einer ersten Tour durch C++11 überschreiten.

Threads

Der Header <thread> inkludiert, und die neue Funktion std::thread steht zur Verfügung, um einen Thread zu erzeugen und sofort zu starten.

Erzeugung von Threads

Ein Thread std::thread benötigt die Funktionalität, die in ihm ausgeführt werden soll. Dazu bieten sich drei Möglichkeiten an:

- Funktionen
- Funktionsobjekte
- Lambda-Funktionen

Das Programm in Beispiel 4-1 stellt die drei Möglichkeiten dar.

Beispiel 4-1: Erzeugen von Threads mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion createThread.cpp

```
01 #include <iostream>
02 #include <thread>
03
04 void helloFunction(){
05   std::cout << "Hello C++11 from function." << std::endl;
06 }
07
08 class HelloFunctionObject  {
09   public:
10     void operator()() const {
11       std::cout << "Hello C++11 from a function object."
                   << std::endl;
12     }
13 };
14
15
16 int main(){
17
18   std::cout << std::endl;
19
20   // thread executing helloFunction
21   std::thread t1(helloFunction);
22
23   // thread executing helloFunctionObject
24   HelloFunctionObject helloFunctionObject;
25   std::thread t2(helloFunctionObject);
26
```

Beispiel 4-1: Erzeugen von Threads mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion (Fortsetzung)

```
27   // thread executing lambda function
28   std::thread t3([]
       {std::cout << "Hello C++11 from lambda function."
                  << std::endl;});
29
30   // ensure that t1, t2 and t3 have finished before main terminates
31   t1.join();
32   t2.join();
33   t3.join();
34
35   std::cout << std::endl;
36
37 }
```

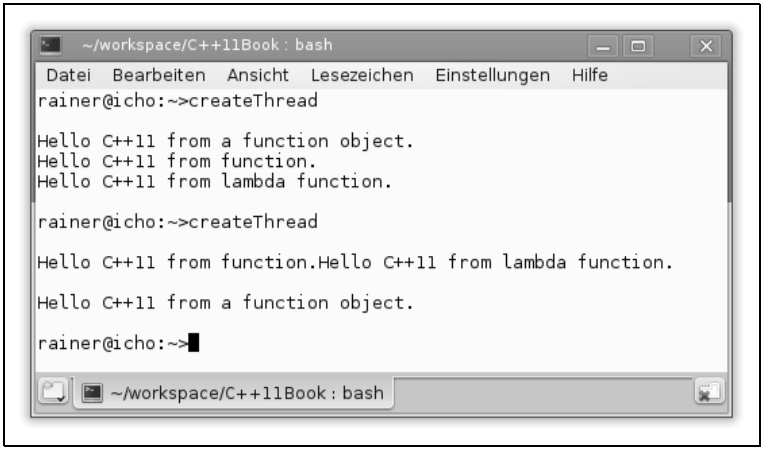
Sowohl der erste Thread `t1` als auch der zweite Thread `t2` in Beispiel 4-1 sollten relativ vertraut wirken. Anders verhält es sich mit dem letzten Thread `t3` (Zeile 28), dessen Funktionalität direkt in der Lambda-Funktion angegeben ist. Da in diesem konkreten Fall die Lambda-Funktion keine Argumente erwartet, ist es nicht notwendig, die Klammerpaare für die Argumente anzugeben. Die Lambda-Funktion `[](){ ... ;}` lässt sich daher auf `[]{ ... ;}` verkürzen. Threads, die nur ein paar Anweisungen ausführen müssen, sind ideale Kandidaten für Lambda-Funktionen, denn sie bieten entscheidende Vorteile:

- Die Codefunktionalität wird direkt dort definiert, wo sie benötigt wird.
- Keine unnötigen Funktionen oder Funktionsobjekte werden erzeugt.

join Eine Funktion fehlt noch in der Erläuterung. In den Zeilen 31 bis 33 wird auf jedem Thread `join` aufgerufen. Dies bewirkt, dass der Vater-Thread auf die Beendigung der drei Threads wartet, sodass diese ihre Aufgabe vollständig ausführen können, bevor der Vater-Thread sich beendet.

detach Durch `detach` wird das Gegenteil erreicht, denn diese Methode löst die Lebenszeit des neuen Threads vom Vater-Thread.

Das Ergebnis der Programmausführung ist, wie erwartet, nicht deterministisch.



◀ Abbildung 4-1
Ausführung von Threads mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion

Zwei Dinge fallen auf:

1. Es ist nicht vorhersagbar, welcher Thread am schnellsten seinen Code ausführt. Im ersten Durchlauf war das Funktionsobjekt der schnellste, im zweiten die Lambda-Funktion.
2. Alle Threads schreiben nach `std::cout`. Dies ist die gemeinsam genutzte Variable aller drei Threads. Das ist der Grund dafür, dass sich die Ausgabeoperationen der Funktion und des Funktionsobjekts überschneiden. Bevor die Funktion ihr `std::endl` nach `std::cout` schreiben kann, schreibt das Funktionsobjekt seine Ausgabe nach `std::cout`. Das fehlende `std::endl` wird daher zum Schluss des Programmlaufs ausgegeben.

Die korrekte Programmausführung setzt voraus, dass `std::cout` nur exklusiv von einem Thread verwendet werden kann. Die naheliegende Lösung ist Locking (dazu bald mehr im Abschnitt »Schutz der Daten« auf Seite 50).

Argumentübergabe

Die Threads waren sehr einfach strukturiert. Nun sollen sie Argumente erhalten. Als Grundlage dient das Programm in Beispiel 4-1.

Beispiel 4-2: Threads mit Argumentübergabe

createThreadWithArguments.cpp

```
01 #include <iostream>
02 #include <string>
03 #include <thread>
04
05 void helloFunction(const std::string& s){
06   std::cout << s << std::endl;
07 }
08
09 class HelloFunctionObject{
10   public:
```

Beispiel 4-2: Threads mit Argumentübergabe (Fortsetzung)

```
11     void operator()(const std::string& s) const {
12       std::cout << s << std::endl;
13     }
14 };
15
16
17 int main(){
18
19   std::cout << std::endl;
20
21   // thread executing helloFunction
22   std::thread t1(helloFunction,
                    "Hello C++11 from function.");
23
24   // thread executing helloFunctionObject
25   HelloFunctionObject helloFunctionObject;
26   std::thread t2(helloFunctionObject,
                    "Hello C++11 from function object.");
27
28   // thread executing lambda function
29   std::thread t3([](const std::string& s)
       {std::cout << s << std::endl;},
       "Hello C++11 from lambda function.");
30
31   // ensure that t1, t2 and t3 have finished before main
        terminates
32   t1.join();
33   t2.join();
34   t3.join();
35
36   std::cout << std::endl;
37
38 }
```

Die Ausgabe des Programmlaufs entspricht im Wesentlichen der von Abbildung 4-1. Das nicht deterministische Verhalten besteht weiter darin, welcher Thread als Erster zum Zuge kommt und ob sich die Ausgaben auf die Konsole überschneiden. Interessanter ist da schon die Übergabe der Parameter an die Funktion (Zeile 22), an das Funktionsobjekt (Zeile 26) und vor allem an die Lambda-Funktion (Zeile 29).

Gemeinsam von Threads genutzte Daten wie `std::cout` müssen geschützt werden. Dafür gibt es in C++11 Mutexe und Locks.

Schutz der Daten

Mutex

Mutex steht für den englischen Ausdruck *mutual exclusion*. Durch wechselseitigen Ausschluss stellt der Mutex sicher, dass nur ein

Thread Zugriff auf einen gemeinsam genutzten kritischen Bereich besitzt. Dieser kritische Bereich kann aus einem Variablenzugriff oder auch aus mehreren Anweisungen bestehen, die es zu schützen gilt.

Will ein Thread in den kritischen Bereich eintreten, muss er den Mutex locken. Dies ist aber nur möglich, wenn dieser nicht gelockt ist. Erhält der Thread den Lock nicht, wird er geblockt.

Der Gebrauch ist denkbar einfach.

mutex

```
std::mutex m;
// ...
m.lock();
//critical region
m.unlock();
```

Trotz dieser einfachen Nutzung sollte ein Mutex nicht direkt verwendet werden, denn er ist nur ein einfaches Werkzeug. Da ein Mutex in der Regel an mehreren Stellen im Sourcecode verwendet wird, ist die Gefahr sehr groß, dass er nicht mehr freigegeben wird. Das kann durch eine Nachlässigkeit oder durch eine Ausnahme passieren. Das Ergebnis ist das gleiche. Der Thread erhält den Mutex nicht mehr und bleibt geblockt.

Praxistipp **Erzeugen Sie einen künstlichen Bereich, um die Lebenszeit einer automatischen Variablen genau vorzugeben.**

Beispiel 4-3: Codeschnipsel künstlicher Bereich (davor)

```
01 . . .
02 MyData myData;
03 . . .
04 myData.doSomething();
05 . . .
06 doMore();
```

Wollen Sie explizit sicherstellen, dass eine automatische Variable wie `MyData myData`; in dem Codeschnipsel in Beispiel 4-3 vor Zeile 5 ihre Gültigkeit verliert und ihr Destruktor automatisch aufgerufen wird, führen Sie einen künstlichen Bereich wie in Beispiel 4-4 ein.

Beispiel 4-4: Codeschnipsel künstlicher Bereich (danach)

```
01 . . .
02 {
03   MyData myData;
04   . . .
05   myData.doSomething();
```

Beispiel 4-4: Codeschnipsel künstlicher Bereich (danach) (Fortsetzung)

```
06   . . .
07 }
08 doMore();
```

Lock

Aus diesem Grund werden Mutexe in C++11 in Locks gepackt. Diese funktionieren nach dem bekannten C++-RAII-Idiom. RAII steht dabei für *Resource Acquisition Is Initialization*. Wie das RAII-Idiom funktioniert, wird im Anhang C, *Resource Acquisition Is Initialization,* auf Seite 489 erläutert.

lock_guard

`std::lock_guard` und `std::unique_lock` sind das Mittel der Wahl in C++11, wenn es darum geht, den Zugriff auf einen kritischen Bereich durch Threads zu synchronisieren. Beide halten eine Referenz auf einen Mutex. Dabei ist `std::lock_guard` für den einfachen Einsatz ausgelegt, denn es bindet den Mutex in seinem Konstruktor und gibt ihn im Destruktor wieder frei, gemäß RAII-Idiom. Damit lässt sich die Race Condition aus Beispiel 4-1, in dem die Threads unkoordiniert auf die Konsole schreiben, einfach lösen.

Definition: Race Condition

Eine *Race Condition*, oder auch *data race* (kritischer Wettlauf), ist eine Situation, in der mindestens zwei Threads versuchen, gleichzeitig eine gemeinsame Variable zu referenzieren, wobei mindestens einer der beiden Thread diese modifiziert. Damit hängt der Wert der Variablen vom Laufzeitverhalten der Threads ab.

Race Conditions sind schwer auffindbare Fehler, da eine leichte Modifikation des Programms dessen Verhalten vollständig verändern kann.

lockStdout.cpp

Beispiel 4-5: Koordiniertes Schreiben auf die Konsole

```
01 #include <iostream>
02 #include <mutex>
03 #include <string>
04 #include <thread>
05
06 std::mutex coutMutex;
07
08 void helloFunction(const std::string& s){
09
10   // acquire lock
11   std::lock_guard<std::mutex> guard(coutMutex);
12   std::cout << s << std::endl;
```

Beispiel 4-5: Koordiniertes Schreiben auf die Konsole (Fortsetzung)

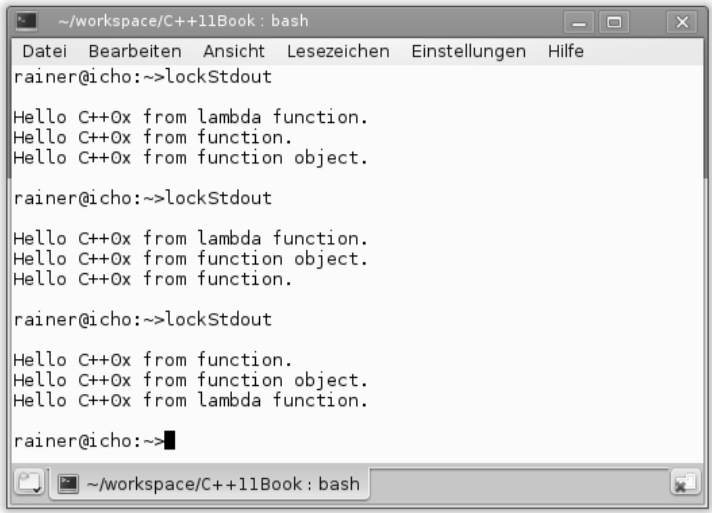
```
13
14 } // release lock automatically
15
16
17 class HelloFunctionObject{
18   public:
19     void operator()(const std::string& s) const {
20
21       // acquire lock
22       std::lock_guard<std::mutex> guard(coutMutex);
23       std::cout << s << std::endl;
24
25     } // release lock automatically
26 };
27
28
29 int main(){
30
31   std::cout << std::endl;
32
33   // thread executing helloFunction
34   std::thread t1(helloFunction,
                   "Hello C++11 from function.");
35
36   // thread executing HelloFunctionObject
37   HelloFunctionObject helloFunctionObject;
38   std::thread t2(helloFunctionObject,
                   "Hello C++11 from function object.");
39
40   // thread executing lambda function
41   std::thread t3([&]{std::lock_guard<std::mutex>
       guard(coutMutex);
       std::cout << "Hello C++11 from lambda function."
                 << std::endl;});
42
43   // ensure that t1, t2 and t3 have finished before main terminates
44   t1.join();
45   t2.join();
46   t3.join();
47
48   std::cout << std::endl;
49
50 }
```

Beispiel 4-5 wartet mit ein paar Neuheiten auf. So wird in Zeile 6 der `coutMutex` angelegt, der durch `std::lock_guard` sowohl von der Funktion (Zeile 11) als auch vom Funktionsobjekt (Zeile 22) und von der Lambda-Funktion (Zeile 41) verwendet wird. Diese Lambda-Funktion ist deutlich anspruchsvoller als alle bisher verwendeten anonymen Funktionen:

- []: Sie bindet den Aufrufkontext per Referenz [&], sodass im Rumpf des Funktionskörpers der Mutex `coutMutex` verwendet werden kann.
- (): Der optionale Argumentbereich () fehlt. Dies ist möglich, da die Lambda-Funktion kein Argument erwartet.
- {}: Ihr Funktionskörper besteht aus zwei Anweisungen.

Die Freigabe des Mutex geschieht automatisch, sodass die drei Aufrufe von `std::lock_guard` für das koordinierte Schreiben nach `std::cout` sorgen.

Abbildung 4-2 ▶
Koordiniertes Schreiben der drei Threads nach std::cout



unique_lock

Der `std::lock_guard` besitzt aber nur eine sehr eingeschränkte Funktionalität. Reicht dieses einfache Interface nicht aus, sollte der `std::unique_lock` verwendet werden. Vereinfacht gesagt, besitzt dieser nicht mehr die strenge 1:1-Beziehung zu seinem Mutex wie `std::lock_guard`. Dieser Aufbruch der engen Assoziation zwischen dem Mutex und seinem Lock besitzt mächtige Auswirkungen auf den `std::unique_guard`. So lassen sich mit ihm Deadlocks elegant verhindern oder zeitliche Bedingungen mit Locks verknüpfen. Die genaueren Details folgen in Teil III, *Multithreading*, auf Seite 223.

Oft ist es nicht nötig, eine Variable während ihres gesamten Lebenszyklus zu schützen, stattdessen muss nur ihre geschützte Initialisierung sichergestellt werden.

Sichere Initialisierung der Daten

Die einfachste Art, Daten geschützt zu initialisieren, sollte nicht vergessen werden, bevor die neuen C++11-Techniken folgen. Das Programm startet im Main-Thread. Daten, die in diesem initialisiert werden, solange noch kein Kind-Thread instanziiert wurde, werden zwangsläufig geschützt initialisiert.

C++11 kennt drei Arten, Variablen geschützt zu initialisieren. Dies sind:

1. Objekte, deren Konstruktor als konstante Ausdrücke (`constexpr`) definiert wurden.
2. Statische Variable mit Block-Gültigkeit.
3. `std::call_once` wird über eine Funktion und ein Flag `std::once_flag` parametrisiert; dabei stellt das Flag sicher, dass die Funktion nur einmal ausgeführt wird.

In Beispiel 4-6 sind alle drei Variationen der Initialisierung von Daten dargestellt.

Beispiel 4-6: Sichere Dateninitialisierung mit C++11 threadingInitialization.cpp

```
01 #include <mutex>
02 #include <thread>
03
04 class MyClass{
05   int i;
06   public:
07
08     constexpr MyClass():i(0){}
09     MyClass(int i_):i(i_){}
10
11 };
12
13 void blockScope(){
14
15   // statically initialized
16   static MyClass myClass(1);
17
18 }
19
20 MyClass* myClass3=nullptr;
21
22 void createInstance(){
23
24   myClass3=new MyClass(2);
25
26 }
27
```

Beispiel 4-6: Sichere Dateninitialisierung mit C++11 (Fortsetzung)

```
28
29 int main(){
30
31   // protected initialized, because of
32
33   // constexpr
34   MyClass myClass;
35
36   // block scope
37   blockScope();
38
39   // threading library functions
40   std::once_flag initFlag;
41   std::call_once(initFlag,createInstance);
42
43 }
```

Durch `constexpr` (Zeile 8) wird der Standardkonstruktoraufruf (Zeile 34) zur Übersetzungszeit ausgeführt. `myClass` (Zeile 16) ist eine statische Variable mit Block-Gültigkeit. In diesem Fall stellt der C++11-Compiler sicher, dass die Funktion nur einmal und atomar ausgeführt wird. Aber auch zur Laufzeit lässt sich eine Variable geschützt initialisieren. Die Funktion `createInstance` (Zeile 22) initialisiert mithilfe des Flags `initFlag` die Variable `myClass3` (Zeile 24) genau einmal.

Schutz von Daten ist aber nur notwendig, wenn diese von den Threads gemeinsam genutzt werden. Thread-lokale Daten verlangen keinen Schutz.

Thread-lokale Daten

Durch das Schlüsselwort `thread_local` wird eine Thread-lokale Variable definiert. Jeder Thread besitzt eine Kopie der Variablen, die an die Lebenszeit des Threads gebunden ist.

Oft reicht es aber nicht aus, dass Threads koordiniert werden, stattdessen ist es notwendig, dass sie synchronisiert auf gemeinsam genutzten Daten arbeiten. Ein Thread kann mit seiner Arbeit erst beginnen, wenn ihm ein anderer Thread das entsprechende Signal sendet.

Synchronisation von Threads

Für die Synchronisation von Threads sollen zwei Anforderungen erfüllt sein:

1. Die Zeit zwischen Arbeit aufnehmendem Thread (Arbeiter) und Signal sendendem Thread (Sender) soll möglichst kurz sein.
2. Das Warten des Arbeiters soll möglichst wenig CPU-Zeit verbrauchen.

Mit dem Lock `std::unique_lock`, der die zu bearbeitenden Daten schützt, der Methode `std::this_thread::sleep_for`, die einen Thread für eine angegebene Zeit schlafen legt, und der neuen Zeitmethode `std::chrono::milliseconds` stehen alle Bausteine bereit, um einen Thread zu implementieren, der durch einen anderen Thread aufgeweckt wird. Ein einfacher Wahrheitswert dient zur Synchronisation der Threads in Beispiel 4-7.

Beispiel 4-7: Einfache Methode für einen Arbeiter-Thread

```
01 std::mutex mutex_;
02 bool dataReady;
03
04 void waitingForWork(){
05
06   std::unique_lock<std::mutex> lck(mutex_);
07
08   while(!dataReady){
09
10     lck.unlock();
11     std::this_thread::sleep_for(
         std::chrono::milliseconds(50));
12     lck.lock(); // need the lock for the while test
13
14   }
15
16   doTheWork(); // require the lock
17
18 }
```

Über den Wahrheitswert `dataReady` signalisiert der Sender, dass die Daten bereit sind. Bevor der Arbeiter den Wahrheitswert prüft und gegebenenfalls seine Arbeit in `doTheWork` (Zeile 16) aufnimmt, setzt er den Lock mit `std::unique_lock` (Zeile 6). Sind die Daten nicht bereit, löst er den Lock, legt sich für 50 Millisekunden schlafen und setzt den Lock wieder, um `dataReady` (Zeile 8) zu testen.

Die Funktion `waitingForWork` (Zeile 4) erfüllt die zwei Anforderungen aber nicht optimal. Zwischen dem Senden des Signals und dem Zeitpunkt, an dem der Worker seine Arbeit aufnimmt, vergehen im Mittel 25 ms (50 ms geteilt durch 2). Zwar lässt sich die Schlafphase einfach verkürzen, indem die Konstante verkleinert wird, dies geht aber auf Kosten der CPU, denn das Sperren und Entsperren des Lock benötigt CPU-Ressourcen.

Beide Bedingungen – kurzes Warten und geringe CPU-Auslastung – lassen sich mit den neuen Bedingungsvariablen in C++11 einfach erfüllen (Beispiel 4-8):

conditionVariable.cpp **Beispiel 4-8:** Sender- und Arbeiter-Thread

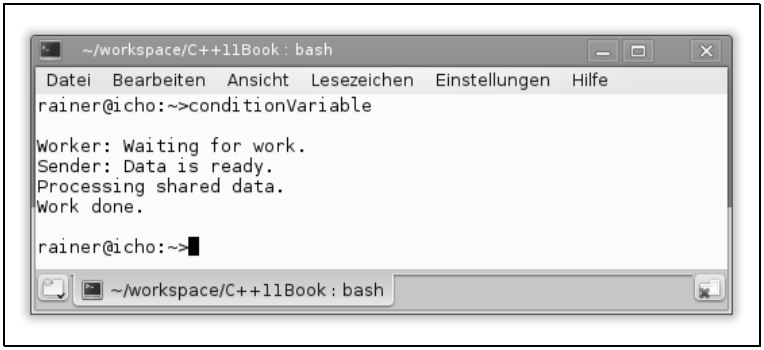
```
01 #include <iostream>
02 #include <condition_variable>
03 #include <mutex>
04 #include <thread>
05
06 std::mutex mutex_;
07 std::condition_variable condVar;
08
09 bool dataReady;
10
11 void doTheWork(){
12   std::cout << "Processing shared data." << std::endl;
13 }
14
15 void waitingForWork(){
16
17     std::cout << "Worker: Waiting for work." << std::endl;
18
19     std::unique_lock<std::mutex> lck(mutex_);
20     condVar.wait(lck,[]{return dataReady;});
21     doTheWork();
22
23     std::cout << "Work done." << std::endl;
24
25 }
26
27 void setDataReady(){
28
29     std::cout << "Sender: Data is ready."  << std::endl;
30
31     std::lock_guard<std::mutex> lck(mutex_);
32     dataReady=true;
33     condVar.notify_one();
34
35 }
36
37 int main(){
```

Beispiel 4-8: Sender- und Arbeiter-Thread (Fortsetzung)

```
38
39   std::cout << std::endl;
40
41   std::thread t1(waitingForWork);
42   std::thread t2(setDataReady);
43
44   t1.join();
45   t2.join();
46
47   std::cout << std::endl;
48
49 }
```

Thread `t1` verwendet die Funktion `setDataReady` (Zeile 27), um dem Thread `t2` zu signalisieren, dass die Daten bereit sind. Durch `condVar.notify_one` (Zeile 33) weckt er den Worker auf. `condVar.wait(lck,[]{return dataReady;}` (Zeile 20) sperrt den Lock, prüft mit der Lambda-Funktion, ob die Bedingung erfüllt ist, und arbeitet `doTheWork` (Zeile 21) ab.

Die Programmausgabe zeigt die Interaktion von Arbeiter und Sender.



◀ **Abbildung 4-3**
Arbeiter- und Sender-Thread in Aktion

Neben `notify_one` kennt die Bedingungsvariable auch die Methode `notify_all`. Damit werden alle Threads, die gerade im Zustand *wait* sind, aufgeweckt.

Ob es die Basiswerkzeuge zum Erzeugen von Threads, zum Koordinieren von Threads wie Lock, zum Synchronisieren von Threads wie Bedingungsvariablen, Thread-lokale Daten oder auch atomare Datentypen waren – dies sind die einfachen Grundwerkzeuge, die jede Threading-Bibliothek mitbringen muss. Komfortabler wird der Umgang mit Threads aber erst, wenn nur die reine Funktionalität spezifiziert werden muss, die im Thread ausgeführt werden soll.

Alle anderen Aspekte rund um das Thread-Handling werden vom System abgenommen. Letztendlich will der Anwender nur das Ergebnis der Tasks abfragen.

Genau diese High-Level-API bietet C++11 mit den asynchronen Tasks.

Asynchrone Aufgaben

Die asynchrone Funktionalität kam relativ spät in den neuen C++11-Standard. Eine asynchrone Aufgabe besteht aus zwei Komponenten:

Promise
- *Promise*: Produziert das Ergebnis in der Regel in einem anderen Thread.

Future
- *Future*: Fordert das Ergebnis des Promise an.

Das Programm in Beispiel 4-5 illustriert, wie viel Tipparbeit investiert werden muss, um drei Threads zu erzeugen, die Ausgaben der Threads nach `std::cout` zu koordinieren und letztendlich mittels `join` zu gewährleisten, dass die Threads ihre Aufgabe vollenden können. Da ein Thread keinen Wert zurückgeben kann, wurde `std::cout` als Ergebniskanal missbraucht. Soll das Programm darüber hinaus die Ergebnisse der Threads in einer definierten Reihenfolge schreiben, müssten wir noch Bedingungsvariablen anwenden. Damit wäre das Programm vollkommen serialisiert und vom Programmablauf einem Single-Threaded-Programm sehr ähnlich.

Ganz schön viel Aufwand. Das geht deutlich einfacher mit `std::async` zum Starten einer asynchronen Aufgabe (Beispiel 4-9).

asyncStdout.cpp **Beispiel 4-9:** Asynchrone Tasks mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion

```
01 #include <future>
02 #include <iostream>
03 #include <string>
04
05 std::string helloFunction(const std::string& s){
06
07   return "Hello C++11 from " + s + ".";
08
09 }
10
11
12 class HelloFunctionObject{
13   public:
```

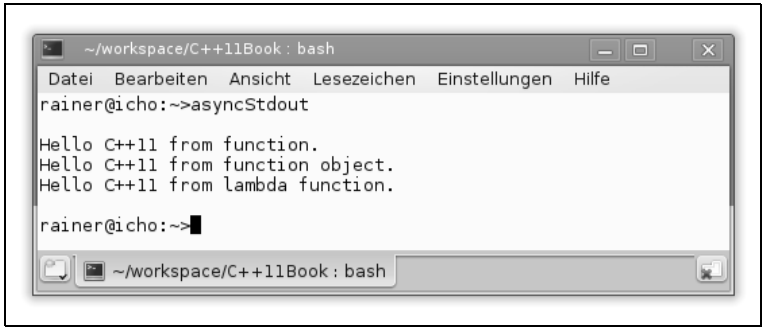
Beispiel 4-9: Asynchrone Tasks mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion (Fortsetzung)

```
14     std::string operator()(const std::string& s) const {
15
16       return "Hello C++11 from " + s + ".";
17
18     }
19 };
20
21 int main(){
22
23   std::cout << std::endl;
24
25   // future with function
26   auto futureFunction= std::async(helloFunction,"function");
27
28   // future with function object
29   HelloFunctionObject helloFunctionObject;
30   auto futureFunctionObject=
       std::async(helloFunctionObject,"function object");
31
32   // future with lambda function
33   auto futureLambda= std::async([](const std::string& s )
      {return "Hello C++11 from " + s + ".";},
       "lambda function");
34
35   std::cout << futureFunction.get() << "\n"
                << futureFunctionObject.get() << "\n"
                << futureLambda.get() << std::endl;
36
37   std::cout << std::endl;
38
39 }
```

Der Aufruf `std::async` lässt sich sowohl über eine Funktion (Zeile 26) als auch über ein Funktionsobjekt (Zeile 29) und eine Lambda-Funktion (Zeile 33) parametrisieren. Der Rückgabewert des Aufrufs, der vom expliziten Typ `std::future<std::string>` ist, wird durch das Schlüsselwort `auto` an die entsprechende Variable gebunden. Mit dem Future lässt sich das Ergebnis des asynchronen Tasks durch den `get`-Aufruf (Zeile 35) abholen. Der `get`-Aufruf eines Future ist blockierend.

Die Ausgabe ist mittlerweile vertraut.

Abbildung 4-4 ▶
Asynchrone Ausgabe nach std::cout



Noch ein paar Worte zu Futures und Promises, die Details folgen in Kapitel 18, *Asynchrone Aufgaben,* auf Seite 283.

Future

Im Gegensatz zu `std::future` aus Beispiel 4-9 bietet `std::shared_future` an, dass das Ergebnis mehrmals angefordert werden kann.

Promise

Neben dem automatischen Starten eines Tasks mit `std::async` ist dies in C++11 auch explizit mit der Funktion `std::thread` möglich. Dazu wird in Beispiel 4-10 ein Promise definiert. Über die `get_future`-Methode des Promise wird ein Future erzeugt und mit dem Promise verbunden. Der neue Thread erhält die Funktion `asyncFunc` und als Parameter den transferierten Promise: `std::move(int Promise)`. Das Ergebnis wird in gewohnter Weise durch den `get`-Aufruf des Future eingefordert.

Beispiel 4-10: Starten eines Promise

```
std::promise<int> intPromise;
std::future<int> intFuture = intPromise.get_future();
std::thread t(asyncFunc, std::move(intPromise));
int result = intFuture.get();
```

Die einzige Unbekannte in Beispiel 4-10 ist nur noch die Funktion `asyncFunc`.

Beispiel 4-11: Funktion mit Promise

```
void asyncFunc(std::promise<int>& intPromise){
  int result;
  try{
    intPromise.set_value(result);
  }
  catch (MyException e) {
    intPromise.set_exception(std::copy_exception(e));
  }
}
```

`asyncFunc` erhält als Argument den Promise. Über seine Methode `set_value` oder gegebenenfalls `set_exception` steht der Rückgabewert für den Future zu Verfügung.

Damit verlassen wir das Feld der neuen Multithreading-Funktionalität in C++11 und kommen zu all den Erweiterungen, die die Standardbibliothek mit sich bringt.