

Modernes C++

<https://www.grimm-jaud.de/index.php/modernes-c-in-der-praxis-linux-magazin-a>

Inhalt

1. Modernes C++ in der Praxis – Folge 1 – Lambda-Funktionen 1	4
Das bessere C++	4
Kompakter Code	5
2. Modernes C++ in der Praxis – Folge 2 – Lambda-Funktionen 2	8
Ohne Umschweife	9
Closure	9
Kopie oder Referenz	10
Closures	12
Problem: Gültigkeit	14
Kurz und knackig	15
Formsache	15
Compilerunterstützung für C++11	16
3. Modernes C++ in der Praxis – Folge 3 – Multithreading 1	16
Rechenaufgabe	17
Zufall beschafft Material	18
Arbeitsteilung	19
Threads erzeugen	21
Erfassen der Variablen	21
Threading vertieft	21
4. Modernes C++ in der Praxis – Folge 4 – Multithreading 2	22
Alle durcheinander	24
Sperren per Mutex	28
Deadlock auflösen	28
Kluge Sperre	29
Wie geht's weiter?	30
5. Modernes C++ in der Praxis – Folge 5 – Multithreading 3	30
Synchronisation per Wahrheitswert	31
Bedingungsvariablen	33
Synchronisation der Arbeiterschaft	34
Arbeitsablauf	37
Wenn es klemmt	37
Wie geht's weiter?	38
6. Modernes C++ in der Praxis – Folge 6 – Multithreading 4	39
Arbeiten auf Zuruf	39
Ein erster Versuch	40

Kommunikationskanal.....	41
Das Meisterwerk.....	41
Wie geht es weiter?.....	44

1. Modernes C++ in der Praxis – Folge 1 – Lambda-Funktionen 1

Von [Rainer Grimm](#)



Die jüngste Reform der Programmiersprache C++ ist abgeschlossen. Das Linux-Magazin widmet dem neuen Standard C++11 eine Artikelserie, die dem Programmierer zeigt, wie er in der Praxis von den Neuerungen profitiert. Die erste Folge stellt Lambda-Funktionen vor.

Nach langer Arbeit trägt der neue C++-Standard jetzt auch einen Namen: Der bisherigen Konvention folgend heißt er C++11, weil er im Jahr 2011 verabschiedet wurde. Die Standardisierungsbehörde ISO bietet die Spezifikation zum Kauf an, Spracherfinder Bjarne Stroustrup hat den letzten kostenlosen Entwurf unter [\[1\]](#) verlinkt. Der Standard präsentiert sich im stolzen Umfang von gut 1300 Seiten. Da war sein Vorgänger von 2003 mit knapp 700 Seiten deutlich schmaler. Bedeutender als der Umfang ist der Inhalt des neuen C++11: Er bietet vieles, was den Einstieg erleichtert und den Profi unterstützt. Der Einsteiger profitiert davon, dass der Compiler seine Datentypen automatisch bestimmt, Klassen einfacher zu definieren sind und sich Daten einheitlich initialisieren lassen. Das ist aber noch lange nicht alles: Dem C++-Novizen gibt der neue Standard außerdem mächtige Bibliotheken an die Hand, die Strings verarbeiten, das Speichermanagement von Variablen automatisieren und die CPUs seines Rechners ausreizen. Das alles erfreut auch den Profi. Doch für ihn beginnt damit das Angebot erst richtig: Er darf sich auf Algorithmen freuen, die sich dank Lambda-Funktionen mit der Leichtigkeit einer Interpreter-Sprache implementieren lassen. Daneben erwarten ihn deutlich leistungsfähigere Werkzeuge zur Template-Programmierung, mit denen er Datenstrukturen für seine Anforderungen maßschneidert. Wer genauer wissen will, welche Features C++11 zu bieten hat, der sei auf die Artikel des Linux-Magazins zur Erweiterung der Kernsprache [\[2\]](#) und zu den neuen Bibliotheken in C++ [\[3\]](#) verwiesen. Wer es noch genauer nachlesen möchte, greift zu Pete Beckers Buch über die Bibliothekserweiterung [\[4\]](#) oder zum Band über die neue Multithreading-Funktionalität von Anthony Williams [\[5\]](#).

Das bessere C++

Diese Artikelserie soll keinen vollständigen Überblick über den neuen C++-Standard bieten. Das ist Aufgabe der oben erwähnten Bücher. Sie möchte alle zwei Monate nur einzelne Komponenten in der Anwendung zeigen und demonstrieren, dass sich der Umstieg auf die moderne Variante lohnt – nicht nur für den C++-Entwickler, denn die Zeichen verdichten sich, dass die Sprache vor einer Renaissance steht [\[6\]](#). Als Einstieg dienen in dieser ersten

Folge die Lambda-Funktionen. In den folgenden Beispielen kommen sie zum Einsatz, um die Elemente eines »std::vector« zu modifizieren. In der Evolution von den C-Funktionen über die C++-Funktionsobjekte bilden die Lambda-Funktionen in C++11 das letzte Glied der Kette ([Abbildung 1](#)).

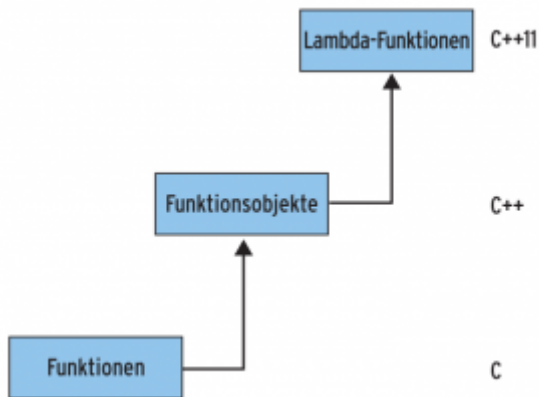


Abbildung 1: Sprachhistorie: Von den Funktionen in C hat die Entwicklung über die Funktionsobjekte in C++ bis zu den Lambda-Funktionen in C++11 geführt.

Kompakter Code

Um Platz zu sparen, setzt das [Listing 1](#) gleich mehrere C++11-Features ein: Die Standardaufgabe, den »std::vector« zu initialisieren, geht in C++11 aufgrund der neuen Initialisierer-Liste »{1,2,3,4, 5,6,7,8,9,10}« in Zeile 13 leicht von der Hand. Die Range-basierte For-Schleife »for (auto v: myVec1)« in Zeile 16 reduziert das Iterieren über einen Container auf die nötigste Schreibarbeit. Zusätzlich ist die automatische Typableitung mit »auto« im Einsatz, die den Typ »int« für »v« ermittelt.

Listing 1

Funktionen

```

01 #include <algorithm>
02 #include <iomanip>
03 #include <iostream>
04 #include <vector>
05
06 void add3(int& i){
07     i +=3;
08 }
09
10 int main(){
11     std::cout << std::endl;
12
13     std::vector<int> myVec1{1,2,3,4,5,6,7,8,9,10};
14     std::cout << std::setw(20) << std::left << "myVec1: i->i+3:      ";
15     std::for_each(myVec1.begin(),myVec1.end(),add3);
16     for (auto v: myVec1) std::cout << std::setw(6) << std::left << v;
17
18     std::cout << "\n\n";
19 }
  
```

Die eigentliche Aufgabe, das Modifizieren des Vektors, löst Zeile 15 in klassischer C-Manier. Um jedes Element eines »std::vector« um 3 zu erhöhen, bietet sich in der Standard Template Library (STL) der Algorithmus »std::for_each« in Kombination mit der Funktion »add3« in

Zeile 15 an. Da jedes Element des Vektors verändert wird, muss das Argument von »add3« als Referenz in »std::for_each« adressiert sein. [Abbildung 2](#) zeigt das Ausführen des Programms in einem Terminalfenster. Soll das Programm jedoch jedes Element um einen beliebigen Wert verändern, endet die Flexibilität einer Funktion – das Mittel der Wahl in C++ heißt Funktionsobjekt. Die Zeilen 20 und 27 in [Listing 2](#) erzeugen ein Funktionsobjekt, das anschließend die Elemente des Containers modifiziert. Kommt ein Funktionsobjekt zum Einsatz, wird dessen überladener Klammeroperator aus Zeile 9 angewandt. In [Abbildung 3](#) ist die Ausgabe des Programms zu sehen.



Listing 2

Funktionsobjekte

```
01 #include <algorithm>
02 #include <iomanip>
03 #include <iostream>
04 #include <vector>
05
06 class AddN{
07 public:
08     AddN(int n):num(n){};
09     void operator()(int& i){
10         i +=num;
11     }
12 private:
13     int num;
14 };
15
16 int main(){
17     std::cout << std::endl;
18
19     std::vector<int> myVec2{1,2,3,4,5,6,7,8,9,10};
20     AddN add4(4);
21     std::for_each(myVec2.begin(),myVec2.end(),add4);
22     std::cout << std::setw(20) << std::left << "myVec2: i->i+4: ";
23     for (auto v: myVec2) std::cout << std::setw(6) << std::left << v;
24     std::cout << "\n";
25
26     std::vector<int> myVec3{1,2,3,4,5,6,7,8,9,10};
27     AddN addMinus5(-5);
28     std::for_each(myVec3.begin(),myVec3.end(),addMinus5);
29     std::cout << std::setw(20) << std::left << "myVec3: i->i-5: ";
30     for (auto v: myVec3) std::cout << std::setw(6) << std::left << v;
31
32     std::cout << "\n\n";
33 }
```

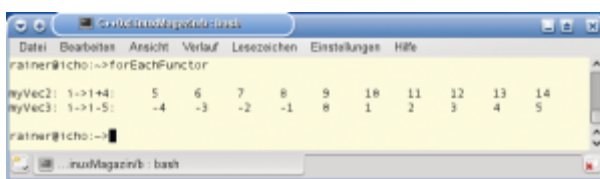


Abbildung 3: Die Elemente des Vektors lassen sich auch durch die typische C++-Vorgehensweise mit Funktionsobjekten manipulieren.

Ist beliebige Funktionalität gewünscht, kommt aber auch die Leistungsfähigkeit der Funktionsobjekte an ihr Ende. Nun schlägt die Stunde der Lambda-Funktionen. Eine Lambda-Funktion, die direkt den Algorithmus »std::for_each« parametrisiert, kann beliebige Funktionalität umsetzen. In [Listing 3](#) modifiziert in Zeile 13 die Lambda-Funktion »[](int& i){i=3*i+5;}« die Elemente des Vektors. Dabei leitet »[]« die Lambda-Funktion ein, »int& i« bezeichnet ihr Argument und »{i=3*i+5;}« stellt ihren Funktionskörper dar.

Listing 3

Lambda-Funktionen

```
01 #include <algorithm>
02 #include <cmath>
03 #include <iomanip>
04 #include <iostream>
05 #include <vector>
06
07
08 int main() {
09     std::cout << std::endl;
10
11     std::vector<int> myVec4{1,2,3,4,5,6,7,8,9,10};
12     std::cout << std::setw(20) << std::left << "myVec4: i->3*i+5: ";
13     std::for_each(myVec4.begin(), myVec4.end(), [](int& i){i=3*i+5;});
14     for (auto v: myVec4) std::cout << std::setw(6) << std::left << v;
15     std::cout << "\n";
16
17     std::vector<int> myVec5{1,2,3,4,5,6,7,8,9,10};
18     std::cout << std::setw(20) << std::left << "myVec5: i->i*i ";
19     std::for_each(myVec5.begin(), myVec5.end(), [](int& i){i=i*i;});
20     for (auto v: myVec5) std::cout << std::setw(6) << std::left << v;
21     std::cout << "\n";
22
23     std::vector<double> myVec6{1,2,3,4,5,6,7,8,9,10};
24     std::for_each(myVec6.begin(), myVec6.end(), [](double&
25 i){i=std::sqrt(i);});
26     std::cout << std::setw(20) << std::left << "myVec6: i->sqrt(i): ";
27     for (auto v: myVec6) std::cout << std::fixed << std::setprecision(2)
28 << std::setw(6) << v ;
29     std::cout << "\n\n";
30 }
```

Ähnlich kompakt gestalten sich die Lambda-Funktionen »[](int& i){i=i*i;}« in Zeile 19, die jedes Element des Vektors auf sein Quadrat, sowie die »[](double& i){i=sqrt(i);}« in Zeile 24, die jedes Element des Vektors auf seine Wurzel abbildet. [Abbildung 4](#) zeigt die Ausgabe des Programmcodes.

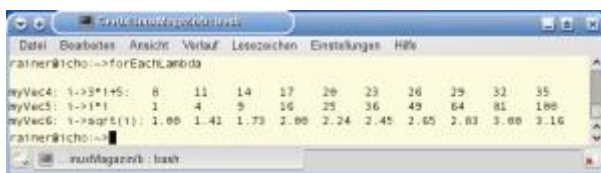


Abbildung 4: Mit Lambda-Funktionen kann der Programmierer den For-Each-Algorithmus parametrisieren.

Nach dieser Einführung bietet der nächste Artikel etwas anspruchsvollere Lektüre, denn zu diesem Feature gibt es noch einige Fragen zu beantworten: Wie funktioniert eine Lambda-Funktion? Wie bindet sie ihren aufrufenden Bereich? Wann muss der Programmierer den

Rückgabebetyp einer Lambda-Funktion angeben? Wann soll ein Software-Entwickler Lambda-Funktionen einsetzen? Es lohnt sich, die Lambda-Funktionen noch genauer zu studieren. Wer sie gut versteht, kann sie in Threads einsetzen oder auch in Form einer so genannten Closure [7]. Die kommende Folge dieser Serie wird sie daher an weiteren Beispielen vorführen. (mhu) Feedback erwünscht

Fehlt Ihnen ein wichtiges C++11-Feature? Hat der Artikel einen wichtigen Aspekt übergangen? Unter <mailto:cpp@linux-magazin.de> erreichen Sie den Autor Rainer Grimm sowie die Redaktion. Ihr Feedback ist herzlich willkommen!

Infos

1. Stroustrups FAQ zu C++0x:
<http://www2.research.att.com/~bs/C++0xFAQ.html#WG21>
2. Rainer Grimm, "Erfrischend Neu": Linux-Magazin 04/10, S. 116
3. Rainer Grimm, "Reichhaltiges Angebot": Linux-Magazin 05/10, S. 110
4. Pete Becker, "The C++ Standard Library Extension": Addison-Wesley 07/2006
5. Anthony Williams, "C++ Concurrency in Action": Manning Publication 02/2010
6. Marius Bancila, "C++ Renaissance at Microsoft":
<http://mariusbancila.ro/blog/2011/06/20/cpp-renaissance-at-microsoft/>
7. Closure: http://en.wikipedia.org/wiki/Closure_%28computer_science%29

[Aus Linux-Magazin 02/2012](#)

2. Modernes C++ in der Praxis – Folge 2 – Lambda-Funktionen 2

Von [Rainer Grimm](#)



Lambda-Funktionen sind die praktischen Helfer der Sprache C++11. Schon nach kurzer Zeit möchte kein C++-Entwickler sie missen, denn mit ihnen ist ein Algorithmus rasch und ohne Umschweife formuliert. Außerdem darf er sie wie Objekte behandeln.

Hinter dem griechischen Buchstaben Lambda verbergen sich in C++11 besondere Funktionen: Funktionen ohne Namen. Als anonyme Ausdrücke verbinden sie das Aufrufverhalten einer Funktion mit einem Gedächtnis.

Ohne Umschweife

Nichts Neues, dürfte mancher C++-Entwickler erwidern, das kann ein Funktionsobjekt [\[1\]](#) auch. Stimmt, denn eine Lambda-Funktion ist nur Syntactic Sugar [\[2\]](#) für ein Funktionsobjekt, aber von der süßesten Art. Denn um ein Funktionsobjekt zu verwenden, sind zwei zusätzliche Arbeitsschritte erforderlich: Zum einen muss der Entwickler die Klasse eines Funktionsobjekts definieren, zum anderen es auch instanzieren.

Diesen insgesamt drei Schritten steht der direkte Einsatz der Lambda-Funktion gegenüber. Dabei folgt diese Funktion der einfachen Struktur in [Abbildung 1](#): Sie besteht aus vier Komponenten. Die eckigen Klammern »[]« besitzen eine Doppelfunktion: Zum einen leiten sie die anonyme Funktion ein, zum anderen kann der Programmierer zwischen ihnen den aufrufenden Kontext erfassen.

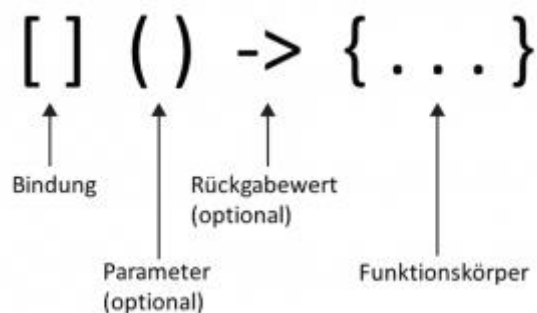


Abbildung 1: Die Notation einer Lambda-Funktion in C++11. Einige Bestandteile darf der Programmierer weglassen.

Weiter geht es mit den runden Klammern »()«. Sie erklären – in Analogie zu einer Funktionsdefinition – die Parameter der Funktion. Erwartet die Lambda-Funktion keine Argumente, so dürfen die runden Klammern entfallen. Entfallen kann auch die Angabe des Rückgabetyps »->«, falls die Lambda-Funktion keinen Wert zurückgibt oder der einfachen Struktur »return *Ausdruck*;<« folgt. Dabei steht der Ausdruck »->int« für einen Rückgabotyp »int« in der neuen, alternativen Funktionssyntax, die bei Lambda-Funktionen obligatorisch ist. Zuletzt folgt in geschweiften Klammern »{ }« der Funktionskörper.

Eine Lambda-Funktion in C++11 kann die Variablen des aufrufenden Kontexts erfassen. Damit besitzt sie ein Gedächtnis und stellt eine Closure dar (siehe [Kasten "Closure"](#)). Dieses Erfassen der Variablen geschieht per Kopie oder per Referenz. Der Entwickler entscheidet nach dem Einsatzzweck, welche Variante er benutzt. In diesem Punkt unterscheidet sich die Lambda-Funktion nicht von einer herkömmlichen Funktion.

Closure

Eine Closure beziehungsweise ein Funktionsabschluss ist eine Funktion, die ihren Erzeugungskontext konservieren kann. Damit erlaubt es eine Closure, Variablen in ihrem Funktionskörper zu binden, sodass sie bei der Ausführung der Funktion zur Verfügung stehen. Dabei wird nicht nur der Wert der Variablen konserviert, sondern auch ihre Lebenszeit verlängert.

Kopie oder Referenz

[Listing 1](#) zeigt die beiden Alternativen: Das Beispielprogramm initialisiert den String »copy« (Zeile 6) sowie den String »ref« (Zeile 7) mit dem gleichen Wert »"original"« . Die Lambda-Funktion in Zeile 8 erfasst »copy« per Kopie und »ref« per Referenz. Beim ersten Ausführen der Lambda-Funktion in Zeile 9 sind die beiden Werte unverändert. Wird der Wert der beiden Strings in den Zeilen 10 und 11 auf »"changed"« gesetzt, zeigt sich der Unterschied: Nur die per Referenz eingebundene Variable »ref« gibt beim nochmaligen Ausführen der Funktion »lambda()« in Zeile 12 die Änderung des Wertes wieder ([Abbildung 2](#)). Weitere Möglichkeiten, den Kontext zu binden, beschreibt der [Kasten "Aufrufkontext erfassen"](#).

Aufrufkontext erfassen

Eine Lambda-Funktion kann ihren aufrufenden Kontext auf drei Arten erfassen: Die leeren eckigen Klammern »[]« verzichten auf den Zugriff. Das kaufmännische Und-Zeichen »&[]« referenziert alle Variablen, die in der Lambda-Funktionen verwendet werden. Der dritte Modus schreibt sich » [=] « und kopiert alle verwendeten Variablen.

Neben diesen Defaultmodi sind auch alle denkbaren Kombinationen möglich. So bewirkt beispielsweise » [= , & var] « , dass C++11 standardmäßig alle in der Lambda-Funktion verwendeten Variablen kopiert, die Variable »var« hingegen referenziert.

Listing 1

Kopie und Referenz

```
01 #include <iostream>
02
03 int main(){
04     std::cout << std::endl;
05
06     std::string copy= "original";
07     std::string ref= "original";
08     auto lambda=[copy,&ref]{std::cout << copy << " " << ref <<
std::endl;};
09     lambda();
10     copy="changed";
11     ref= "changed";
12     lambda();
13
14     std::cout << std::endl;
15 }
```

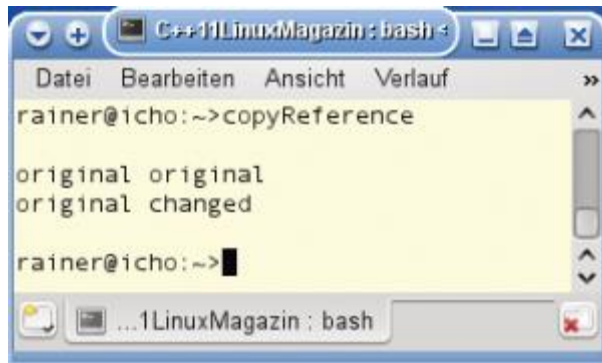


Abbildung 2: Eine Lambda-Funktion erfasst Variablen durch Kopie (erste Ergebniszeile) oder durch Referenz (zweites Ergebnis).

Wer die besonderen Regeln zum Erfassen von Variablen verinnerlicht, kann eine »join()« - Funktion in C++11 rasch umsetzen. Die Funktion nimmt einen Vektor von Strings »str« und einen String »sep« als Argumente an, verbindet die Elemente des Vektors mit dem Separator und gibt den resultierenden String als Ergebnis zurück. Wer sich an die gleichnamige Python-Funktion erinnert fühlt, liegt vollkommen richtig.

Die Anwendung der Funktion in [Listing 2](#) ist schnell erklärt: Enthält der Vektor mehr als einen String, verbindet der Trenner dessen Elemente in den Zeilen 22, 25, 28, 31 und 34. Der Defaultwert für den Separator ist der leere String »""« (Zeile 6). Die Ausgaben des Programms zeigt [Abbildung 3](#).

Listing 2

Join-Funktion

```

01 #include <algorithm>
02 #include <iostream>
03 #include <string>
04 #include <vector>
05
06 std::string join(std::vector<std::string>& str, std::string sep=""){
07
08     std::string joinStr="";
09     if (not str.size()) return joinStr;
10     std::for_each(str.begin(), str.end()-1,
11         [&joinStr, sep](std::string v) {joinStr+= v + sep;});
12     joinStr+= str.back();
13     return joinStr;
14
15 }
16
17 int main(){
18     std::vector<std::string> myVec;
19     std::cout << join(myVec) << std::endl;
20
21     myVec.push_back("One");
22     std::cout << join(myVec) << std::endl;
23
24     myVec.push_back("Two");
25     std::cout << join(myVec) << std::endl;
26
27     myVec.push_back("Three");

```

```

28  std::cout << join(myVec,":") << std::endl;
29
30  myVec.push_back("Four");
31  std::cout << join(myVec,"/") << std::endl;
32
33  myVec.push_back("Five");
34  std::cout << join(myVec,"XXX") << std::endl;
35
36  std::cout << std::endl;
37 };

```

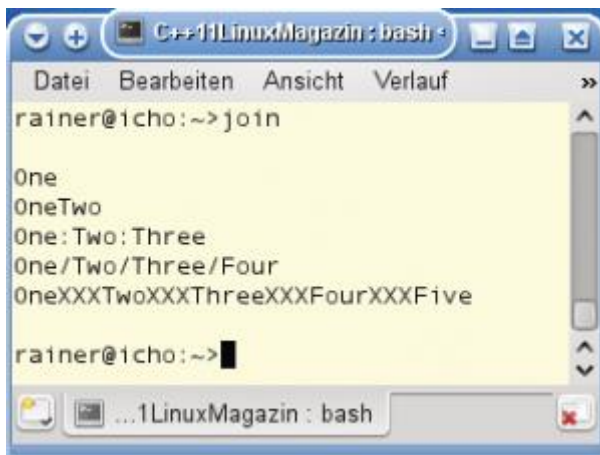


Abbildung 3: Mit der anonymen Funktion ist schnell ein Programm geschrieben, das die Elemente eines Vektors aufreiht und mit Separatorzeichen trennt.

Die Funktion »join()« ist eine genauere Betrachtung wert: Die Variable »joinStr« in Zeile 8 stellt den zusammengefügte String dar. Ist der Vektor leer (Zeile 9), kommt ein leerer String als Ergebnis zurück. Das Zusammenfügen des neuen Strings findet in Zeile 11 statt, in der Lambda-Funktion »[&joinStr,sep](std::string v) {joinStr+= v + sep;}« des »std::for_each« - Algorithmus.

Dabei verarbeitet der Algorithmus jedes Element »v« des Vektors mit Ausnahme des letzten Strings (»str.begin,str.end()-1«), indem er den String »v« und den Trenner »sep« an den resultierenden String »joinStr« anhängt: »joinStr+= v + sep« .

Zum Abschluss erweitert Zeile 12 die Variable »joinStr« um den letzten String »str.back()« . Dieser Schritt findet auch statt, wenn der Vektor nur aus einem einzigen String besteht. Damit der Funktionskörper der Lambda-Funktion nicht in jedem Schritt von »std::for_each« versucht eine Kopie von »joinStr« zu modifizieren, ist es notwendig, dass sie die Zeichenkette per »[&joinStr,sep]« als Referenz erfasst.

Closures

Eine »join()« -Funktion in C++11 ist bei Weitem noch nicht das Ende der Geschichte. Eine Lambda-Funktion kann der Programmierer nicht nur als Closure verwenden, die ihren Erzeugungskontext erfasst – er darf sie auch wie ein Objekt kopieren. Verknüpft er diese beiden Eigenschaften, erhält er eine besondere Funktion wie beispielsweise »inRange()« (Zeile 6) in [Listing 3](#), die auf Anfrage eine Funktion erzeugt.

Listing 3

Einfaches Erzeugen eines Filters

```

01 #include <algorithm>
02 #include <functional>
03 #include <iostream>
04 #include <random>
05
06 std::function<bool(int)> inRange(int low, int up){
07     return [low,up](int d){return d >= low and d <= up;};
08 }
09
10 int main(){
11     std::cout << std::boolalpha << std::endl;
12
13     std::cout << "4 inRange(5,10): " << inRange(5,10)(4) << std::endl;
14     auto filt= inRange(5,10);
15     std::cout << "5 inRange(5,10): " << filt(5) << std::endl;
16
17     std::cout << std::endl;
18
19     const int NUM=60;
20     std::random_device seed;
21
22     // generator
23     std::mt19937 engine(seed());
24
25     // distribution
26     std::uniform_int_distribution<int> six(1,6);
27
28     std::vector<int> dice;
29     for ( int i=1; i<= NUM; ++i) dice.push_back(six(engine));
30
31     std::cout << std::count_if(dice.begin(),dice.end(),inRange(6,6)) << "
of " << NUM << " inRange(6,6) " << std::endl;
32     std::cout << std::count_if(dice.begin(),dice.end(),inRange(4,6)) << "
of " << NUM << " inRange(4,6) " << std::endl;
33
34     // remove all elements for 1 to 4
35     dice.erase(std::remove_if(dice.begin(),dice.end(),inRange(1,4)),dice.end())
;
36     std::cout << "All numbers 5 and 6: ";
37     for (auto v: dice ) std::cout << v << " ";
38
39     std::cout << "\n\n";
40 }

```

Die Funktion »inRange()« erwartet zwei Argumente und gibt wiederum eine Funktion zurück. Sie hat den Typ »std::function<bool(int)>« und zeichnet sich dadurch aus, dass sie eine natürliche Zahl annimmt und als Ergebnis einen Wahrheitswert zurückgibt. Derartige Funktionen, die einen Wahrheitswert zurückgeben, werden Prädikat genannt und sind in der Standard Template Library häufig im Einsatz.

So funktioniert das Ganze: Wird die Funktion »inRange(5,10)« aufgerufen, führt dies dazu, dass ihre Lambda-Funktion teilweise evaluiert wird. Der Ausdruck »inRange(5,10)« bindet die Werte von »low« =5 und »up« =10, denn die Lambda-Funktion »[low,up](int d){return d >= low and d <= up;}« ist eine Closure. Das Ergebnis von »inRange(5,10)« ist die Funktion »[](int d){return d >= 5 and d <= 10;}« . Das Ergebnis von »inRange(5,10)(4)« ist »false« ,

denn »4« in die Lambda-Funktion »[](int d){return d >= 5 and d <= 10;}« eingesetzt, ergibt »false« .

Zeile 14 von [Listing 3](#) bindet die anonyme Funktion an einen Namen, damit sie sich wie eine gewöhnliche Funktion verwenden lässt. Die Mächtigkeit von »inRange()« zeigt sich am besten, wenn man die Funktion auf beliebige Zahlen als Filter anwendet.

Um geeignetes Zahlenmaterial zu erzeugen, regiert in den Zeilen 20 bis 29 der Zufall. Mit der neuen C++11-Funktionalität initialisiert Zeile 23 den Zufallszahlenerzeuger mit einem zufälligen Startwert »seed()« . Die resultierenden Zufallszahlen verteilen sich gleichmäßig auf die Zahlen von 1 bis 6 (Zeile 26). Zum Abschluss wird der virtuelle Würfel noch 60-mal in Zeile 29 geworfen, der Vektor »dice« speichert alle Ergebnisse.

Nun ist es Zeit für die Statistik: Die Aufrufe von »inRange()« in Zeile 31, 32 und 35 erzeugen die Prädikate, die das Programm sogleich anwendet. Damit ist schnell ermittelt, wie oft die 6 (Zeile 31) oder wie oft die 5 und die 6 (Zeile 32) gewürfelt wurden. Natürlich lässt sich »inRange()« auch wie in Zeile 35 einsetzen, um alle Ergebnisse mit ein bis vier Würfelaugen zu löschen.

[Listing 3](#) verwendet Lambda-Funktionen (Zeile 7) und Closures sowie Funktionen als Objekte erster Klasse, die sich beispielsweise kopieren lassen (Zeile 14). Bei »inRange()« in Zeile 6 handelt es sich um eine Funktion höherer Ordnung, da sie eine Funktion zurückgibt. Damit bewegt sich das Beispiel tief in der funktionalen Programmierung.

Problem: Gültigkeit

Ein Problem, das der Programmierer beim Einsatz von Lambda- oder auch anonymen Funktionen im Auge behalten muss, ist die Gültigkeit der verwendeten Variablen. Gefahr ist dann im Spiel, wenn die Variablen per Referenz erfasst werden. So quittierte der Rechner des Autors das Ausführen des Programms in [Listing 4](#) mit einem Speicherzugriffsfehler. Der Grund: Die Funktion »makeLambda()« in Zeile 4 erzeugt eine Lambda-Funktion. Diese gibt den String »val« zurück.

Listing 4

Ungültige Variable

```
01 #include <functional>
02 #include <iostream>
03
04 std::function<std::string()> makeLambda() {
05     const std::string val="on stack created";
06     return [&val]{return val;};
07 }
08
09 int main(){
10     auto bad= makeLambda();
11     std::cout << bad();
12 }
```

Kurz und knackig

Die Einsatzgebiete von Lambda-Funktionen überschneiden sich oft mit denen von Funktionen und Funktionsobjekten. Eine einfache Faustregel für die Anwendung von Lambda-Funktionen ist, ob sich deren Funktionskörper kurz und knackig niederschreiben lässt. Und das ist dann der Fall, wenn folgende Kriterien für die Lambda-Funktion zutreffen:

- Der Funktionskörper besteht nur aus einem einfachen Ausdruck.
- Die Lambda-Funktion wird nicht mehrmals definiert und angewandt.
- Der Einsatz und die Funktionalität der Lambda-Funktion sind selbsterklärend, sodass keine umfangreiche Dokumentation erforderlich ist.

Das ist nichts Ungewöhnliches, doch Vorsicht: Die Lambda-Funktion erfasst »val« mit »[&val]« per Referenz. Die Gültigkeit von »val« endet aber mit der Ausführung der Funktion »makeLambda()«. Damit ist das Ergebnis des Aufrufs von »bad()« in Zeile 11 und somit das ganze Programm undefiniert. Es zeigt unvorhersagbares Verhalten [\[3\]](#).

Diese Problematik verschärft sich bei in Threads eingesetzten Lambda-Funktionen. Erfasst eine derartige Funktion in einem Thread eine Variable per Referenz, gilt es, neben der Gültigkeit den Schutz der Variablen zu beachten. So viel sei als Appetitanreger auf die nächste Folge verraten: In der neuen Threading-Funktionalität von C++11 werden Lambda-Funktion gerne verwendet, um die Arbeitspakete für die Threads zu schnüren.

Formsache

Ein Tipp noch zum Schluss: In Lambda-Funktionen sollte sich der Programmierer kurz fassen (siehe [Kasten “Kurz und knackig”](#)). Wer dennoch komplexere Lambda-Funktionen einsetzen möchte, sollt das gewohnte Layout einer C-Funktion verwenden. Obwohl sie unterschiedlich aussehen, bieten die beiden Lambda-Funktionen in [Listing 5](#) die gleiche Funktionalität: Sie berechnen die Summe und das Produkt aller Elemente des Containers »val« .

Listing 5

Layout-Varianten

```
01 int sum= 0;
02 int prod= 1;
03 std::for_each(val.begin(),val.end(),[&](int v){sum+= v; prod*= v;
std::cout << sum << " " << prod << std::endl;});
04
05 sum= 0;
06 prod= 1;
07 std::for_each(val.begin(),val.end(),
08             [&](int v){
09                 sum+= v;
10                 prod*= v;
11                 std::cout << sum << " " << prod << std::endl;
12             });
```

Ersetzt man in der zweiten Lambda-Funktion in Zeile 8 den Ausdruck »[&]« durch einen Funktionsnamen und den Rückgabewert »void calcSumAndProduct«, dann wird die syntaktische Verwandtschaft der Lambda- zur normalen Funktion offensichtlich. (*mhu*)

Compilerunterstützung für C++11

Die Compilerunterstützung der C++11-Features ist recht weit fortgeschritten, ein aktueller Compiler ist aber in jedem Fall erforderlich [4]. Der aktuelle GCC 4.6 hat bei der Umsetzung von C++11 die Nase vorn [5]. Mit ihm lassen sich nahezu alle Features der Kernsprache sowie die neuen und verbesserten Bibliotheken in Aktion beobachten. Lediglich bei der Bibliothek für die regulären Ausdrücke muss der Anwender auf Boost zurückgreifen.

Beim Compiler »g++« muss der Anwender C++11 explizit einschalten. Das bewirkt ein Aufruf in der Form »g++ -std=c++0x lambda.cpp -o lambda«. GCC 4.7 erlaubt es, den Quelltext mit »g++ -std=c++11 lambda.cpp -o lambda« zu übersetzen.

In Microsofts Visual C++ Compiler (VC10) dagegen ist der C++11-Standard von Haus aus aktiviert. Auch die Regular Expressions sind schon eingebaut [6].

Infos

1. Rainer Grimm, "Die Elf spielt auf": Linux-Magazin 12/11, S. 94
2. Syntactic Sugar: http://en.wikipedia.org/wiki/Syntactic_sugar
3. undefiniertes Verhalten: http://en.wikipedia.org/wiki/Undefined_behavior
4. C++11-Unterstützung: <http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport>
5. C++11-Unterstützung für den GCC-Compiler: <http://gcc.gnu.org/projects/cxx0x.html>
6. C++11-Unterstützung für den Visual C++ Compiler: <http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx>

[Aus Linux-Magazin 04/2012](#)

3. Modernes C++ in der Praxis – Folge 3 – Multithreading 1

Von [Rainer Grimm](#)



Mit seiner Multithreading-Fähigkeit hilft C++11 dem Programmierer dabei, von der Leistung moderner Mehrkern- und Mehrprozessorsysteme zu profitieren. Dieser Artikel führt in die mehrgleisige Datenverarbeitung ein und zeigt, dass die ersten eigenen Threads rasch programmiert sind.

Die wegweisenden Artikel "The Free Lunch is Over" [1] und "Welcome to the Jungle" [2] der C++-Koryphäe Herb Sutter bringen ein Problem moderner Software-Entwicklung auf den Punkt: Der Gleichschritt bei Performanceverbesserung der CPU und Anforderungen der Software ist nach 30 Jahren merklich aus dem Takt geraten. Es reicht nicht mehr aus, einfach auf eine schnellere CPU zu setzen. In der modernen Hardwarewelt müssen Programmiersprachen mit homogenen, ja sogar mit heterogenen Multicore-Architekturen umgehen können. C++11 schafft das mit Hilfe der neuen Multithreading-Funktionalität.

Rechenaufgabe

Bevor diese Artikelserie auf deren verzwickte Details eingeht, soll zuerst ein einfaches Beispiel den Respekt vor dem Unbekannten nehmen. Das Skalarprodukt zweier gleich großer Vektoren »v« und »w« der Länge »n+1« ist (vereinfacht) dadurch definiert, dass die einzelnen Werte »v[i]« und »w[i]« multipliziert und deren Produkte addiert werden:

$$v[0]*w[0] + v[1]*w[1] + \dots v[n]*w[n]$$

Wem diese Erklärung nicht formal genug ist, der sei auf [3] verwiesen. Die Berechnung erledigt in Listing 1 der Algorithmus »std::inner_product()« aus der Standard Template Library [4].

Listing 1

Sequenzielles Berechnen des Skalarprodukts

```
01 #include <chrono>
02 #include <iostream>
03 #include <random>
04 #include <vector>
05
06 static const int NUM= 10000000;
07
08 long long getDotProduct(std::vector<int>& v, std::vector<int>& w) {
09     return std::inner_product(v.begin(), v.end(), w.begin(), 0LL);
10 }
11
12 int main() {
13
14     std::cout << std::endl;
15
16     // get NUM random numbers from 0 .. 100
17     std::random_device seed;
18
19     // generator
20     std::mt19937 engine(seed());
21
22     // distribution
23     std::uniform_int_distribution<int> dist(0,100);
24
25     // fill the vectors
26     std::vector<int> v, w;
27     v.reserve(NUM);
28     w.reserve(NUM);
29     for (int i=0; i< NUM; ++i){
30         v.push_back(dist(engine));
31         w.push_back(dist(engine));
32     }
33
34     // measure the execution time
```

```

35  std::chrono::system_clock::time_point start =
std::chrono::system_clock::now();
36  std::cout << "getDotProduct(v,w): " << getDotProduct(v,w) <<
std::endl;
37  std::chrono::duration<double> dur = std::chrono::system_clock::now()
- start;
38  std::cout << "Sequential Execution: " << dur.count() << std::endl;
39
40  std::cout << std::endl;
41
42 }

```

Das abgedruckte Programm berechnet das Skalarprodukt zweier Vektoren mit 10 Millionen Elementen. Die Rechenarbeit findet in der Funktion »getDotProduct()« in Zeile 8 statt. Ungewöhnlich ist nur die Initialisierung des Rückgabewerts »0LL«. Dieses C++11-Literal steht für die Zahl »long long 0« und stellt sicher, dass das Ergebnis der summierten Multiplikationen in den Rückgabotyp passt. Diese ungewöhnlich große Variable ist notwendig, denn das Ergebnis des Skalarprodukts ([Abbildung 1](#)) bewegt sich in einem sehr hohen Bereich: Der Datentyp »long long« kann Zahlen im Wertebereich von mindestens -9 223 372 036 854 755 808 bis +9 223 372 036 854 755 807 (mehr als 9 Trillionen) darstellen. Einen schönen Überblick über Datentypen für verschieden große natürliche Zahlen gibt der Wikipedia-Artikel unter [\[5\]](#).



Abbildung 1: Berechnen des Skalarprodukts durch einen einzigen Thread.

Zufall beschafft Material

Um die Vektoren in den Zeilen 30 und 31 ([Listing 1](#)) mit Werten für das Rechenbeispiel zu füllen, initialisiert Zeile 20 den Zufallszahlengenerierer mit einem zufälligen Startwert. Dieser produziert gleich verteilte Zufallszahlen im Bereich von 0 bis 100. Die Zeilen 27 und 28 stellen mit »v.reserve(NUM)« sicher, dass der Speicher für den Vektor in einem Rutsch reserviert wird. Das ergibt bei Vektoren dieser Länge durchaus Sinn, denn es erspart das aufwändige mehrfache Reservieren von Speicher. Zeile 36 veranlasst die Berechnung und gibt das Skalarprodukt der zwei Vektoren aus. Dank der neuen Zeitbibliothek »chrono« ist ziemlich einfach zu bestimmen, wie lange das Ausführen der Funktion dauert. Zeile 35 ermittelt die Zeit vor dem Funktionsaufruf, Zeile 37 stellt die Zeit danach fest und berechnet die Zeitdifferenz. Wer eine kompakte Darstellung der neuen Zeitbibliothek sucht, wird in der Webpräsenz von Anthony Williams [\[6\]](#) fündig. [Abbildung 1](#) zeigt, wie lange der PC des Autors zum Berechnen des Skalarprodukts braucht.

Arbeitsteilung

Das sollte deutlich schneller gehen, da die Ausführungszeit von der Rechenpower der Hardware abhängt. Immerhin steht ein Vierkernsystem unter dem Schreibtisch. Bei der ersten Implementierung hatte ein einziges Core die ganze Arbeit zu erledigen, während die drei verbleibenden auf Beschäftigung warteten. Dabei lässt sich das Verarbeiten eines Vektors leicht in vier gleich große Arbeitspakete aufteilen, sodass auch das Skalarprodukt etwa viermal so schnell berechnet sein sollte ([Abbildung 2](#)).



Abbildung 2: Der Einsatz von vier parallelen Threads verkürzt die Laufzeit deutlich.

Der Code in [Listing 2](#) verteilt in der Funktion »getDotProduct()« die Berechnung des Skalarprodukts auf vier Threads. Dazu zerschneidet er die zu verarbeitenden Vektoren in vier Portionen. Grafisch ist die Aufteilung in [Abbildung 3](#) dargestellt.

Listing 2

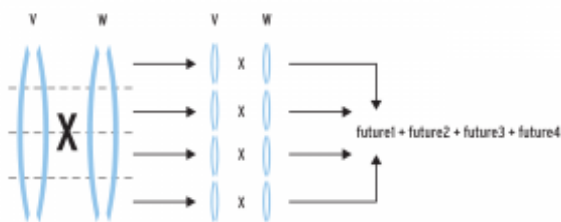
Paralleles Berechnen des Skalarprodukts

```
01 #include <chrono>
02 #include <iostream>
03 #include <future>
04 #include <random>
05 #include <thread>
06 #include <vector>
07
08 static const int NUM= 10000000;
09
10 long long getDotProduct(std::vector<int>& v, std::vector<int>& w){
11
12     auto future1= std::async(
13         [&]{return std::inner_product(&v[0],&v[v.size()/4],
14                                         &w[0],0LL);}
15     );
16
17     auto future2= std::async(
18         [&]{return std::inner_product(&v[v.size()/4],&v[v.size()/2],
19                                         &w[v.size()/4],0LL);}
20     );
21
22     auto future3= std::async(
23         [&]{return std::inner_product(&v[v.size()/2],&v[v.size()*3/4],
24                                         &w[v.size()/2],0LL);}
25     );
26
27     auto future4= std::async(
28         [&]{return std::inner_product(&v[v.size()*3/4],&v[v.size()],
29                                         &w[v.size()*3/4],0LL);}
29     );
```

```

30         );
31     return future1.get() + future2.get() + future3.get() + future4.get();
32 }
33
34
35 int main(){
36     std::cout << std::endl;
37
38     // get NUM random numbers from 0 .. 100
39     std::random_device seed;
40
41     // generator
42     std::mt19937 engine(seed());
43
44     // distribution
45     std::uniform_int_distribution<int> dist(0,100);
46
47     // fill the vectors
48     std::vector<int> v, w;
49     v.reserve(NUM);
50     w.reserve(NUM);
51     for (int i=0; i< NUM; ++i){
52         v.push_back(dist(engine));
53         w.push_back(dist(engine));
54     }
55
56     // measure the execution time
57     std::chrono::system_clock::time_point start =
58     std::chrono::system_clock::now();
59     std::cout << "getDotProduct(v,w): " << getDotProduct(v,w) <<
60     std::endl;
61     std::chrono::duration<double> dur = std::chrono::system_clock::now()
62     - start;
63     std::cout << "Parallel Execution: "<< dur.count() << std::endl;
64
65     std::cout << std::endl;
66 }

```



Das Übersetzen von [Listing 2](#) ist ein wenig aufwändiger als im vorigen Fall: Die Threading-Funktionalität erfordert es, unter Linux gegen die »pthread« -Bibliothek zu linken. Die Kommandozeile

```
g++ -std=c++0x -o dotProductParadotProductPara.cpp -lpthread
```

stellt dies sicher. Die parallele Ausführung der Funktion »getDotProduct()« ist tatsächlich um den Faktor 4 schneller, wie [Abbildung 2](#) zeigt. C++11 hat sichergestellt, dass jeder Thread auf einem anderen Prozessorkern abläuft.

Threads erzeugen

Der Unterschied der beiden Implementierungen reduziert sich auf die Funktion »getDotProduct()«. Der Schlüssel zum Verständnis liegt in [Listing 2](#) in den Funktionsaufrufen dieser Form:

```
auto future= std::async([&]{return std::inner_product(...);})
```

Durch »std::async« wird das Arbeitspaket asynchron in einem eigenen Thread ausgeführt. Dies ist aber nur die halbe Wahrheit: Tatsächlich startet »std::async()« nicht notwendigerweise einen neuen Thread. Die C++11-Implementierung behält es sich vor, »std::async()« wie einen gewöhnlichen Funktionsaufruf zu verwenden, der im aufrufenden Prozess ausgeführt wird. Entscheidungskriterium für C++11, einen neuen Thread zu starten, kann die Anzahl der Prozessorkerne, die Anzahl der bereits aktiven Threads oder die Größe des Arbeitspakets sein: Die intelligente C++11-Implementierung optimiert die Thread-Ausführung für die vorhandene Hardware-Architektur. Der Aufruf »std::async()« gibt mit »auto futureN« eine Referenz auf sich selbst zurück. Über sie lässt sich das Ergebnis der Thread-Ausführung durch »futureN.get()« (Zeile 31) abfragen. Während der ausführende Thread als Promise bezeichnet wird, nennt sich der Thread, der das Ergebnis anfordert, Future. Der Aufruf von »future.get« ist blockierend. Das heißt: Liegt das Ergebnis des Promise noch nicht vor, wartet der Future auf den Rückgabewert. Nach dem Aufruf von »std::async()« in Zeile 12 bleibt nur eine alte Bekannte, die Lambda-Funktion [\[7\]](#), in der Form

```
[&]{return std::inner_product(&v[0],&v[v.size()/4],&w[0],0LL);})
```

übrig. Sie berechnet das Skalarprodukt von »v« und »w«, in diesem Thread für das erste Viertel der beiden Vektoren. Übrigens erfasst die Lambda-Funktion in diesem Beispiel die zwei Vektoren per Referenz »&« , um nicht mehrfach kopieren zu müssen. Das ist allerdings nicht in jedem Fall eine gute Idee. Der [Kasten “Erfassen der Variablen”](#) erläutert Hintergründe und Alternativen.

Erfassen der Variablen

Eine Quelle für subtile Bugs sei gleich im ersten Artikel zum Thema Multithreading in C++11 genannt: Erfasst ein Thread eine Variable per Referenz und nicht per Copy, ist Vorsicht geboten. Zum einen muss der Programmierer gegebenenfalls den Zugriff auf die gemeinsame Variable schützen, zum anderen muss die Variable für die gesamte Lebenszeit der Threads gültig sein. Beide Punkte werden durch die Threads in »getDotProduct()« nicht verletzt. Der Zugriff auf die Elemente der Vektoren ist nur lesend und die Vektoren gelten für die ganze Lebenszeit der Threads. Aufgrund dieser impliziten Gefahren sollte ein Thread in der Regel seine Variablen per Copy erfassen und nur im begründeten Anwendungsfall per Referenz. Eine Ausnahme kann – wie im konkreten Fall »getDotProduct()« – in der Ausführungsgeschwindigkeit des Algorithmus oder darin begründet sein, dass alle Threads die gleiche Variable referenzieren sollen.

Threading vertieft

Zugegeben, dieses Beispiel war einigermaßen konstruiert. Die Lösung beruht nämlich auf dem Wissen des Entwicklers, dass der verwendete PC vier Prozessoren besitzt. Doch C++11

bietet weitere Werkzeuge, um Aufgaben wie die Berechnung des Skalarprodukts auf die jeweils vorliegende PC-Architektur maßzuschneidern. Deren Einsatz erfordert allerdings mehr Wissen über die Multithreading-Fähigkeiten von C++11. Der nächste Artikel befasst sich daher nochmals mit Wissenwertem zu Threads. Dieses Know-how umfasst die Verwaltung der Threads sowie ihre Parametrisierung mit einem Arbeitspaket. Das ist aber erst der Anfang fortgeschrittener Techniken – bleiben Sie dran. (*mhu*)

Infos

1. Herb Sutter, “The Free Lunch is Over”: <http://www.gotw.ca/publications/concurrency-ddj.htm>
2. Herb Sutter, “Welcome to the Jungle”: <http://herbsutter.com/welcome-to-the-jungle/>
3. Skalarprodukt: <http://de.wikipedia.org/wiki/Skalarprodukt>
4. »inner_product()« : http://www.cplusplus.com/reference/std/numeric/inner_product/
5. »0LL« : http://de.wikipedia.org/wiki/Integer_%28Datentyp%29
6. Zeitbibliothek: <http://www.stdthread.co.uk/doc/headers/chrono.html>
7. Rainer Grimm, “Kurz und knackig”: Linux-Magazin 02/12, S. 92
8. Listings zu diesem Artikel: <https://www.linux-magazin.de/static/listings/magazin/2012/04/cpp>

4. Modernes C++ in der Praxis – Folge 4 – Multithreading 2

Von [Rainer Grimm](#)



Eifrig machen sich mehrere Threads zeitgleich an die Arbeit. Doch ohne gegenseitige Abstimmung produzieren sie nur Chaos. Mit Hilfe von Daemon-Threads, Mutexen und leistungsfähigen Locking-Mechanismen sorgt der C++11-Programmierer wieder für Ordnung.

Ein Thread lässt sich in C++11 leicht mit seinem Arbeitspaket ausstatten und starten [1]. Doch der Programmierer muss Sorgfalt walten lassen, damit der Vaterthread mit seinen Kinderthreads koordiniert zusammenarbeitet: Er muss sicherstellen, dass der Vater mindestens so lange lebt wie seine Kinder, und zudem die Zugriffe auf die gemeinsam genutzten Variablen schützen. Dass C++11 dafür das passende Handwerkszeug anbietet, zeigt dieser Artikel. Wie sich in [Listing 1](#) nachvollziehen lässt, geht das Erzeugen und Ausführen

eines Thread in C++11 recht schnell von der Hand. Den Umgang mit den Namensräumen in den Codebeispielen erläutert der Kasten „Explizite »using« -Deklaration“. Die Zeile 16 des Listings erzeugt den Thread »child«. Dessen Aufgabe besteht darin, seinen Namen und seine Identität auf die Standardausgabe »cout« zu schreiben.

Listing 1

Der Vaterthread wartet nicht auf sein Kind

```

01 #include <iostream>
02 #include <thread>
03
04 using std::cout;
05 using std::endl;
06
07 using std::thread;
08 using std::this_thread::get_id;
09
10 int main() {
11
12     cout << endl;
13
14     cout << "father: " << get_id() << endl;
15
16     thread child([]{ cout << "child:  " << get_id() << endl; } );
17
18     cout << endl;
19
20 };

```

Explizite using-Deklaration

Die Codebeispiele der C++11-Serie machen die Funktionen der Namensräume auf eine spezielle Weise bekannt. Sie setzen »using« -Deklarationen ein, etwa [Listing 1](#) für »std« in den Zeilen 4 bis 8. Dadurch wird der Sourcecode deutlich einfacher lesbar. Zudem stellt das explizite Einführen der Funktion »get_id()« durch »using std::this_thread::get_id;« sicher, dass die Zuordnung zwischen der Funktion und dem Namensraum eindeutig ist. Der voll qualifizierte Aufruf von »get_id()« lautet somit »std::this_thread::get_id()«.

Übersetzt und ausgeführt offenbart das Programm aber Probleme ([Abbildung 1](#)): Zum einen schreibt nur der Vater seine Identität auf die Konsole, dem Kind gelingt dies nicht. Zum anderen beendet sich das Programm mit »std::terminate()«. Das liegt daran, dass der Vater nicht wartet, bis der Kinderthread seine Aufgabe erledigt hat. In [Listing 2](#) dagegen stellt der Aufruf von »join()« in Zeile 18 sicher, dass der Vaterthread lange genug lebt. Damit verhält sich das Programm wie erwartet ([Abbildung 2](#)).

Listing 2

Der Vaterthread wartet auf sein Kind

```

01 #include <iostream>
02 #include <thread>
03
04 using std::cout;
05 using std::endl;
06
07 using std::thread;
08 using std::this_thread::get_id;
09

```



```

10 int main(){
11
12     cout << endl;
13
14     cout << "father: " << get_id() << endl;
15
16     thread child([]{ cout << "child:  " << get_id() << endl; } );
17
18     child.join();
19
20     cout << endl;
21
22 };

```

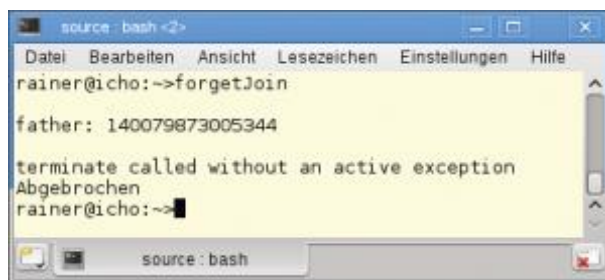


Abbildung 1: Der Kindthread kommt nicht zum Schreiben, bevor der Vaterthread sich beendet.

Bei Anwendungen mit mehr als einem Thread muss sich der Programmierer unbedingt Gedanken über die Lebenszeit des Vaterthread und seiner Kinder machen. Er hat entweder dafür zu sorgen, dass der Vater mindestens so lange lebt wie die Kinder, wofür ein Aufruf der Methode »join()« für jeden Thread sorgt. Oder er macht die Kinder vom Vaterthread unabhängig, was durch Aufrufen von »detach()« gelingt. Die Detach-Methode erzeugt einen so genannten Daemon-Thread, der im Hintergrund läuft und seinen Vater überleben kann [\[2\]](#).

Alle durcheinander

Nach wenigen Anpassungen schreibt das erste Multithreading-Programm definiert auf die Konsole, die hier die von allen Threads gemeinsam genutzte Variable darstellt. Definiert? [Listing 2](#) provoziert im Grunde undefiniertes Verhalten. Doch wie es der Zufall will (oder die Threads es wollen), tritt dieses Verhalten im Beispiel nicht auf. Für Klarheit lässt sich aber sorgen. [Listing 3](#) gibt den Threads deutlich mehr zu tun. Der Boss (Vaterthread) koordiniert seine sechs Arbeiter (Kinderthreads) namens Herb, Andrei, Scott, Bjarne, Andrew und David. Jeder Arbeiter muss drei Arbeitspakete (Zeilen 20 bis 25) ausführen, die er in Form eines

Funktionsobjekts (Zeilen 39 bis 44) erhält. Ein Funktionsobjekt ist ein Objekt, das sich wie eine Funktion aufrufen lässt. Zeile 19 erreicht dies durch das Überladen des Klammeroperators.

Listing 3

Alle Arbeiter rufen durcheinander

```

01 #include <chrono>
02 #include <iostream>
03 #include <thread>
04
05 using std::cout;
06 using std::endl;
07
08 using std::string;
09
10 using std::chrono::milliseconds;
11
12 using std::thread;
13 using std::this_thread::sleep_for;
14
15 class Worker{
16 public:
17     Worker(string n):name(n){};
18
19     void operator() (){
20         for (int i= 1; i <= 3; ++i){
21             // begin work
22             sleep_for(milliseconds(200));
23             // end work
24             cout << name << ": " << "Work " << i << " done !!!" << endl;
25         }
26     }
27
28 private:
29     string name;
30 };
31
32
33 int main(){
34
35     cout << endl;
36
37     cout << "Boss: Let's start working.\n\n";
38
39     thread herb= thread(Worker("Herb"));
40     thread andrei= thread(Worker("  Andrei"));
41     thread scott= thread(Worker("    Scott"));
42     thread bjarne= thread(Worker("      Bjarne"));
43     thread andrew= thread(Worker("        Andrew"));
44     thread david= thread(Worker("          David"));
45
46     herb.join();
47     andrei.join();
48     scott.join();
49     bjarne.join();
50     andrew.join();
51     david.join();
52
53     cout << "\n" << "Boss: Let's go home." << endl;
54

```

```
55     cout << endl;
56
57 }
```

Jedes Arbeitspaket benötigt eine Fünftelsekunde (Zeile 22). Hat ein Arbeiter das soundsovielte Arbeitspaket geschafft, ruft er seinen Namen, damit der Chef im Bilde bleibt. Beispielsweise schreit Herb beim dritten Arbeitspaket: »Herb: Work 3 done !!!«. Der Boss bestimmt in Zeile 37 den Arbeitsbeginn und in Zeile 53, dass das Tagwerk vollbracht ist. [Abbildung 3](#) zeigt, welches Durcheinander beim Ausführen auf dem Terminal landet. Die Arbeiter rufen in ihrem Arbeitseifer die Benachrichtigungen vollkommen wild durcheinander. “Was für ein Chaos! Das muss morgen besser werden”, denkt sich der Boss. Er vereinbart mit den Mitarbeitern für den nächsten Tag, dass jeder seine Benachrichtigung vollständig beenden darf, bevor der nächste an der Reihe ist. [Listing 4](#) bringt ein kleines bisschen mehr Disziplin ins Spiel, der Vorgesetzte ist zu jedem Zeitpunkt im Bilde ([Abbildung 4](#)).

Listing 4

Jeder Arbeiter lässt seine Kollegen ausreden

```
01 #include <chrono>
02 #include <iostream>
03 #include <thread>
04
05 using std::cout;
06 using std::endl;
07
08 using std::string;
09
10 using std::chrono::milliseconds;
11
12 using std::mutex;
13 using std::thread;
14 using std::this_thread::sleep_for;
15
16 mutex coutMutex;
17
18 class Worker{
19 public:
20     Worker(string n):name(n){};
21
22     void operator() (){
23         for (int i= 1; i <= 3; ++i){
24             // begin work
25             sleep_for(milliseconds(200));
26             // end work
27             coutMutex.lock();
28             cout << name << ": " << "Work " << i << " done !!!" << endl;
29             coutMutex.unlock();
30         }
31     }
32
33 private:
34     string name;
35 };
36
37
38 int main(){
39
40     cout << endl;
41
```

```

42  cout << "Boss: Let's start working." << "\n\n";
43
44  thread herb= thread(Worker("Herb"));
45  thread andrei= thread(Worker(" Andrei"));
46  thread scott= thread(Worker("  Scott"));
47  thread bjarne= thread(Worker("   Bjarne"));
48  thread andrew= thread(Worker("    Andrew"));
49  thread david= thread(Worker("     David"));
50
51  herb.join();
52  andrei.join();
53  scott.join();
54  bjarne.join();
55  andrew.join();
56  david.join();
57
58  cout << "\n" << "Boss: Let's go home." << endl;
59
60  cout << endl;
61
62 }

```

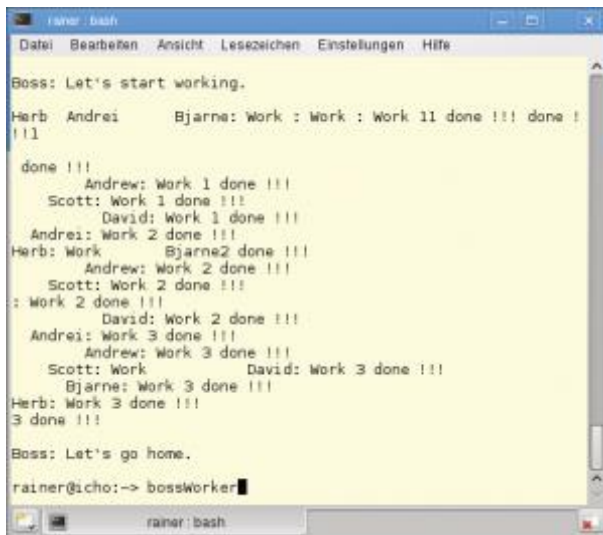


Abbildung 3: Ohne Koordination der Threads rufen die Arbeiter wild durcheinander.

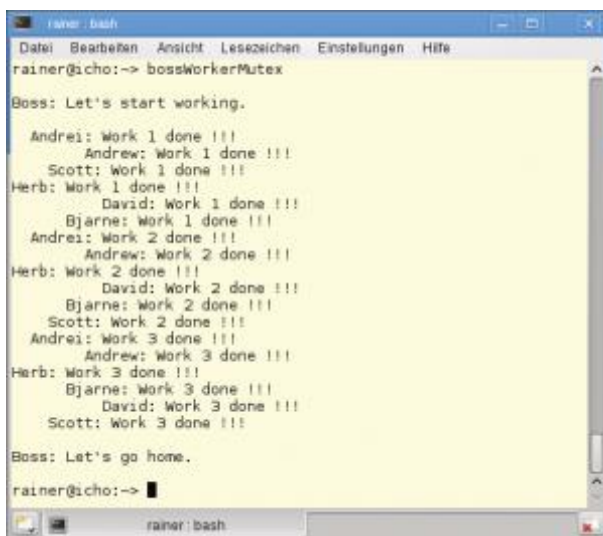


Abbildung 4: Nur ein Arbeiter auf einmal darf rufen, die anderen dürfen ihm währenddessen nicht ins Wort fallen.

Sperren per Mutex

Der Unterschied von [Listing 4](#) zum vorigen steckt in den Zeilen 27 und 29. Der Code verwendet dort in »coutMutex.lock()« einen Mutex, um den Zugriff auf »std::cout« zu sperren, und »coutMutex.unlock()« , um den Mutex wieder freizugeben. Dadurch ist sichergestellt, dass nur ein Thread auf »std::cout« schreiben darf – anders ausgedrückt, dass »std::cout« als gemeinsam genutzte Variable geschützt ist. Erhält ein Thread den Mutex nicht, muss er warten bis dieser frei wird. Genau dies Verhalten findet sich in dem Namen der Datenstruktur wieder: Ein Mutex stellt Mutual Exclusion (gegenseitiges Ausschließen) sicher. Neben diesem einfachen Mutex gibt es in C++11 noch rekursive Mutexe und Mutexe mit Zeitgrenze. Genauer lässt sich das auf der Wikiseite zum C++-Standard [\[3\]](#) nachlesen. Der Boss der Arbeiter ist nun zufrieden, der Autor dieses Artikels aber nicht. Wie sich später noch zeigen wird, sollte der Programmierer Mutexe nicht direkt anwenden, sondern in einem Lock kapseln. Denn was passiert, wenn der Arbeiter »David« beim Ausrufen seiner Meldung – wir wollen es nicht hoffen – stirbt? Alle Arbeiter warten vergebens darauf, dass er seine Nachricht beendet. »David« ist damit – bildlich gesprochen – Besitzer des Lock und gibt es nicht mehr frei. So hindert er alle anderen Arbeiter daran, ihre Nachricht auszurufen und damit den Rest ihrer Arbeit zu erledigen. Die ganze Arbeit kommt zum Erliegen – ein so genanntes Deadlock tritt ein. Ein Deadlock beschreibt eine Situation, in der sich mindestens zwei Prozesse so blockieren, dass keiner mehr einen Fortschritt macht. Die klassische Situation für ein Deadlock besteht darin, dass zwei Threads jeweils dieselben zwei Locks benötigen und diese in verschiedener Reihenfolge anfordern. Grafisch ist das in [Abbildung 5](#) dargestellt: Thread 1 fordert Lock 1, und Thread 2 fordert Lock 2 erfolgreich an und hält es. Damit Thread 1 weiter fortschreiten kann, benötigt er Lock 2, das derzeit aber Thread 2 besitzt. Ähnlich verhält es sich bei Thread 2: Damit er fortfahren kann, benötigt er das bereits vergebene Lock 1. Da kein Thread sein Lock freiwillig abgibt, blockieren sich die beiden Threads gegenseitig.

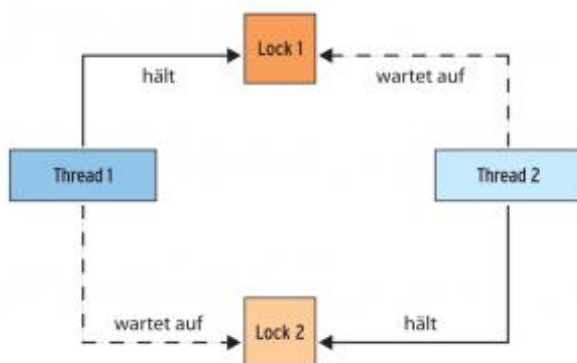


Abbildung 5: Deadlock: Jeder Thread hält ein Lock und wartet darauf, dass der andere seines freigibt.

Deadlock auflösen

C++11 bietet aber nun mit »std::lock« eine Lösung für dieses Problem an, denn mit dessen Hilfe lassen sich mehrere Locks vom Typ »std::unique_lock« atomar locken. Dies bewirkt, dass ein Thread entweder alle oder gar kein Lock erhält. Das ist auch der Grund, warum der Programmierer einen Mutex nicht direkt verwenden, sondern in ein Lock verpacken sollte.

Der leicht modifizierte Worker in [Listing 5](#) zeigt, dass »lock_guard« in Zeile 10 den »coutMutex« kapselt. Dabei bindet dieser seinen Mutex im Konstruktoraufwurf und gibt ihn automatisch im Destruktoraufwurf wieder frei. Dieses bekannte und bewährte C++-Idiom ist als “Resource Acquisition Is Initialization” (RAII, [\[4\]](#)) bekannt. Der Destruktor von »lock_guard« wird genau dann aufgerufen, wenn dieser seinen Gültigkeitsbereich verlässt. Das tritt regulär beim Beenden der For-Schleife oder auch irregulär beim vorzeitigen Ableben des Thread ein.

Listing 5

Sperren durchlock_guard

```
01 class Worker{
02 public:
03     Worker(string n):name(n){};
04
05     void operator() (){
06         for (int i= 1; i <= 3; ++i){
07             // begin work
08             sleep_for(milliseconds(200));
09             // end work
10             lock_guard<mutex>coutGuard(coutMutex);
11             cout << name << ": " << "Work " << i << " done !!!" << endl;
12         }
13
14     }
15 private:
16     string name;
17 };
```

[Listing 6](#) stellt eine Variation des Arbeiters aus [Listing 5](#) dar. Um den Mutex in [Listing 5](#) zu kapseln, erzeugt jeder Aufruf in Zeile 10 ein »lock_guard« -Objekt. Dies ist nötig, da sich »lock_guard« nur durch einen Mutex instanzieren lässt.

Listing 6

Sperren durch unique_lock

```
01 class Worker{
02 public:
03     Worker(string n):name(n){};
04
05     void operator() (){
06         unique_lock<mutex>coutGuard(coutMutex,defer_lock);
07         for (int i= 1; i <= 3; ++i){
08             // begin work
09             sleep_for(milliseconds(200));
10             // end work
11             coutGuard.lock();
12             cout << name << ": " << "Work " << i << " done !!!" << endl;
13             coutGuard.unlock();
14         }
15
16     }
17 private:
18     string name;
19 };
```

Kluge Sperre

Dagegen ist »unique_lock« ([Listing 6](#), Zeile 6) deutlich leistungsfähiger: Es wird nur einmal instanziiert und bindet seinen Mutex. Im Gegensatz zum »lock_guard« sperrt es den Mutex nicht im Konstruktoraufruf. Dafür sorgt »defer_lock« in Zeile 6. Durch den Aufruf »coutGuard.lock()« in Zeile 11 sperrt das »unique_lock« seinen Mutex und gibt ihn durch »lockGuard.unlock()« in Zeile 13 wieder frei. Die Gefahr eines Deadlock besteht hier nicht, da »unique_lock« ein lokales Objekt in der Funktion »operator« in Zeile 5 ist und somit sein Mutex automatisch freigibt, wenn »unique_lock« seinen Gültigkeitsbereich verlässt. Die Klasse »std::unique_lock« kann deutlich mehr als ihr kleiner Bruder »std::lock_guard« . Neben dem expliziten Sperren und Freigeben des Mutex erlaubt es ein »unique_lock« , das Sperren des Mutex an absolute oder relative Zeitangaben zu knüpfen. Auch den Mutex testweise zu sperren unterstützt »unique_lock« . Die Details lassen sich unter [\[5\]](#) nachlesen.

Wie geht's weiter?

Die Abstimmung per Zuruf mag in der menschlichen Arbeitswelt funktionieren, für Threads gibt es aber wesentlich zuverlässigere Mechanismen. Mit Bedingungsvariablen bietet C++11 verlässliche Werkzeuge an, die auf ihren Einsatz warten. Wie man sie verwendet, wird Thema des nächsten Artikels sein. (*mhu*)

Infos

1. Rainer Grimm, "Mehrgleisig unterwegs": Linux-Magazin 04/12, S. 96
2. Thread als Daemon: <http://en.cppreference.com/w/cpp/thread/thread/detach>
3. Mutexe und Locks: <http://en.cppreference.com/w/cpp/thread>
4. Resource Acquisition Is Initialization:
http://de.wikipedia.org/wiki/Ressourcenbelegung_ist_Initialisierung
5. »unique_lock« : http://en.cppreference.com/w/cpp/thread/unique_lock

5. Modernes C++ in der Praxis – Folge 5 – Multithreading 3

Von [Rainer Grimm](#)



Die C++11-Reihe beschäftigt sich weiter mit dem Synchronisieren von Threads. Diesmal setzt der Chef-Thread Bedingungsvariablen ein, um die Tätigkeit seiner Mitarbeiter-Threads zu koordinieren.

Multicore-Rechner und Multithreading gehören zum modernen IT-Alltag. Doch die Arbeit mehrerer Threads will koordiniert sein. Dieser Artikel greift das Programmierbeispiel aus der vorigen Folge wieder auf, in dem ein Chef-Thread sechs Mitarbeiter per Zuruf zu koordinieren versuchte. Zunächst riefen sie alle durcheinander – erst als der Boss die Regel ausgab, dass jeder Arbeiter seinen Kollege zuerst ausreden lassen soll, kehrte Ordnung ein. Dabei repräsentierte je ein Thread einen Arbeiter, und der Ausgabekanal »std::cout« stand für die gemeinsam genutzte Variable, die es zu schützen gilt [\[1\]](#).

Synchronisation per Wahrheitswert

Statt mit Zurufen möchte der Boss seine Mitarbeiter nun mit Hilfe eines Wahrheitswerts synchronisieren. Er hat sich das ganz einfach vorgestellt: Ein Arbeiter beginnt genau dann zu arbeiten, wenn der Wahrheitswert auf »true« gesetzt ist. Da dieser Wert zu Anfang den Wert »false« besitzt, kann der Boss seinen Arbeitern das Startsignal geben. Der erste Prototyp eines kleinen Arbeitsablaufs geht in C++11 schnell von der Hand, wie [Listing 1](#) zeigt. Die Codebeispiele sind aus Platzgründen gekürzt, unter [\[2\]](#) steht der vollständige Quelltext zum Download bereit.

Listing 1

Synchronisation per Wahrheitswert

```
01 [...]
02
03 mutex notifiedMutex;
04 bool notified;
05
06 void worker() {
07
08     cout << "Worker: Waiting for Notification." << endl;
09
10     unique_lock<mutex> lck(notifiedMutex);
11
12     while(!notified) {
13
14         lck.unlock();
15         sleep_for(milliseconds(1000));
16         lck.lock();
17
18     }
19
20     cout << "Worker: Get Notification." << endl;
21
22 }
23
24 void boss() {
25
26     cout << "Boss: Notify Worker." << endl;
27     lock_guard<std::mutex> lck(notifiedMutex);
28     notified=true;
29
30 }
```

```

31
32 int main() {
33
34     cout << endl;
35
36     std::thread workerThread(worker);
37     std::thread bossThread(boss);
38
39     workerThread.join();
40     bossThread.join();
41
42     cout << endl;
43
44 }

```

Das war einfach, denkt der Boss. In Zeile 27 verpackt er den Mutex »notifiedMutex« in einen »lock_guard«. Dies stellt sicher, dass nur ein einziger Thread die kritische Region in Zeile 28 betreten darf.

Der Arbeiter-Thread in den Zeilen 6 bis 22 hat wesentlich mehr zu tun. In Zeile 10 sperrt er den Mutex »notified_mutex« mit einem »unique_lock«. Dies ist notwendig, da ein »unique_lock« mehrmals freigeben wie auch sperren kann. Damit ist der »unique_lock« deutlich mächtiger als der »lock_guard«, den der Chef in Zeile 27 einsetzt, denn dieser bindet den Mutex im Konstruktoraufruf.

Das eigentliche Tagwerk beginnt für den Arbeiter-Thread in der »while« -Schleife (Zeilen 12 bis 18). Solange der Wahrheitswert »notified« nicht »true« ist, führt er immer den gleichen monotonen Job aus: Er gibt den Mutex frei, schläft für 1 Sekunde und bindet den Mutex wieder. Zum einen achtet er darauf, dass selbst das Lesen des Wahrheitswerts »notified« durch den Mutex »notifiedMutex« geschützt ist, zum anderen darauf, dass er den Mutex in Zeile 15 für 1 Sekunde freigibt, damit der Boss den Wert auf »true« setzen kann.

Obwohl der Boss alles richtig gemacht hat ([Abbildung 1](#)), ist er mit seinem Prototyp nicht zufrieden. Ihn stört, dass bis zu einer ganzen Sekunde vergeht, bis der Arbeiter auf eine Benachrichtigung reagiert. Der Arbeiter fühlt sich hingegen für seinen Job überqualifiziert, da er seine ganze Arbeitsenergie auf das immerwährende Kontrollieren des Wahrheitswerts verschwenden muss.

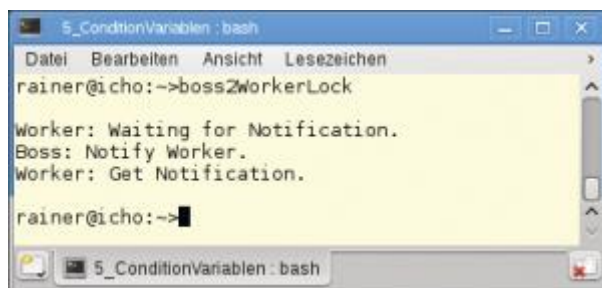


Abbildung 1: Die erste Implementierung: Die Synchronisation über einen gemeinsamen Wahrheitswert funktioniert.

Hinsichtlich der Threads hat der Vorgesetzte bei seiner Implementierung folgende zwei Anforderungen an einen Benachrichtigungsmechanismus nicht gut bedient:

- Die Zeit zwischen dem Senden und dem Empfangen der Benachrichtigung soll möglichst kurz sein.
- Der wartende Thread soll möglichst wenig CPU-Zeit beanspruchen.

Der Boss ist wieder in der Pflicht und sieht sich die Bedingungsvariablen in C++11 genauer an, denn diese setzen die beiden Anforderungen deutlich besser um. Einerseits wacht der Empfänger bei Benachrichtigung sofort auf, andererseits verwendet er wenig CPU-Zeit auf das Warten.

Bedingungsvariablen

Bedingungsvariablen bieten zwei Typen von Operationen: Sie erlauben es, wahlweise ein Signal zu senden oder auf eins zu warten. Dabei kann der Adressat einer Benachrichtigung ein einzelner (»notify_one()«) oder mehrere Threads (»notify_all()«) sein, die sich im wartenden Zustand (»wait()«) befinden. Sendet ein Thread eine Benachrichtigung an einen wartenden Thread, so wacht der Empfänger-Thread auf und fährt mit seiner Arbeit fort. Der Empfänger muss dabei die gleiche Bedingungsvariable benutzen wie der Sender-Thread.

Mit dem Arbeitsablauf in [Listing 2](#) sind der Arbeiter und der Boss deutlich zufriedener. Zuerst zum Boss: Durch den Aufruf »condVar.notify_one()« in Zeile 21 schickt er seine Benachrichtigung an einen Arbeiter. Dieser wartet in Zeile 11 darauf und fährt genau dann mit seiner Arbeit fort, wenn er die Nachricht erhält. Dabei verwenden beide den gleichen Mutex »notifiedMutex«.

Listing 2

Synchronisation per Bedingungsvariable

```

01 [...]
02
03 mutex notifiedMutex;
04 condition_variable condVar;
05 bool notified;
06
07 void worker(){
08
09 cout << "Worker: Waiting for Notification." << endl;
10 unique_lock<mutex> lck(notifiedMutex);
11 condVar.wait(lck, []{return notified;});
12 cout << "Worker: Get Notification." << endl;
13
14 }
15
16 void boss(){
17
18 cout << "Boss: Notify Worker." << endl;
19 lock_guard<std::mutex> lck(notifiedMutex);
20 notified=true;
21 condVar.notify_one();
22
23 }
24
25 int main(){
26
27 cout << endl;
28

```

```

29 std::thread workerThread(worker);
30 std::thread bossThread(boss);
31
32 workerThread.join();
33 bossThread.join();
34
35 cout << endl;
36
37 }

```

Etwas befremdlich wirkt, dass der Boss und der Arbeiter weiter den Wahrheitswert »notified« verwenden. Tatsächlich erhält der Arbeiter nicht nur die Benachrichtigung, sondern versichert sich zusätzlich im »wait()« -Aufruf (Zeile 11) durch die Lambda-Funktion »[] {return notified;}« [\[3\]](#), dass der Wert auf »true« gesetzt ist. Das ist notwendig, weil es vorkommt, dass der Arbeiter irrtümlich eine Benachrichtigung erhält. Dieses Phänomen ist unter dem Namen Spurious Wakeups [\[4\]](#) bekannt. Der Wahrheitswert »notified« verhindert, dass der Thread wegen eines solchen Fehlalarms seine Arbeit aufnimmt.

Synchronisation der Arbeiterschaft

Die weiteren Details rund um Bedingungsvariablen hat der Boss unter [\[5\]](#) nachgelesen. Nun will er sie im großen Stil für die Koordination seiner Arbeiter einsetzen. In [Abbildung 2](#) ist sein Plan dargestellt. Jeder Arbeiter bereitet zuerst seine Arbeit vor und gibt dem Boss Bescheid, sobald er fertig ist. Sind alle Arbeiter mit der Vorbereitung fertig, signalisiert der Boss allen gleichzeitig, dass sie mit ihrem Tagwerk beginnen können. Nun gilt es für den Boss, zu warten, bis jeder Arbeiter ihm zurückgemeldet hat, dass er sein Arbeitspaket erledigt hat. Zum Schluss schickt er Herb, Scott, Bjarne, Andrei, Andrew und David mit »GO HOME« in den Feierabend. Der Boss hat es geschafft. Der komplexe Arbeitsablauf seiner Arbeiter läuft geordnet ab ([Abbildung 3](#)).

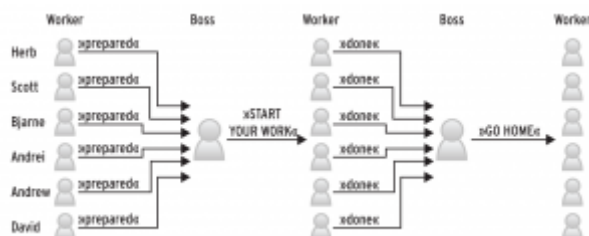


Abbildung 2: Mit Hilfe von Benachrichtigungen lässt sich der Arbeitsablauf effizient gestalten: Die Arbeiter geben Bescheid, wenn sie bereit oder fertig sind, der Chef gibt das Kommando für Arbeit und Feierabend.

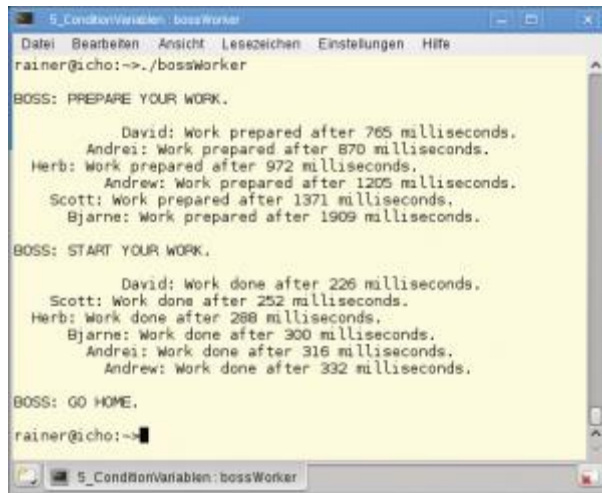


Abbildung 3: Dank Bedingungsvariablen klappt der Arbeitsablauf wie am Schnürchen. Die Arbeiter warten auf den Boss und dieser auf die Arbeiter.

Ein Blick auf den Sourcecode in Listing 3 zeigt: Um die sechs Arbeiter zu koordinieren, muss der Boss einiges an Buchhaltung erledigen. Neben Bedingungsvariablen kommen Locks, Mutexe und auch atomare Variablen zum Einsatz. Atomare Variablen bieten atomare Lese- und Schreib-Operationen an, sodass sie von mehreren Threads gleichzeitig ohne weiteren Schutzmechanismus modifiziert werden dürfen. Sie sind Bestandteil des neuen C++11-Standards [6]. Dabei stehen nicht nur Wahrheitswerte (`std::atomic_bool`), sondern auch Zeichen (`std::atomic_char`) und natürliche Zahlen (`std::atomic_int`) in verschiedenen Variationen als atomare Datentypen zu Verfügung.

Listing 3

Synchronisieren mit Bedingungsvariablen

```

01 [...]
02
03 int getRandomTime(int start, int end){
04
05     random_device seed;
06     mt19937 engine(seed());
07     uniform_int_distribution<int> dist(start,end);
08
09     return dist(engine);
10 };
11
12 class Worker{
13 public:
14     Worker(string n):name(n){};
15
16     void operator() (){
17
18         // prepare the work and notify the boss
19         int prepareTime= getRandomTime(500,2000);
20         sleep_for(milliseconds(prepareTime));
21         preparedCount++;
22         cout << name << ": " << "Work prepared after " << prepareTime << "
23         milliseconds." << endl;
24         worker2BossCondVariable.notify_one();
25     }
26 };

```

```

25 { // in order to release the lock startWorkMutex
26
27 // wait for the start notification of the boss
28 unique_lock<mutex> startWorkLock( startWorkMutex );
29 boss2WorkerCondVariable.wait( startWorkLock, []{ return startWork; });
30
31 }
32
33 // do the work and notify the boss
34 int workTime= getRandomTime(200,400);
35 sleep_for(milliseconds(workTime));
36 doneCount++;
37 cout << name << ": " << "Work done after " << workTime << "
milliseconds." << endl;
38 worker2BossCondVariable.notify_one();
39
40 { // in order to release the lock goHomeMutex
41
42 // wait for the go home notification of the boss
43 unique_lock<mutex> goHomeLock( goHomeMutex );
44 boss2WorkerCondVariable.wait( goHomeLock, []{ return goHome;});
45
46 }
47
48 }
49 private:
50 string name;
51 };
52
53 int main(){
54
55 cout << endl;
56
57 Worker herb(" Herb");
58 thread herbWork(herb);
59
60 // start thread scott, bjarne, andrei, andrew and david
61 [...]
62
63 cout << "BOSS: PREPARE YOUR WORK.\n " << endl;
64
65 // waiting for the worker
66 preparedCount.store(0);
67 unique_lock<mutex> preparedUniqueLock( preparedMutex );
68 worker2BossCondVariable.wait(preparedUniqueLock, []{ return preparedCount
== 6; });
69
70 // notify the worker about the begin of the work
71 startWork= true;
72 cout << "\nBOSS: START YOUR WORK. \n" << endl;
73 boss2WorkerCondVariable.notify_all();
74
75 // waiting for the worker
76 doneCount.store(0);
77 unique_lock<mutex> doneUniqueLock( doneMutex );
78 worker2BossCondVariable.wait(doneUniqueLock, []{ return doneCount == 6;
});
79
80 // notify the worker about the end of the work
81 goHome= true;
82 cout << "\nBOSS: GO HOME. \n" << endl;
83 boss2WorkerCondVariable.notify_all();
84

```

```

85 herbWork.join();
86 scottWork.join();
87 bjarneWork.join();
88 andreiWork.join();
89 andrewWork.join();
90 davidWork.join();
91
92 }

```

Mit Hilfe des neuen Klassentemplates »std::atomic<T>« kann der erfahrene C++-Entwickler zudem einen eigenen Datentyp definieren, der den Datentyp »T« um atomare Eigenschaften erweitert. Datenstrukturen, die durch den Einsatz von atomaren Datentypen gänzlich ohne Locks und Mutexe auskommen, werden als Lock-free bezeichnet. Tiefere Einsichten vermittelt der Artikel [\[7\]](#) von Anthony Williams.

In Listing 3 initiiert der Boss den Arbeitsablauf, indem er seine Arbeiter ab Zeile 60 startet. Anschließend wartet er in Zeile 68, bis alle Arbeiter melden, dass sie ihre Arbeit vorbereitet haben. Er verwendet die atomare Variable »preparedCount«, um die Anzahl der Nachrichten zu zählen.

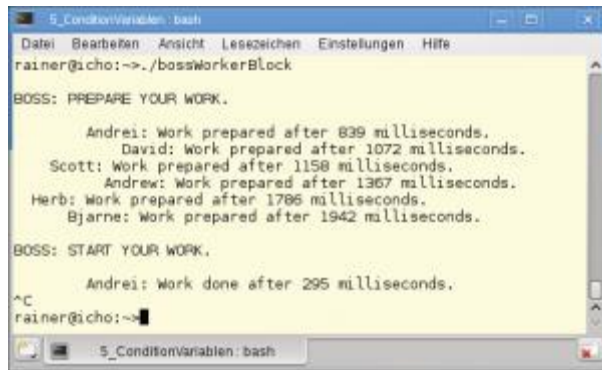
Sind alle Arbeiter mit ihrer Vorbereitung fertig, gibt er mit »boss2WorkerCondVariable.notify_all()« (Zeile 73) allen die Benachrichtigung, dass sie mit ihrer Arbeit beginnen können. Nun gilt es wieder für den Boss, in Zeile 78 zu warten, bis alle ihr Tagwerk vollendet haben. Ist dies geschehen, schickt er durch seine Benachrichtigung Herb, Scott, Bjarne, Andrei, Andrew und David nach Hause.

Arbeitsablauf

Der Gang der Ereignisse ist bei den Arbeitern genau spiegelbildlich zu denen beim Boss. Als Beispiel soll Bjarnes Arbeitstag dienen. Hat Bjarne die Vorbereitung seiner Arbeit abgeschlossen – ein Zufallswert bestimmt in Zeile 19, wie lange sie dauert –, inkrementiert er die atomare Variable »preparedCount« in Zeile 21 und gibt anschließend dem Boss durch »worker2BossCondVariable.notify_one()« Bescheid, dass er fertig ist. Nun wartet er in Zeile 29 auf die Nachricht vom Boss, dass er mit seiner Arbeit beginnen kann. Erhält er diesen Bescheid, beginnt für ihn das Tagwerk und die Ereignisse wiederholen sich. In Zeile 38 teilt er dem Boss mit, dass er mit seiner Arbeit fertig ist. In Zeile 44 wartet er schließlich darauf, dass er endlich Feierabend hat.

Wenn es klemmt

Zu Recht ist der Boss stolz auf seinen ausgefeilten Arbeitsplan. Dabei sei nicht verschwiegen, dass er bei seinem ersten Lösungsansatz den künstlichen Bereich um den Lock »startWorkLock« von Zeile 25 bis 31 und genauso um den Lock »goHomeLock« von Zeile 40 bis 46 vergessen hatte. Das Ergebnis war ein Deadlock ([Abbildung 4](#)).



```

5_ConditionVariablen: bash
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~$ ./bossWorkerBlock

BOSS: PREPARE YOUR WORK.

Andrei: Work prepared after 839 milliseconds.
David: Work prepared after 1072 milliseconds.
Scott: Work prepared after 1158 milliseconds.
Andrew: Work prepared after 1367 milliseconds.
Herb: Work prepared after 1786 milliseconds.
Bjarne: Work prepared after 1942 milliseconds.

BOSS: START YOUR WORK.

Andrei: Work done after 295 milliseconds.
^C
rainer@icho:~$
  
```

Abbildung 4: Die Arbeiter blockieren sich gegenseitig, weil der erste noch den steuernden Mutex hält.

Die Ursache: Der erste Arbeiter, der den Mutex »startWorkMutex« in Zeile 28 verwendet, bindet diesen und wartet in Zeile 44 darauf, dass er in den Feierabend gehen darf. Er wartet, hält aber dabei noch den Mutex. Damit harren die anderen Arbeiter vergeblich auf den Mutex und der ganze Arbeitsablauf klemmt. Die zunächst vergessenen künstlichen Bereiche in den Zeilen 25 bis 31 und 40 bis 46 lösen das Problem: »unique_lock« verliert seine Gültigkeit und gibt automatisch seinen Mutex frei.

Wie geht's weiter?

Die anfängliche Euphorie des Chefs ist allmählich der Skepsis gewichen. Zwar hat sich sein Programm zur Koordination der Arbeiter in den letzten Wochen im Großen und Ganzen bewährt – sieht man von dem Vorfall ab, als die Arbeiter ihre »prepared« -Nachricht schon verschickten, bevor sich der Boss in den Wartezustand begeben hatte. Das Ergebnis war, dass die Arbeiter vergeblich auf ihre Benachrichtigung zum Arbeitsbeginn warteten.

Den Boss stört aber vor allem, dass die ganze Koordination der Arbeiter viel zu komplex ist. Sie erfordert Synchronisationsmittel wie etwa Bedingungsvariablen, Locks, Mutexe, ja sogar atomare Variablen. Kein Wunder, dass sich Fehler einschleichen. Da entdeckt er beim Schmökern im neuen C++11-Standard die Futures. Sie erlauben es, in Programmen auf die Vollendung einer Aufgabe zu warten. Die nächste Folge dieser Artikelserie wird zeigen, ob es dem Boss mit Futures tatsächlich leichter von der Hand geht, seine sechs Mitarbeiter zu koordinieren.

Infos

1. Rainer Grimm, "Gemeinsam ins Ziel": Linux-Magazin 06/12, S. 90
2. Listings zum Artikel: <https://www.linux-magazin.de/static/listings/magazin/2012/08/cpp/>
3. Rainer Grimm, "Kurz und knackig": Linux-Magazin 02/12, S. 92
4. Spurious Wakeups: http://en.wikipedia.org/wiki/Spurious_wakeup
5. Condition Variable: http://en.cppreference.com/w/cpp/thread/condition_variable
6. Header »<atomic>« : <http://www.stdthread.co.uk/doc/headers/atomic.html>
7. Lock-free: **Fehler! Linkreferenz ungültig.**

6. Modernes C++ in der Praxis – Folge 6 – Multithreading 4

Von [Rainer Grimm](#)



Promise und Future erweisen sich als nützliche Neuerungen im C++11-Standard. Das Gespann macht die bisher aufwändige Synchronisation mehrerer Threads einfach und übersichtlich.

Dieser Beitrag der C++11-Reihe knüpft an den im Linux-Magazin 08/12 an [\[1\]](#). Er vollendet das Programmierbeispiel vom Boss und seinen sechs Mitarbeitern. Um die Arbeitskraft seiner Arbeiter in geregelte Bahnen zu lenken, setzt der Chef diesmal Promise (Versprechen) und Future (Zukunft) ein. Wie diese neuen Features funktionieren und warum sie die mühselige Synchronisation der Arbeiter-Threads durch atomare Variablen, Mutexe, Locks und Bedingungsvariablen überflüssig machen, zeigt dieser Artikel.

Compilerversionen

Nur der aktuelle GCC 4.7 kann die Codebeispiele aus diesem Artikel übersetzen. Denn die älteren GCC-Implementierungen setzen voraus, dass die Argumente des Thread kopiert werden. Weder Promise noch Future sind jedoch kopierbar.

Arbeiten auf Zuruf

Die Aufgabe, vor der der Boss steht, ist schnell skizziert ([Abbildung 1](#)) – Leser der vorigen Folgen kennen sie bereits: Die Arbeiter Herb, Scott, Bjarne, Andrei, Andrew und David benachrichtigen den Boss, sobald sie die Vorbereitungen für ihre Arbeit abgeschlossen haben. Sobald der Boss alle Benachrichtigungen erhalten hat, gibt er ihnen das Kommando, das Tagwerk zu beginnen. Jeder der sechs Arbeiter teilt später dem Boss mit, wenn er mit seiner Arbeit fertig ist.

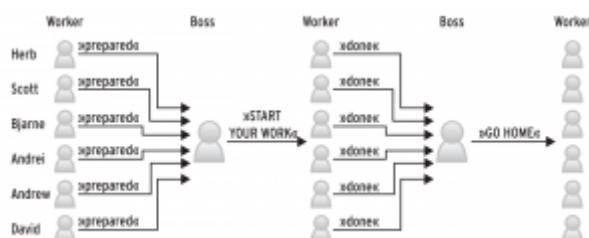


Abbildung 1: Der Arbeitsablauf für die Arbeiter Herb, Scott, Bjarne, Andrei, Andrew und David.

Hat der Boss diese Nachricht von allen erhalten, ist das Arbeitspensum erledigt und er schickt alle in den Feierabend. Damit gibt es vier Synchronisationspunkte für ein Programm, das diesen Arbeitsablauf mittels mehrerer Threads abbildet: Zwei muss der Boss einhalten, die anderen beiden die Arbeiter.

Ein erster Versuch

Dieser Arbeitsablauf ist nicht so trivial, wie er klingt. Verständlich, dass der Boss zuerst einen Prototyp entwirft und diesen mit seinem Vorarbeiter Bjarne testet, zumal er mit Future und Promise vollkommenes Neuland betritt ([Listing 1](#)). Auf der informativen Wikiseite C++ Reference [\[2\]](#) lässt sich das ganze Leistungsspektrum des Paares Promise [\[3\]](#) und Future [\[4\]](#) nachlesen. Das Ergebnis des Prototyps sieht vielversprechend aus. Das Paar »promise« in Zeile 36 und »future« in Zeile 39 baut einen Datenkanal auf, in dem Promise als Datensender und Future als Datenempfänger dient.

Listing 1

Prototyp

```
01 #include <future>
02 #include <iostream>
03 #include <thread>
04 #include <utility>
05
06 using std::cout;
07 using std::endl;
08
09 using std::string;
10 using std::move;
11
12 using std::promise;
13 using std::future;
14
15 using std::thread;
16
17 class Worker{
18 public:
19     Worker(string n):name(n){};
20
21     void operator() (future<string>&& boss2Worker){
22
23         // still waiting for the permission to start working
24         string message= boss2Worker.get();
25         cout << "Worker from Boss: " << message << endl;
26     }
27 private:
28     string name;
29 };
30
31 int main(){
32
33     cout << endl;
34
35     // define the promise => Instruction from the boss
```



```

36  promise<string> startWorkPromise;
37
38  // get the futures from the promise
39  future<string> startWorkFuture= startWorkPromise.get_future();
40
41  Worker bjarne("Bjarne");
42  thread bjarneWork(bjarne,move(startWorkFuture));
43
44  // notify the worker about the begin of the work
45  cout << "Boss: Notifying the worker. \n" << endl;
46  startWorkPromise.set_value("START YOUR WORK!");
47
48  bjarneWork.join();
49
50  cout << endl;
51
52 }

```

Dabei ist es weder notwendig, dass die beiden Kommunikationsendpunkte sich in verschiedenen Threads befinden, noch müssen tatsächlich Daten geschickt werden. Beispielsweise kann der Promise auch nur eine Benachrichtigung oder eine Ausnahme an den Future schicken. Exemplarisch sind Datensender und -empfänger in [Abbildung 2](#) dargestellt.

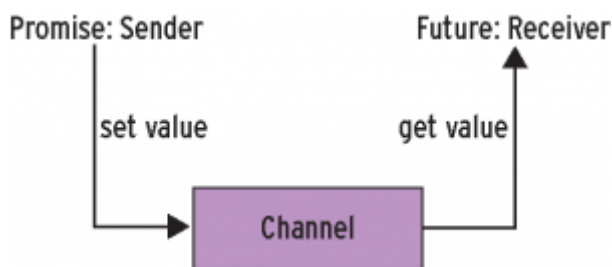


Abbildung 2: Promise und Future als Sender und Empfänger von Nachrichten.

Kommunikationskanal

Zeile 36 von [Listing 1](#) erzeugt den Promise, der einen »string« als Rückgabewert besitzt. Durch die Methode »get_future()« in Zeile 39 gibt dieser einen Future zurück, der durch einen Kanal mit ihm verbunden ist. Dieser Future erwartet einen »string« vom Promise. Der Arbeiter-Thread Bjarne erhält den Future als Parameter. Dabei verschiebt Zeile 42 mit »move(startWorkFuture)« den Future in einen neuen Thread.

Den Future nimmt der Thread in Zeile 21 mit »future<string>&&« an, einer so genannten Rvalue-Referenz. Dies ist notwendig, da ein Future sich nicht kopieren lässt. Schreibt der Boss in Zeile 46 durch den Aufruf »startWorkPromise.set_value(“START YOUR WORK!”)« den Wert in den Kanal, steht dieser bereit, um vom Arbeiter abgeholt zu werden. Da der Aufruf von »boss2Worker.get()« in Zeile 24 blockiert, muss der Arbeiter darauf warten, dass der Boss den Wert explizit setzt. Dieses Warten auf den Wert des Promise kann ein Future durch eine absolute oder relative Zeitangabe einschränken.

Das Meisterwerk

Stolz präsentiert der Boss seinen neu entworfenen Arbeitsablauf mit [Listing 2](#) und führt ihn gleich aus ([Abbildung 3](#)). Der wesentliche Unterschied zu früheren Versionen ist, dass er mit

seinen Arbeitern vier Datenkanäle verwendet: zwei, um Nachrichten von den Arbeitern zu empfangen, und zwei, um ihnen seine Anweisungen zu erteilen. So erhält der Arbeiter-Thread »herbWork()« in Zeile 59 die zwei Promises »herbPrepared« und »herbDone«, mit denen Herb signalisiert, dass er die Arbeit vorbereitet beziehungsweise vollendet hat. Da Promises nicht kopierbar sind, müssen sie wie im Prototyp verschoben werden.

Listing 2

Promise und Future (gekürzt)

```

01 [...]
02
03 int getRandomTime(int start, int end){
04
05     random_device seed;
06     mt19937 engine(seed());
07     uniform_int_distribution<int> dist(start,end);
08
09     return dist(engine);
10 };
11
12 class Worker{
13 public:
14     Worker(string n):name(n){};
15
16     void operator() (promise<void>&& preparedWork, shared_future<void>
boss2WorkerStartWork,
17     promise<void>&& doneWork, shared_future<void>boss2WorkerGoHome ){
18
19         // prepare the work and notfiy the boss
20         int prepareTime= getRandomTime(500,2000);
21         sleep_for(milliseconds(prepareTime));
22         preparedWork.set_value();
23         cout << name << ": " << "Work prepared after " << prepareTime << "
milliseconds." << endl;
24
25         // still waiting for the permission to start working
26         boss2WorkerStartWork.wait();
27
28         // do the work and notify the boss
29         int workTime= getRandomTime(200,400);
30         sleep_for(milliseconds(workTime));
31         doneWork.set_value();
32         cout << name << ": " << "Work done after " << workTime << "
milliseconds." << endl;
33
34         // still waiting for the permission to go home
35         boss2WorkerGoHome.wait();
36
37     }
38 private:
39     string name;
40 };
41
42 int main(){
43
44     cout << endl;
45
46     // define the promise => Instruction from the boss
47     promise<void> startWorkPromise;
```

```

48 promise<void> goHomePromise;
49
50 // get the shared futures from the promise
51 shared_future<void> startWorkFuture= startWorkPromise.get_future();
52 shared_future<void> goHomeFuture= goHomePromise.get_future();
53
54 promise<void> herbPrepared;
55 future<void> waitForHerbPrepared= herbPrepared.get_future();
56 promise<void> herbDone;
57 future<void> waitForHerbDone= herbDone.get_future();
58 Worker herb(" Herb");
59 thread
herbWork(herb,move(herbPrepared),startWorkFuture,move(herbDone),goHomeFuture);
60
61 // start thread scott, bjarne, andrei, andrew and david
62 [...]
63
64 cout << "BOSS: PREPARE YOUR WORK.\n " << endl;
65
66 // waiting for the worker
67 waitForHerbPrepared.wait(), waitForScottPrepared.wait(),
waitForBjarnePrepared.wait(), waitForAndreiPrepared.wait(),
waitForAndrewPrepared.wait(), waitForDavidPrepared.wait();
68
69 // notify the worker about the begin of the work
70 cout << "\nBOSS: START YOUR WORK. \n" << endl;
71 startWorkPromise.set_value();
72
73 // waiting for the worker
74 waitForHerbDone.wait(), waitForScottDone.wait(),
waitForBjarneDone.wait(), waitForAndreiDone.wait(),
waitForAndrewDone.wait(), waitForDavidDone.wait();
75
76 // notify the worker about the end of the work
77 cout << "\nBOSS: GO HOME. \n" << endl;
78 goHomePromise.set_value();
79
80 herbWork.join();
81 scottWork.join();
82 bjarneWork.join();
83 andreiWork.join();
84 andrewWork.join();
85 davidWork.join();
86
87 }

```

```

C++11Serie/b: bash
rainer@icho:~$ ./bossWorkerFutures
BOSS: PREPARE YOUR WORK.
    Andrew: Work prepared after 831 milliseconds.
    Bjarne: Work prepared after 900 milliseconds.
    David: Work prepared after 1328 milliseconds.
    Herb: Work prepared after 1732 milliseconds.
    Andrei: Work prepared after 1787 milliseconds.
    Scott: Work prepared after 1952 milliseconds.
BOSS: START YOUR WORK.
    Bjarne: Work done after 210 milliseconds.
    David: Work done after 269 milliseconds.
    Herb: Work done after 296 milliseconds.
    Andrei: Work done after 333 milliseconds.
    Andrew: Work done after 369 milliseconds.
    Scott: Work done after 390 milliseconds.
BOSS: GO HOME.
rainer@icho:~$

```

Abbildung 3: Der fertige Arbeitsablauf mit den Nachrichten von Boss und sechs Arbeitern.

Das Besondere an Herbs Benachrichtigung ist dieses Mal, dass er in den Zeilen 22 und 31 keinen Wert mitschickt, sondern nur den Boss benachrichtigt. Dies ist auch der Grund dafür, dass der Boss in Zeile 67 lediglich wartet, bis er Nachrichten von Herb und den restlichen Arbeitern erhält.

Um die Arbeiter zu informieren, dass sie mit der Arbeit beginnen beziehungsweise nach Hause gehen können, verwendet der Boss zwei »shared_future«, nämlich »startWorkFuture« und »goHomeFuture«. Während der Future »future« im Prototyp genau einen Sender mit einem Empfänger verbindet, erlaubt es der »shared_future« dem Boss, eine Benachrichtigung an alle Arbeiter zu verschicken (Zeilen 71 und 78). Im Gegensatz zum »future« ist der »shared_future« kopierbar, sodass man ihn nicht in den Arbeiter-Thread verschieben muss (Zeile 59). [Listing 2](#) ist gekürzt wiedergegeben, die vollständige Version gibt es unter [\[5\]](#).

Wie geht es weiter?

Dieser Artikel über Promise und Future schließt die Miniserie über Multithreading in C++11 vorerst ab. Wer noch den einen oder anderen Artikel zum Thema wünscht, schreibt dem Autor unter <mailto:cpp@linux-magazin.de>.

Die Multithreading-Fähigkeit von C++11 hat nämlich noch ein paar interessante Features zu bieten. Dazu gehören das sichere Initialisieren von Daten in Threads, Thread-lokaler Speicher und insbesondere atomare Datentypen sowie das sehr anspruchsvolle C++11-Speichermodell. Dieses Modell definiert, in welcher Reihenfolge atomare Datenoperationen in einem Thread ausgeführt werden und wann diese Operationen zwischen den Threads sichtbar sind.

In den Erläuterungen zu [Listing 1](#) hieß es relativ oberflächlich, dass sich weder Future noch Promise kopieren lassen, sie müssten verschoben werden. Diese einstweilige Oberflächlichkeit hat einen Grund: Der nächste Artikel dieser Reihe wird sich genau diesem Thema widmen. Er erläutert die erwähnten Rvalue-Referenzen und deren Anwendung in der Move-Semantik sowie beim Perfect Forwarding.

Move-Semantik erlaubt es, an der Performance-Schraube eines Programms zu drehen, da sie das aufwändige Kopieren eines Objekts durch das wirtschaftliche Verschieben ersetzt. Perfect

Forwarding hingegen löst ein recht einfaches, aber in C++ bisher ungelöstes Problem: Funktionsargumente generisch annehmen und unverändert weitergeben. (*mhu*)

Infos

1. Rainer Grimm, “Im Gleichtakt”: Linux-Magazin 08/12, S. 88
2. C++ Reference: <http://en.cppreference.com/w/cpp>
3. Promise: <http://en.cppreference.com/w/cpp/thread/promise>
4. Future: <http://en.cppreference.com/w/cpp/thread/future>
5. Vollständige Quelltext-Listings zu diesem Artikel: <https://www.linux-magazin.de/static/listings/magazin/2012/10/cpp>