Rainer Grimm

C++11

Der Leitfaden für Programmierer zum neuen Standard

ADDISON-WESLEY

ALWAYS LEARNING

PEARSON

C++11

Hier eine Auswahl:

Clean Coder
Verhaltensregeln für professionelle Programmierer

Robert C. Martin
240 Seiten
€ 34,80 [D] € 35,80 [A]
ISBN 978-3-8273-3104-5

Programmiersprachen und Entwicklungsplattformen kommen und gehen. Eine Zeitlang wird der Markt von einer bestimmten Software dominiert bis zur Veröffentlichung eines neuen Produkts. Methoden werden ausführlich diskutiert, bis eine Einigung erzielt wird - die letzten Endes doch bald wieder verworfen wird. Wen wundert es da, dass die Entwicklung von Software einer hohen Fluktuation unterliegt. Diejenigen Programmierer, die in diesem von stetigem Wandel geprägten Berufsfeld beständigen Erfolg vorweisen, haben alle eine Sache gemeinsam: Sie schaffen ihre Software mit größter Sorgfalt und sehen ihre Tätigkeit als Kunsthandwerk.
In diesem Buch erklärt Software-Legende Robert C. Martin, weshalb sich Programmierer bei ihrem Job viel Mühe geben sollten, wie Firmen ein Umfeld pflegen können, welches man zum erfolgreichen Programmieren braucht und was für den einzelnen Software-Entwickler bedeutet, wirklich wie ein Kunsthandwerker zu arbeiten. Das Buch zeichnet ein komplettes Bild vom Berufsfeld des Programmierers, indem neben einem gewissen Berufsethos auch verschiedene Fachrichtungen, Techniken, Tools und Anwendungen beschrieben werden, die man als erfolgreicher Software-Entwickler braucht.

Effektiv C++ programmieren

Scott Meyers
336 Seiten
€ 34,80 [D] € 35,80 [A]
ISBN 978-3-8273-3078-9

Dieses Buch ist in 55 Themen gegliedert, die jeweils eine Maßnahme beschreiben, um besseren C++-Code zu schreiben. Jedes dieser Themen wird durch Beispiele illustriert. Mehr als die Hälfte des Inhalts dieser dritten Ausgabe ist neu, unter anderem die Kapitel über die Verwaltung von Ressourcen und die Verwendung von Templates. Die Themen aus der zweiten Ausgabe wurden sorgfältig überarbeitet, um die Anforderungen modernen Softwaredesigns widerzuspiegeln - darunter Ausnahmen, Entwurfsmuster und Multithreading.

Rainer Grimm

C++11

Der Leitfaden für Programmierer zum neuen Standard

ADDISON-WESLEY

An imprint of Pearson

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

10 9 8 7 6 5 4 3 2 1

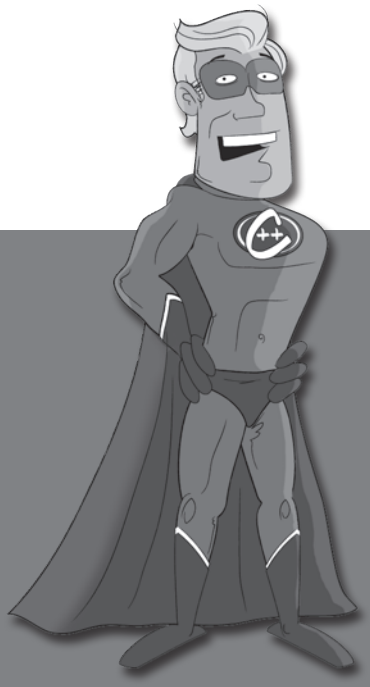
14 13 12

ISBN 978-3-8273-3088-8

ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Brigitte Bauer-Schiewek, bbauer@pearson.de
Fachlektorat: Dirk Frischalowski
Korrektorat: Petra Kienle
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Coverkonzeption und -gestaltung: Marco Lindenbeck, webwo GmbH, mlindenbeck@webwo.de
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

3 Kernsprache



Vergleichen wir C++ mit einer modernen Interpretersprache wie Python oder einer Compiler-Sprache wie Java, dann sind die Hürden, um die Sprache zu meistern, die es in C++ zu überwinden gibt, viel höher. Klar, der Vergleich ist ungerecht, muss sich doch Python nicht mit statischer Typisierung auseinandersetzen und verfolgt Java relativ streng die objektorientierte Programmiertechnik. Doch die Hürde wird mit C++11 deutlich niedriger als mit C++. Die Usability steht bei C++11 im Fokus.

3.1 Usability

DEFINITION

Usability

Usability wird auf Wikipedia als die einfache Handhabung und Erlernbarkeit eines von Menschen geschaffenen Objekts bezeichnet. (»*Usability is the ease of use and learnability of a human-made object.*«) (Usability)

3.1.1 Die Range-basierte For-Schleife

Ein kleines, aber feines Feature ist die Range-basierte For-Schleife, die das Iterieren über Container deutlich einfacher von der Hand gehen lässt. Sie ist dem einen oder anderen sicher aus Python oder Java schon bekannt.

Wird diese Schleife mit dem Schlüsselwort `auto` kombiniert, so lässt sich sehr kompakt über die Elemente eines C-Arrays, der Standard Library Container oder auch einer Initialisiererliste iterieren (Listing 3.1: Range-basierte For-Schleife). Auch wenn die automatische Typableitung mit `auto` und die praktische Initialisierung eines Containers mit Initialisiererlisten noch nicht dargestellt wurden (wie sollte dies auch möglich sein?), sollte sich ihre Anwendung intuitiv erschließen.

rangeBased-ForLoop.cpp

```
01 #include <iostream>
02 #include <map>
03 #include <vector>
04
05
06 int main(){
07
08   std::cout << "\n";
09
10   // iterating over a C-Array
11   int myArray[5] = {1, 2, 3, 4, 5};
12   for (int &x : myArray) x *= 2;
13   for (int x: myArray) std::cout << x << " ";
14   std::cout << std::endl;
15
16   // iterating over a std::vector
17   std::vector<int> vecInt({1, 2, 3, 4, 5});
18   for (int &x: vecInt) x *= 2;
19   for (int x: vecInt) std::cout << x << " ";
20   std::cout << std::endl;
21
22   // iterating over a initializer list
23   for (const auto x : {1,2,3,5,8,13,21,34}) std::cout << x << " ";
24   std::cout << std::endl;
25
26   // iterating over a initialiser list
27   std::initializer_list<std::string>initList{"Only","For",
       "Testing","Purpose"};
28   for ( const auto x: initList) std::cout << x << " ";
29   std::cout << std::endl;
30
31   //iterating over a std::map
32   std::map<std::string,std::string> phonebook{
       {"Bjarne Stroustrup","+1 (212) 555-1212"},
       {"Gabriel Dos Reis", "+1 (858) 555-9734"},
       {"Daveed Vandevoorde","+44 99 74855424"}};
```

```
33   for ( auto mapIt: phonebook) std::cout << mapIt.first << ": " <<
mapIt.second << std::endl;
34
35   std::cout << "\n";
36
37 }
```

Listing 3.1: Range-basierte For-Schleife

Tour de C++11: rangeBasedForLoop.cpp **WEBSITE**

Werden die Elemente des C-Arrays oder STL-Containers als Referenzen angenommen, so können die Elemente direkt modifiziert werden (Listing 3.1, Zeile 12 und Zeile 18). Selbst das Iterieren über ein `std::map` geht schnell von der Hand. Sehr beeindruckend ist es, die neue C++11-Syntax (Zeile 33)

```
for ( auto mapIt: phonebook) std::cout << mapIt.first << ": "
<< mapIt.second << std::endl;
```

der klassischen C++-Syntax gegenüberzustellen:

```
std::map <std::string,std::string>::iterator mapIt;
for (mapIt= phonebook.begin();mapIt!= phonebook.end();++mapIt){
     std::cout << mapIt->first << ": " <<
                  mapIt->second << std::endl;
}
```

Nun fehlt noch die Ausgabe des Programms.

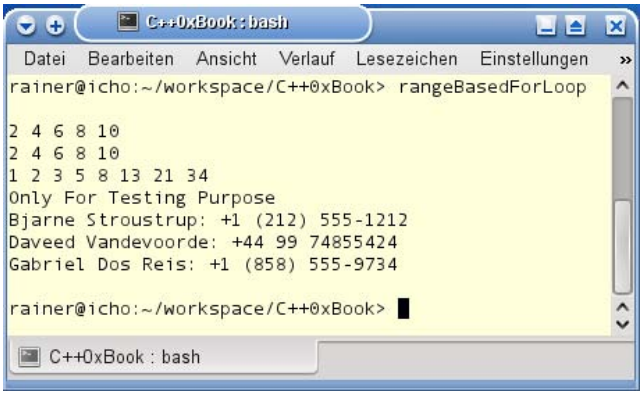


Abbildung 3.1: Range-basierte For-Schleife

Wie versprochen, das erste Geheimnis zum neuen Schlüsselwort `auto` in C++11 wird aufgelöst.

3.1.2 Das automatische Ableiten von Typen

Das automatische Ableiten von Typen, bisher vor allem aus funktionalen Sprachen wie Haskell bekannt, verbindet die dynamische Typisierung einer Interpreter- mit der statischen Typisierung einer Compiler-Sprache. Dafür führt C++11 zwei neue Schlüsselwörter ein, `auto` und `decltype`.

auto und decltype

Der feine Unterschied ist, dass `auto` den Typ automatisch aus einem Initialisierer ableitet, während `decltype` einen Ausdruck benötigt, um den Typ zur Übersetzungszeit zu ermitteln. Dabei ist diese automatische Typableitung (*type inference*) deutlich mehr als *syntactic sugar*. Können doch Rückgabewerte von Templates so komplex sein, dass es nicht trivial ist, den richtigen Typ zu spezifizieren.

DEFINITION

Syntactic sugar

Syntactic sugar bezeichnet die Syntaxerweiterung einer Programmiersprache, um ein bestimmtes Sprachfeature einfacher ausdrücken zu können. Syntactic sugar erweitert daher nicht die Funktionalität der Programmiersprache. Der Begriff geht nach Edsger W. Dijkstra (Edsger_W.Dijkstra) auf Peter J. Landin zurück.

Ein bekanntes Beispiel ist das Überladen von Operatoren. Sind die Operatoren für den eigenen Datentyp richtig implementiert, so lässt sich mit zwei Instanzen des eigenen Datentyps `a` und `b` auf natürliche Weise rechnen:

```
a * ((b * 60) + 31) + a
```

Im Gegensatz hierzu ist die äquivalente Schreibweise mit expliziten Methoden deutlich schwieriger zu lesen und daher wesentlich fehleranfälliger:

```
a.mul(b.mul(60).add(31)).add(a)
```

auto und decltype

Mit `auto` oder auch `decltype` lässt sich schnell eine neue Variable, Referenz oder auch ein Iterator auf einen Container der Standard Template Library definieren.

In den drei folgenden Listings habe ich als Erstes die aktuell gültige C++98-Syntax in Listing 3.2 verwendet. Es folgt die zukünftige C++11-Syntax, zuerst mit `decltype` in Listing 3.3 und anschließend mit `auto` in Listing 3.4.

```
int a= 5;

int b;
int& bRef= b;

const std::vector<int> v;
std::vector<int>::const_iterator itV= v.begin();
```

Listing 3.2: Variablen definieren mit C++98

```
decltype(5) a= 5;

int b;
decltype(b)& bRef = b;

const std::vector<int> v;
decltype(v.begin()) itV= v.begin();
```

Listing 3.3: Variablen definieren mit C++11 und `decltype`

```
auto a= 5;

int b;
auto& bRef= b;

const std::vector<int> v;
auto itV= v.begin();
```

Listing 3.4: Variablen definieren mit C++11 und `auto`

Es wird noch mächtiger. Um eine anonyme Funktion oder auch Lambda-Funktion in einer Variablen zu speichern, muss in klassischem C++ ein Funktionszeiger definiert werden. In C++11 reduziert sich die ganze Schreibarbeit auf das Schlüsselwort `auto`.

myAdd.cpp

```
01 #include <iostream>
02
03 int main(){
04
05   // define the function pointer
06   int (*myAdd1)(int,int)= [](int a, int b){return a + b;};
07
08   // use type inference of the C++11 compiler
09   auto myAdd2= [](int a, int b){return a + b;};
10
11   std::cout << "\n";
12
13   // use the function pointer
14   std::cout << "myAdd1(1,2)= " << myAdd1(1,2) << std::endl;
15
16   // use the auto variable
17   std::cout << "myAdd2(1,2)= " << myAdd2(1,2) << std::endl;
18
19   std::cout << "\n";
20
21 }
```

Listing 3.5: Funktionszeiger und `auto` für Lambda-Funktionen

Tour de C++11: myAdd.cpp **WEBSITE**

Neben `auto` enthält das Listing noch ein weiteres, neues Feature von C++11. Die Lambda-Funktion `[](int a, int b){return a + b;}` nimmt zwei natürliche Zahlen `a` und `b` an, addiert sie und gibt das Ergebnis zurück (Abbildung 3.2).

Abbildung 3.2: Funktionszeiger und `auto` für Lambda-Funktionen

3.1.3 Lambda-Funktionen

Lambda-Funktionen sind eine Anleihe aus der funktionalen Programmierung. Da sie Funktionen ohne Namen sind, werden sie auch gerne anonyme Funktionen genannt.

Die Struktur einer Lambda-Funktion ist schnell erklärt.

$$[\,](\,)_{optional} \rightarrow_{optional} \{\}$$

Komponente	Bereich der Lambda-Funktion
[ ]	Bindung an die Variablen des lokalen Bereichs [] keine Bindung [=] die Werte werden kopiert [&] die Werte werden referenziert
( )	Argumente des Funktionskörpers (optional)
->	Rückgabewert (optional)
{ }	Funktionskörper

Tabelle 3.1: Struktur einer Lambda-Funktion in C++11

INFO

Die Details zu Lambda-Funktionen folgen im Kapitel Kernsprache.

Streng genommen sind Lambda-Funktionen lediglich *syntactic sugar* in C++11, kann mit ihnen doch nichts ausgedrückt werden, was mit klassischem C++ nicht schon möglich ist. Richtig eingesetzt, erhöhen sie deutlich die Lesbarkeit des Codes, da durch sie die Funktionalität genau auf den Punkt gebracht wird. Beispiel gefällig?

DEFINITION

Aufrufbare Einheit

Eine aufrufbare Einheit bezeichnet in diesem Buch eine Struktur, die sich wie eine Funktion verhält. Diese kann insbesondere aufgerufen werden.

Aufrufbare Einheiten umfassen in diesem Werk Funktionen, Referenzen und Zeiger auf Funktionen, aber auch Funktionsobjekte und Lambda-Funktionen. Aufrufbare Einheiten sind ein sehr mächtiges Konzept in C++11, denn durch sie werden die Algorithmen der Standard Template Library oder auch Threads parametrisiert.

Der Begriff *callable object* aus der Python Community verwirrt in C++, da Funktionen in C++ keine speziellen Objekte wie in Python sind.

EXKURS

Duck-Typing

Aufrufbare Einheiten erlauben es dem statisch typisierten C++, eine Flexibilität anzubieten, die nur aus dynamisch typisierten Programmiersprachen wie Python bekannt ist. Ein bisschen Duck-Typing in C++, denn alles, was sich wie eine Funktion verhält, ist eine Funktion oder um es mit James Whitcomb Rileys Gedicht auszudrücken, dem das Idiom seinen Namen verdankt:

»When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.« (»Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.«) (James Whitcomb Riley, 2010)

In Listing 3.6 wird ein Vektor von Strings sortiert, wobei das Sortierkriterium die Länge der Strings ist. Das erste Sortierkriterium ist die Funktion `lessLength`, die an die Sortierfunktion in Zeile 26 `std::sort(myStrVec.begin(),myStrVec.end(),lessLength)` übergeben wird. Das Funktionsobjekt `GreaterLength` in Zeile 11 kommt in der nächsten Sortierroutine in Zeile 31 zum Einsatz. Die Lambda-Funktion in Zeile 36 bringt es ohne Definition einer Funktion oder eines Funktionsobjekts direkt auf den Punkt.

Die Details rund um Funktionsobjekte sind im Anhang genauer erklärt.

mySort.cpp

```
01 #include <algorithm
02 #include <iostream>
03 #include <iterator>
04 #include <string>
05 #include <vector>
06
07 bool lessLength(const std::string& f, const std::string& s){
08   return f.length() < s.length();
```

```
09 }
10
11 class GreaterLength{
12   public:
13     bool operator()(const std::string& f,
                       const std::string& s) const{
14       return f.length() > s.length();
15     }
16 };
17
18 int main(){
19
20   // initializing with a initializer lists
21   std::vector<std::string> myStrVec=
       {"12345","123456","1234","1","12","123","12345"};
22
23   std::cout << "\n";
24
25   // sorting with the function
26   std::sort(myStrVec.begin(),myStrVec.end(),lessLength);
27   std::copy(myStrVec.begin(),myStrVec.end(),
       std::ostream_iterator<std::string>(std::cout, " "));
28   std::cout << "\n";
29
30   // sorting with the function object
31   std::sort(myStrVec.begin(),myStrVec.end(),
       GreaterLength());
32   std::copy(myStrVec.begin(),myStrVec.end(),
       std::ostream_iterator<std::string>(std::cout, " "));
33   std::cout << "\n";
34
35   // sorting with the lambda function
36   std::sort(myStrVec.begin(),myStrVec.end(),
       [](const std::string& f,const std::string& s)
       {return f.length() < s.length();});
37   std::copy(myStrVec.begin(),myStrVec.end(),
       std::ostream_iterator<std::string>(std::cout, " "));
38   std::cout << "\n";
39
40   // using the lambda function for output
41   std::for_each(myStrVec.begin(), myStrVec.end(),
       [](const std::string& s {std::cout << s << ",";});
42
43   std::cout << "\n\n";
44
45 }
```

Listing 3.6: Sortieren mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion

WEBSITE Tour de C++11: mySort.cpp

Noch ein paar Worte zur Ausgabe des Programms auf der Konsole. Durch `std::copy` ist es möglich, die Ausgabe direkt nach `std::cout` zu kopieren (Zeile 32). Das geht mit Lambda-Funktionen einfacher. In Zeile 41 benutze ich `std::for_each`, um die Strings direkt nach `std::cout` zu schreiben.

Die Ausgabe des Programms zeigt die sortierten Strings.

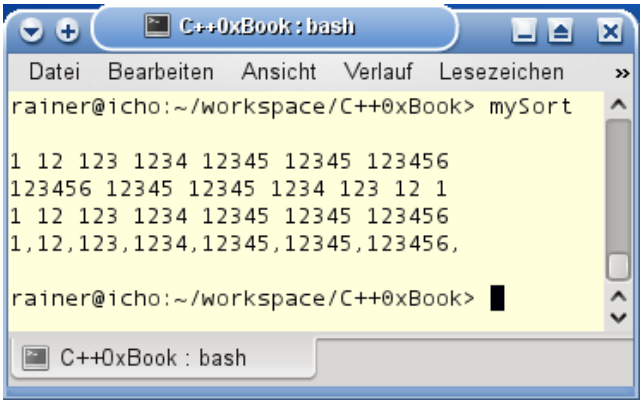


Abbildung 3.3: Sortieren mit einer Funktion, einem Funktionsobjekt und einer Lambda-Funktion

Dem aufmerksamen Leser wird die sehr kompakte Definition eines Vektors `myStrVec={"12345","123456",... ,"12345"}` in Listing 3.6 nicht entgangen sein. Durch Initialisiererlisten wird das Initialisieren von Datentypen nicht nur für den C++-Novizen deutlich einfacher in C++11. Das moderne C++ hat einiges rund um die vereinheitlichte Initialisierung zu bieten.

3.1.4 Vereinheitlichte Initialisierung

Strukturen, Arrays und Container

Die Initialisierung von Objekten in klassischem C++ setzt einiges Wissen voraus, gibt es doch viele verschiedene Arten, diese zu initialisieren. So lassen sich die C-Strukturen `struct` und Arrays über Initialisiererlisten initialisieren (Listing 3.7, Zeile 7 und 10), C++-Container der Standard Template Library aber nicht. Als Alternative bietet es sich an, jedes Element beim `std::vector` einzeln (Listing 3.7, Zeile 14 – 18) oder die Elemente indirekt über ein Array (Listing 3.7, Zeile 21) zu initialisieren.

```
01 struct MyStruct{
02   int a;
03   double b;
04 };
05
06 // direct initialization with initializer list
07 MyStruct myStruct = {4,5.5};
```

```
08
09 // direct in initialization with initializer list
10 int intArray[]= {1,2,3,4,5};
11
12 // elementwise initialization
13 std::vector <int> myIntVec;
14 myIntVec.push_back(1);
15 myIntVec.push_back(2);
16 myIntVec.push_back(3);
17 myIntVec.push_back(4);
18 myIntVec.push_back(5);
19
20 // using intArray for initialization
21 std::vector<int> myIntVec2(intArray,intArray+4);
```

Listing 3.7: Strukturen, Arrays und Vektoren initialisieren

Konstante Element- und Heap-Arrays

Es wird noch komplizierter. C++ kann kein Array `myData` als Datenelement und ein konstantes Heap-Array `pData` initialisieren (Listing 3.8), da es dafür keine Syntax gibt.

```
// impossible to initialize myData
class Array{
  public:
    Array(): myData( ... ) {}
  private:
    int myData[5];
};

// impossible to initialize pData
int* const pData = new const int[5];
```

Listing 3.8: C++ kann keine Element- und Heap-Arrays initialisieren.

{}-Initialisiererlisten

Hier räumt C++11 auf. C++11 erlaubt {}-Initialisiererlisten für alle Initialisierungen. Damit ist die Initialisierung von Strukturen, Arrays und Containern vereinheitlicht und Element- und konstante Heap-Arrays lassen sich in C++11 einfach initialisieren.

uniformInitialisation.cpp

```
01 #include <vector>
02
03 struct MyStruct{
04   int a;
05   double b;
06 };
07
08 class Array{
09   public:
10     Array(): myData{1,2,3,4,5} {}
11   private:
12     int myData[5];
13 };
```

```
14
15 int main(){
16
17   // valid for C++
18   MyStruct myStruct = {4,5.5};
19
20   // valid for C++
21   int invArray[]= {1,2,3,4,5};
22
23   // valid for C++11
24   std::vector <int> myIntVec{1,2,3,4,5};
25
26   // valid for C++11
27   Array myArray;
28
29   // valid for C++11
30   const float* pData = new const float[5]{1,2,3,4,5};
31 }
```

Listing 3.9: Vereinheitlichte Initialisierung mit {}-Initialisiererlisten

WEBSITE

Tour de C++11: uniformInitialisation.cpp

Da stellt sich natürlich die Frage, was muss ein Datentyp bieten, damit er mit Initialisiererlisten initialisiert werden kann. Diese Frage führt uns direkt zum nächsten Kapitel.

3.2 Entwurf von Klassen

Der Entwurf von Klassen wird in C++11 viel mächtiger und expliziter. Einerseits gibt es neue Features rund um die Definition von Konstruktoren, andererseits können Methoden mit Bezeichnern annotiert werden, so dass der Compiler dies prüft.

3.2.1 Mächtigere Initialisierung

Neben der Initialisiererliste für Konstruktoren, die wir im letzten Kapitel in Aktion gesehen haben, unterstützt C++11 jetzt auch deren Delegation und Vererbung. Aber nicht nur der Umgang mit Konstruktoren ist mächtiger, einfacher und mit weniger Schreibaufwand verbunden, auch das direkte Initialisieren von Klassenelementen ist jetzt möglich.

Initialisiererliste-Konstruktor

Initialisiererliste-Konstruktoren sind der Widerpart zu den Initialisiererlisten in Listing 3.9. Das Schöne ist, dass die Container der Standard Template Library diese speziellen Konstruktoren schon definiert haben, so dass ein Vektor über eine Initialisiererliste direkt initialisiert werden kann. Aber

auch eigene Datentypen lassen sich mit diesen Konstruktoren einfach ausstatten (Listing 3.10).

initializerList-Constructor.cp

```
01 #include <iostream>
02 #include <map>
03 #include <string>
04
05 // class template, parametrized with T
06 template <typename T>
07 class MyContainer{
08   public:
09     MyContainer(std::initializer_list<T> values){
10
11       for (auto v : values) std::cout << v << " ";
12
13     }
14 };
15
16 int main(){
17
18   // using a initialiser list for a string
19   std::string cppInventor={"Bjarne Stroustrup"};
20
21   std::cout << "\n";
22   std::cout << "Name of the cpp Inventor: "
               << cppInventor << std::endl;
23
24   // using a initializer list for a map
25   std::cout << "\nA few import cpp developer: "
               << std::endl;
26   std::map<std::string,std::string> phonebook{
       {cppInventor,"+1 (212) 555-1212"},
       {"Gabriel Dos Reis", "+1 (858) 555-9734"},
       {"Daveed Vandevoorde","+44 99 74855424"}};
27
28   for (auto mapIt= phonebook.begin();
         mapIt!= phonebook.end();++mapIt){
29     std::cout << mapIt->first << ": "
                 << mapIt->second << std::endl;
30   }
31
32   std::cout << "\n";
33
34   // using MyContainer with int
35   MyContainer<int> myIntCont{1,2,3,4,5,6,7,8,9,10};
36   std::cout << "\n";
37
38   // using MyContainer with string
39   MyContainer<std::string>
        myStringCont{"Range","based","for","loop."};
40
```

```
41   std::cout << "\n\n";
42
43 }
```

Listing 3.10: Initialisiererlisten-Konstruktor

WEBSITE

Tour de C++11: InitialiserListConstructor.cpp

Sowohl ein String (Zeile 19) als auch der Standardcontainer `std::map` (Zeile 26) in Listing 3.10 lassen sich über eine Initialisiererliste initialisieren. Das Klassen-Template `MyContainer` in Zeile 6 nimmt als Argument eine Initialisiererliste von Ganzzahlen und Strings (Zeile 35 und 39) an. Im Initialisiererlisten-Konstruktor von `MyContainer` (Zeile 9) wird die Initialisiererliste direkt ausgegeben. Hier sehen wir eine generische Range-based For-Schleife im Einsatz. In Kombination mit `auto` lässt sich so äußerst kompakt über STL-Container iterieren.

Nun fehlt nur noch die Ausgabe des Programms.

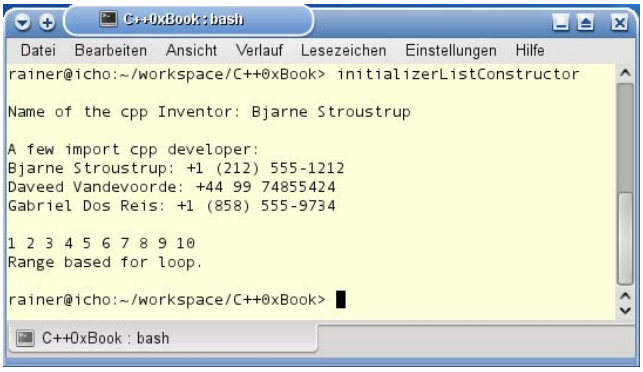


Abbildung 3.4: Initialisiererliste-Konstruktor

Delegation von Konstruktoren

Java oder auch D kennen die Delegation von Konstruktoren. C++ führt sie mit C++11 ein.

Besitzt in klassischem C++ eine Klasse mehrere Konstruktoren, die ähnliche Initialisierungsschritte ausführen müssen, gibt es zwei Lösungen. Die naheliegende Lösung ist es, den Initialisierungscode in jedem Konstruktor zu duplizieren. Dies ist fehleranfällig und mit viel Schreibaufwand verbunden. Da ist es schon deutlich besser, eine private Methode `init` zu definieren, in diese den gemeinsamen Code auszulagern und die Initialisierungsmethode in jedem Konstruktor aufzurufen.

Im neuen C++11 kann der Initialisierungscode in einem Konstruktor definiert werden, der dann von allen anderen Konstruktoren verwendet wird.

delegating-Constructor.cpp

```
01 #include <cmath>
02 #include <iostream>
03
04 class MyHour{
05   int myHour_;
06   public:
07
08     // constructor validating the data
09     MyHour(int hour){
10       if (0 <=hour and (hour<=23)) myHour_= hour;
11       else myHour_=0;
12       std::cout << "hour= " << hour << std::endl;
13     }
14
15     // default constructor for setting hour to 0
16     MyHour(): MyHour(0){};
17
18     // accept also doubles
19     MyHour(double hour)
         :MyHour( static_cast<int>(ceil(hour))) {};
20
21 };
22
23 int main(){
24
25   // use the validating constructor
26   MyHour(10);           // hour= 10
27
28   // use the validating constructor
29   MyHour(100);          // hour= 0
30
31   // use the default constructor
32   MyHour();             // hour= 0
33
34   // use the constructor accepting doubles
35   MyHour(22.45);        // hour= 23
36
37 }
```

Listing 3.11: Delegation von Konstruktoren

WEBSITE Tour de C++11: delegatingConstructor.cpp

Der Konstruktor `MyHour(int hour)` (Listing 3.11, Zeile 9) validiert seinen Eingabewert. Daher können die zwei folgenden Konstruktoren in Zeile 16 und 19 ihre Validierung der Daten direkt an diesen durch `MyHour():MyHour(0)` beziehungsweise `MyHour(double hour):MyHour(…)`delegieren.

Da der aktuelle GCC 4.6 (C++0x Support in GCC) weder die Delegation und Vererbung von Konstruktoren noch die direkte Initialisierung von Klassenelementen unterstützt, sind die Ausgaben in den entsprechenden Kommentaren der `main`-Funktion enthalten. Dies betrifft Listing 3.11, Listing 3.12 und Listing 3.13.

Vererbung von Konstruktoren

inheritingConstructor.cpp

Dieses Vererben von Konstruktoren erspart einige Schreibarbeit. Ein einfaches `using Base::Base` in der Definition der Klasse `Derived` (Listing 3.12) reicht aus und alle Konstruktoren der Basisklasse stehen in der abgeleiteten Klasse zur Verfügung.

```
01 #include <iostream>
02 #include <string>
03
04 class Base{
05  public:
06
07     Base(int i){
08       std::cout << "Base::Base("<< i << ")" << std::endl;
09     }
10
11     Base(std::string s){
12       std::cout << "Base::Base("<< s << ")" << std::endl;
13     }
14 };
15
16 class Derived: public Base{
17   public:
18
19     using Base::Base;
20
21     Derived(double d){
22       std::cout << "Derived::Derived("<< d << ")" << std::endl;
23     }
24
25 };
26
27 int main(){
28
29   // inheriting Base
30   Derived(2011);          // Base::Base(2011)
31
32   // inheriting Base      // Base::Base(C++11)
33   Derived("C++11");
34
35   // using Derived
36   Derived(0.33);          // Derived::Derived(0.33)
37
38 }
```

Listing 3.12: Vererben von Konstruktoren

WEBSITE Tour de C++11: inheritingConstructor.cpp

Direktes Initialisieren der Klassenelemente

Aber nicht nur das Initialisieren mit Hilfe des Konstruktors, auch das direkte Initialisieren der Klassenelemente wird in modernem C++ unterstützt. Konnten in C++98 nur statische, konstante Elemente integralen Typs initialisiert werden, so gilt die Einschränkung in C++11 nicht mehr. Diese Einschränkung stellte sicher, dass die Initialisierung zur Übersetzungszeit möglich ist. Falls ein Klassenelement sowohl direkt als auch über den Konstruktor initialisiert wird, wird nur Letzteres angewandt.

Ein paar Beispiele zeigen die neue Funktionalität. Der Einfachheit halber verwende ich den Datentyp `struct`, da hier alle Klassenelemente öffentlich sind.

```
01 #include <iostream>
02 #include <string>
03 #include <vector>
04
05 struct ClassMemberInitializer{
06
07     ClassMemberInitializer(int override):x(override){};
08
09     //valid with C++98
10     const static int oldX=5;
11
12     // valid with C++11
13     int X=5; //class member initializer
14
15     // valid with C++11
16     std::string s="Hello C++11";
17
18     // valid with C++11
19     std::vector<int> myVec{1,2,3,4,5};
20
21 };
22
23
24 int main(){
25
26     std::cout << "\n";
27
28     // class member initialization
29     ClassMemberInitializer cMI;
30     std::cout << "cMI.oldX " << cMI.oldX << "\n";     // 0
31     std::cout << "cMI.x " << cMI.x << "\n";           // 5
32     std::cout << "cMI.s " << cMI.s << "\n";   // Hello C++11
33     for (auto vecIt= CMi.myVec.begin(), myVec)
34                     std::cout << *vecIt << " ";  //1 2 3 4 5
35
36     std::cout << "\n";
```

```
37
38     // class member initialization
39     // x will be overriden by the constructor value
40     ClassMemberInitializer cMI2(10);
41     std::cout << "cMI2.oldX " << cMI2.oldX << "\n";   // 0
42     std::cout << "cMI2.x " << cMI2.x << "\n";         // 10
43     std::cout << "cMI2.s " << cMI2.s << "\n"; // Hello C++11
44     for (auto vecIt= CMi2.myVec.begin(), myVec)
45                     std::cout << *vecIt << " ";  // 1 2 3 4 5
46
47     std::cout << "\n\n";
48
49 }
```

Listing 3.13: Direkte Initialisierung der Klassenelemente

WEBSITE

Tour de C++11: classMemberInitializer.cpp

Gibt es rund um die Initialisierung von Objekten und Klassenelementen viele neue Features, so können Methoden in C++11 mit Bezeichnern annotiert werden, die das Verhalten der Methode explizit beschreiben.

3.2.2 Explizite Klassendefinitionen

Wieso soll Pythons Designprinzip »*Explicit is better then implicit*.« von Tim Peters (Peters, 2004) nicht auch für C++11 gelten?

Methoden können in modernem C++ mit den Identifiers `default`, `delete`, `override`, `final` oder auch `explicit` ausgezeichnet werden. Der Compiler sorgt dafür, dass der Vertrag eingehalten wird.

default und delete

Für eine Klasse werden viele spezielle Methoden vom Compiler erzeugt, um den Lebenszyklus seiner Instanzen zu gewährleisten. Dies betrifft den Standard- und den Kopierkonstruktor, den Zuweisungsoperator und den Destruktor. Aber auch spezielle Methoden wie der `operator new` werden vom Compiler bei Bedarf erzeugt.

Für C++-Entwickler gibt es viele Idiome in klassischem C++, um den Lebenszyklus eines Objekts direkt zu kontrollieren. Diese Idiome setzen viel Wissen und noch mehr Disziplin voraus. Mit den Bezeichnern `default` und `delete` hat sich der C++-Standard dieser Problematik angenommen.

EXKURS

Design Pattern und Idiome

Ein Design Pattern oder auch Entwurfsmuster beschreibt eine bewährte Lösung für ein in einem bestimmten Kontext wiederkehrendes Entwurfsproblem. Bekannte Design Pattern sind das Singleton Pattern, die Template-Methode oder auch das Visitor Pattern. Sie gehen auf den Architekten und Philosophen Christopher Alexander (Christopher Alexander, 2011) zurück. Richtig populär sind Design Pattern aber erst seit dem Buch »Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software« von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (Entwurfsmuster (Buch), 2011), gemeinhin bekannt als die Gang of Four (Gof).

Ist ein Design Pattern darauf ausgelegt, eine Lösung unabhängig von der verwendeten Sprache anzubieten, so beschreibt ein Idiom ein bewährtes programmiersprachenspezifisches Lösungsrezept.

Soll eine Klasse zum Beispiel nicht kopierbar sein, werden der Kopierkonstruktor und der Zuweisungsoperator lediglich privat deklariert, aber nicht definiert. Hier setzt der `delete`-Bezeichner an. Ein Name `delete` hinter dem Methodennamen und die Methode wird vom Compiler nicht mehr erzeugt.

Aber es lauern auch Gefahren rund um den Lebenszyklus eines Objekts. Wird in einer Klasse ein Konstruktor definiert, erzeugt der Compiler keinen Standardkonstruktor mehr. Genau dies kann der Klassendesigner dem Compiler aber durch den Bezeichner `default` vorschreiben. `default` sorgt dafür, dass der Compiler seine Default-Version der Methode erzeugt.

Der Lebenszyklus eines Objekts lässt sich in C++11 rein deklarativ beschreiben.

defaultedDeleted-Methods.cpp

```
01 #include <iostream>
02
03 class NonCopyableClass{
04   public:
05
06     // state the compiler generated default constructor
07   NonCopyableClass()= default;
08
09     // disallow copying
10   NonCopyableClass& operator=
11                   (const NonCopyableClass&)= delete;
12   NonCopyableClass(const NonCopyableClass&)= delete;
13 };
14
15 class SomeType{
16   public:
17
18   // state the compiler generated default constructor
19   SomeType()= default;
20
21   // constructor for int
```

```
22   SomeType(int value){};
23
24 };
25
26 class TypeOnStack {
27   public:
28
29     void* operator new(std::size_t)= delete;
30 };
31
32 int main(){
33
34   NonCopyableClass nonCopyableClass;
35   SomeType someType;
36   TypeOnStack typeOnStack;
37
38   // force the compiler error
39   NonCopyableClass nonCopyableClass2(nonCopyableClass);
40
41   // force the compiler error
42   TypeOnStack* typeOnHeap= new TypeOnStack;
43
44 }
```

Listing 3.14: `default` und `delete` für Methoden

Tour de C++11: defaultedDeletedMethods.cpp **WEBSITE**

Vor der Übersetzung des Programms in Listing 3.14 noch ein paar Worte zum Sourcecode.

In der Klasse `NonCopyableClass` (Zeile 3) werden sowohl der Kopier-Zuweisungsoperator als auch der Kopierkonstruktor als `delete` erklärt und der Standardkonstruktor des Compilers verwendet. Der Standardkonstruktor wird in der Klasse `NonCopyableClass` nicht automatisch vom Compiler erzeugt, da der Kopierzuweisungsoperator und der Kopierkonstruktor als `delete` deklariert wurden. Damit lassen sich Objekte der Klasse nur direkt instanziieren. Auch die Klasse `SomeType` (Zeile 15) nutzt den vom Compiler erzeugten Standardkonstruktor, den der Compiler in diesem Fall nicht erzeugt, da ein spezieller Konstruktor für `int` vorhanden ist. Interessant ist auch die Klasse `TypeOnStack` (Zeile 26), denn das Setzen des `new`-Operators auf `delete` bewirkt, dass deren Instanzen nicht mehr mit `new` erzeugt werden können. Der interessanteste Teil des Programms ist aber die `main`-Funktion, denn in ihr wird sowohl ein nicht kopierbares Objekt kopiert als auch ein Objekt auf dem Heap angelegt, obwohl dessen `new`-Operator auf `delete` gesetzt ist.

Wie geht der GCC-Compiler mit dem Vertragsbruch um?

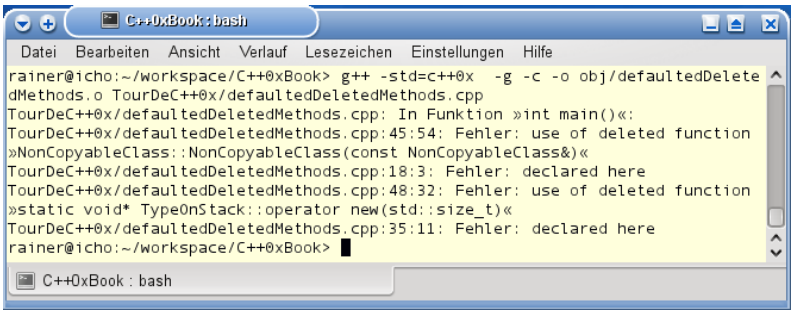


Abbildung 3.5: Compiler-Fehler beim Übersetzen von defaultedDeletedMethods.cpp

Der GCC 4.6 schreibt eine aussagekräftige Fehlermeldung. Was will man mehr?

Explizite Virtualität

Es bleibt deklarativ. Eine beliebte Fehlerquelle beim Überschreiben von virtuellen Funktionen ist, dass die Signatur der neuen Methode nicht der der überschriebenen Methode entspricht. Das Ergebnis zeigt sich erst sehr viel später zur Laufzeit, wenn sich das Programm unerwartet verhält. Diese Fehlerquelle lässt sich mit C++11 elegant beseitigen. Wird die neue Funktion mit dem Schlüsselwort `override` versehen, stellt der Compiler sicher, dass diese auch tatsächlich eine Methode der Basisklasse überschreibt. Der Compiler stellt auch sicher, dass Methoden, die als `final` deklariert sind, nicht überschrieben werden können.

Vom Novizen zum Profi

Bisher hatte C++11 viel für den C++-Novizen zu bieten: das automatische Ableiten von Typen, die vereinheitlichte Initialisierung von Datentypen, die mächtigere Initialisierung von Objekten, die deklarativen Klassendefinitionen. Selbst Lambda-Funktionen sind, ist die erste Fremdheit überwunden, einfach zu schreiben und zu lesen. Nun ist der Profi an der Reihe. In den nächsten beiden Themenblöcken Rvalue-Referenzen und Generische Programmierung findet dieser die Werkzeuge, um seine Datentypen und Bibliotheken genau auf seine Bedürfnisse abzustimmen.

3.3 Rvalue-Referenzen

Zwei spezielle Methoden sind im Kapitel Mächtigere Initialisierung nicht genannt worden: der neue Move-Konstruktor und der Move-Zuweisungsoperator, den jeder STL-Container und auch der Datentyp String besitzt.

Betrachten wir die vereinfachte Implementierung des `std::vector`-Containers in Listing 3.15, dann fällt auf, dass neben dem klassischen Kopierkonstruktor (Zeile 6) und Zuweisungsoperator (Zeile 7) zwei sehr ähnliche, neue Methodendeklarationen in Zeile 10 und 11 existieren. Der auffälligste Unterschied ist, dass die traditionellen Methoden ihre Argumente als

Lvalue-Referenz mit einem & annehmen, während die neuen Methoden ihre Argumente als Rvalue-Referenz mit && annehmen.

```
01 template<typename T>
02 class vector {
03   public:
04
05     // copy semantic
06     vector(const vector& v);
07     vector& operator=(const vector& v);
08
09     // move semantic
10     vector(vector&& v);
11     vector& operator=(vector&& v);
12
13     // …
14 }
```

Listing 3.15: Copy-Semantik und Move-Semantik

Beides sind Referenzen im klassischen C++-Sinn. Der C++-Compiler entscheidet aber, welche Implementierung bei der Konstruktion oder auch Zuweisungsoperation verwendet wird, denn mit Lvalue-Referenzen wird die klassische und bekannte Copy-Semantik implementiert, mit Rvalue-Referenzen die neue Move-Semantik.

PRAXISTIPP

Unterscheiden Sie die Copy- von der Move-Semantik.

Beim Erzeugen von neuen aus bestehenden Objekten werden die Inhalte der Objekte bei der Move-Semantik transferiert, hingegen bei der Copy-Semantik kopiert. Damit wird bei der Move-Semantik kein neues Objekt erzeugt.

Verwirrt? Verständlich! Das kleine Beispiel in Listing 3.16 soll für Aufklärung sorgen.

copyMoveSemantic.cpp

```
01 #include <iostream>
02 #include <string>
03
04 int main(){
05
06   std::string str1{"ABCEF"};
07   std::string str2;
08
09   std::cout << "\n";
10
11   // initial value
12   std::cout << "str1= " << str1 << std::endl;
13   std::cout << "str2= " << str2 << std::endl;
14
15   // copy semantik
16   str2= str1;
```

```
17   std::cout << "str2= str1;\n";
18   std::cout << "str1= " << str1 << std::endl;
19   std::cout << "str2= " << str2 << std::endl;
20
21   std::cout << "\n";
22
23   std::string str3;
24
25   // initial value
26   std::cout << "str1= " << str1 << std::endl;
27   std::cout << "str3= " << str3 << std::endl;
28
29   // move semantik
30   str3= std::move(str1);
31   std::cout << "str3= std::move(str1);\n";
32   std::cout << "str1= " << str1 << std::endl;
33   std::cout << "str3= " << str3 << std::endl;
34
35   std::cout << "\n";
36
37 }
```

Listing 3.16: Copy- und Move-Semantik im Vergleich

WEBSITE Tour de C++11: copyMoveSemantic.cpp

Die neue C++11-Funktion `std::move` in Zeile 30 hat zur Folge, dass der String `str1` als Rvalue vom C++-Compiler interpretiert wird. Damit wird der Inhalt von `str1` nach `str3` transferiert.

In den nächsten zwei Abbildungen ist der Unterschied zwischen der Copy- und der Move-Semantik dargestellt. Während nach dem Kopieren sowohl die Quelle `str1` als auch das Ziel `str2` den gleichen Inhalt besitzen, ist die Quelle `str1` nach dem Transferieren des Inhalts leer.

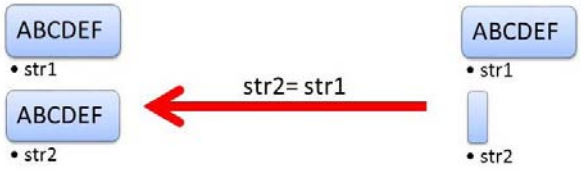


Abbildung 3.6: Copy-Semantik

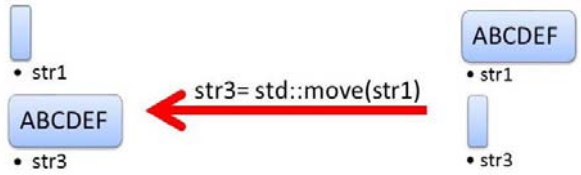


Abbildung 3.7: Move-Semantik

Die Ausgabe des Programmlaufs zeigt die Ergebnisse auf der Konsole.

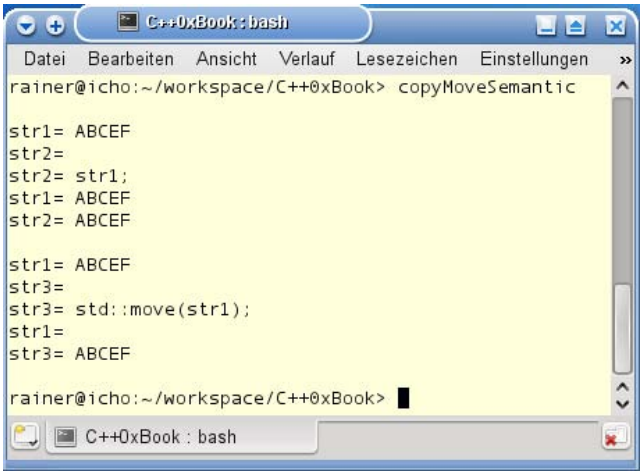


Abbildung 3.8: Copy- und Move-Semantik

Die Grundlage für diese Optimierungen sind die Rvalues. Anhand dieser kann der C++-Compiler entscheiden, welche Implementierung der Methode verwendet werden soll.

PRAXISTIPP

Rvalues besitzen keinen Namen.

Besitzt ein Objekt einen Namen, ist es ein Lvalue, ansonsten ein Rvalue.

Perfect Forwarding

Wie der C++-Entwickler eigene Datentypen entwirft, die mit Move-Semantik ausgestattet sind, das wird in dem Buchabschnitt Kernsprache erläutert. Dies gilt auch für *Perfect Forwarding*, bei dem Argumente an eine andere Funktion weitergegeben werden, ohne ihre Lvalue- oder Rvalue-Eigenschaft zu verändern. Ein bisher ungelöstes Problem in C++.

Weiter geht es mit neuen Features für den C++-Profi. Das Programmieren mit Templates wird deutlich mächtiger in C++11.

3.4 Generische Programmierung

Das generische Programmieren, auf dem die Standard Template Library basiert, ist in C++ ein wichtiges Paradigma. Daher verwundert es nicht, dass hier C++11 einiges Neues zu bieten hat:

» Templates, die beliebig viele Parameter annehmen,

» Zusicherungen, die zur Compile-Zeit ausgewertet werden,

» Konstanten, die zur Compile-Zeit evaluiert werden,

» Aliase Templates, um einfache Namen für teilweise gebundene Templates zu definieren.

3.4.1 Variadic Templates

Variadic Templates erlauben es in C++11, Templates zu schreiben, die beliebig viele Argumente annehmen können. Ein prominentes Beispiel für den Einsatz von Variadic Templates ist der neue heterogene, sequentielle Datentyp `std::tuple`, der eine beliebige Länge haben kann.

Da die neue Syntax der Variadic Templates doch recht ungewohnt wirkt, zuerst ein einfaches Beispiel. Das Funktions-Template `countMe` in Listing 3.17 zählt die Anzahl seiner Argumente.

countMe.cpp

```
01 #include <iostream>
02 #include <list>
03
04 template <typename ... Args>
05 int countMe(Args ... args){
06   return (sizeof ... args);
07 }
08
09 int main(){
10
11   std::cout << "\n";
12
13   std::list<int> myList{1,2,3,4,5,6,7,8,9};
14
15   std::cout << "countMe() has " << countMe()
                << " arguments" << std::endl;
16   std::cout << "countMe(\"one\", 3.14 , myList ) has "
                << countMe("one", 3.14 , myList )
                << " arguments" << std::endl;
17   std::cout << "countMe(myList) has " << countMe(myList)
                << " argument" << std::endl;
18
```

```
19   std::cout << "\n";
20
21 }
```

Listing 3.17: Variadic Templates und `sizeof`

Tour de C++11: countMe.cpp **WEBSITE**

Ungewohnt an dem Funktions-Template `countMe` sind zuallererst die drei Punkte `...`. Dabei gilt es aufmerksam zu sein, ob diese links `<typename ... Args>` (Zeile 4) oder rechts `(Args ... args)` (Zeile 5) von `Args` stehen. Links packt der Ellipsen-Operator `...` das sogenannte Parameter Pack, rechts entpackt er es wieder. Neu ist auch der Operator `sizeof ...` (Zeile 6), der direkt mit Parameter Packs umgehen kann.

Jetzt fehlt nur noch die Ausgabe des Programms.

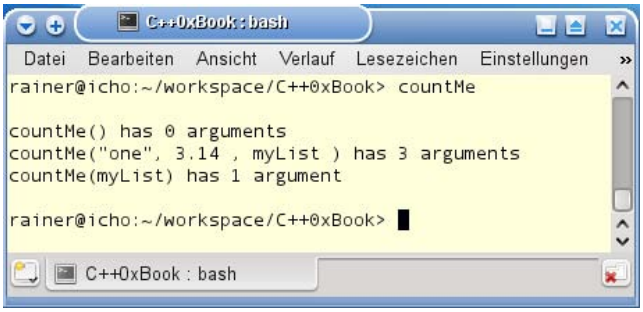


Abbildung 3.9: Variadic Templates und `sizeof`

Das Programm `countMe` in Listing 3.17 ist aber nicht der typische Anwendungsfall für Variadic Templates. Deutlich typischer ist das folgende Muster aus der funktionalen Programmierung, wenn es um die Verarbeitung beliebig langer Listen geht.

Dazu wird die Liste in zwei Teile `first` und `rest` getrennt. Dabei bezeichnet:

» first: das erste Element der Liste,

» rest: den Rest der Liste.

Für beide Bereiche der Liste werden zwei Aktionen registriert.

1. Aktion (first): Verarbeite first.
2. Aktion (rest): Verarbeite first und führe Aktion(rest) auf der verbleibenden Liste aus.

Der Trick ist, dass `rest` bei jeder Rekursion um das erste Element `first` gekürzt wird.

Genau diesem Muster folgt das Programm in Listing 3.18. Als Aktion auf dem Kopf der Liste `first` wird dessen Typinformation und Größe herausgegeben.

printInfo.cpp

```
01 #include <iomanip>
02 #include <iostream>
03 #include <typeinfo>
04
05 template <typename T>
06 void printInfoFor(T value){
07
08   std::cout << std::boolalpha;
09   std::cout << std::setw(5) << value << ": " << "(type: "
                 << std::setw(3) << typeid(value).name()
                 << ",size: " << sizeof(value) << ")\n";
10
11 }
12
13 template<typename T>
14 void printValueInfo(T value){
15
16     // print the information of the value
17     printInfoFor(value);
18
19 }
20
21 template<typename First,typename ... Rest>
22 void printValueInfo(First first,Rest ... rest){
23
24     // print the information of the value
25     printInfoFor(first);
26
27     // invoke value Information for the rest,
          excluding first
28     printValueInfo(rest...);
29
30 }
31
32 int main(){
33
34   std::cout << std::endl;
35
36   printValueInfo(); // => compile error
37
38   printValueInfo(true,42,2.3,'c',"C++11");
39
40   std::cout << std::endl;
41
42 }
```

Listing 3.18: Typinformation und -größe eines Werts

WEBSITE

Tour de C++11: printInfo.cpp

In Zeile 38 wird die Aktion `printValueInfo` auf der ganzen Liste angestoßen. Die Liste besitzt mehr als ein Element, so dass das Funktions-Template `printValueInfo(First first, Rest ... rest )` in Zeile 22 aufgerufen wird. Dabei wird `true` an `first` und die verbleibenden Argumente werden an `rest` gebunden. `true` wird über die Hilfsfunktion `printInfoFor` (Zeile 5) ausgegeben und `rest` wird rekursiv wieder aufgerufen (Zeile 28). Die Rekursion terminiert, sobald `rest` nur noch ein Element besitzt, denn in diesem Fall wird das Funktions-Template für ein Argument `printValueInfo(T value)` in Zeile 13 aufgerufen.

ACHTUNG

In der funktionalen Programmierung hat sich für das Paar (first,rest) einer Liste das Namenspaar (head,tail) oder auch (car,cdr) etabliert.

Die Ausgabe zeigt die Typ- und Größeninformationen der Werte.

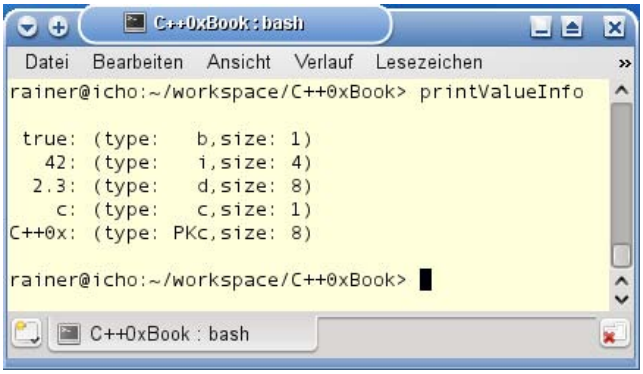


Abbildung 3.10: Typ- und Größeninformation von Werten

ACHTUNG

Leider ist das vorgestellte Programm nicht sehr robust. Zum einen setzt es voraus, dass `printValueInfo` mindestens ein Argument erhält, und zum anderen, dass die Argumente direkt auf `std::cout` ausgegeben werden können.

Wird das Programm mit den falschen oder keinen Argumenten aufgerufen, moniert dies der Compiler sofort mit einer Fehlermeldung.

Es gilt als Codierungsstandard in C++, beschrieben von Herb Sutter und Andrei Alexandrescu in ihrem Buch »C++ Coding Standards« (Sutter & Alexandrescu, 2005), Fehler zur Übersetzungszeit denen zur Laufzeit vorzuziehen. C++11 führt für die Zusicherung zur Übersetzungszeit das neue Schlüsselwort `static_assert` ein.

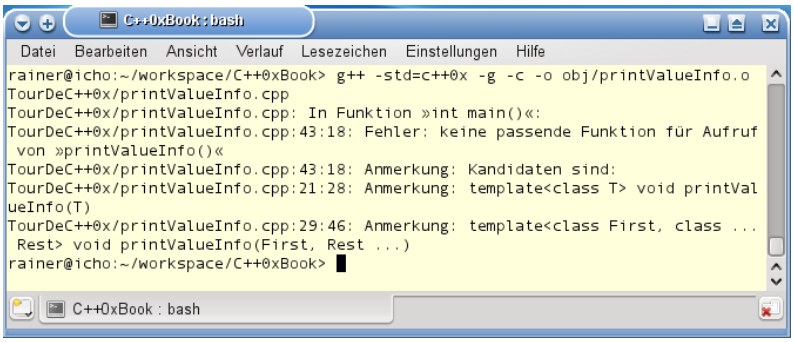


Abbildung 3.11: Compiler-Fehler beim Übersetzen von printValueInfo.cpp

3.4.2 Zusicherungen zur Compile-Zeit

static_assert

Zwar gibt es schon Werkzeuge in C++, um Zusicherung an den Programmcode zu formulieren. So wirkt die Präprozessordirektive `#error` während der Ausführung des Präprozessors, hingegen das Makro `assert` während der Laufzeit des Programms. Die Lücke zwischen Präprozessorlauf und Ausführung des Programms schließt das neue C++11-Schlüsselwort `static_assert`, denn es wird zur Übersetzungszeit ausgeführt. Daher ist es sehr nützlich, wenn es darum geht, Bedingungen an den Template-Code zu verifizieren. Dies trifft umso mehr zu, da Concepts, ein Typsystem für Templates, aus dem aktuellen Standard entfernt wurden.

3.4.3 Aliase Templates

Dienen `static_assert` dazu, den Template-Code robuster zu machen, so adressieren *Aliase Templates* vor allem die Lesbarkeit des Codes. *Aliase Templates* erlauben es, mittels des Schlüsselworts `using` Synonyme auf Templates zu erzeugen, die ihre Template-Parameter teilweise gebunden haben.

```
01 template< typename T, int V>
02 class MyType;
03
04 template< typename T>
05 using MyType10 = MyType<T,10>;
06
07 template<typename T>
08 using VecMyAlloc = std::vector<T,MyAllocator<T>>;
```

In `MyType10` wird der zweite Template-Parameter von `MyType` mit 10 gebunden (Zeile 5). `VecMyAlloc` geht aus `std::vector` hervor, indem als zweites Element der Speicheranforderer (Zeile 8) gebunden wird.

EXKURS

Allgemeine, partiell spezialisierte und vollständig spezialisierte Templates

Die kleine Gegenüberstellung in Listing 3.19 soll die Unterschiede der drei Spezialisierungen anhand des Klassen-Template `Matrix` verdeutlichen.

```
01 template <int line, int column>
02 class Matrix{
03 . . .
04 };
05
06 template <int line>
07 class Matrix<line,1>{
08 . . .
09 };
10
11 template <>
12 class Matrix<3,3>{
13 . . .
14 };
15 . . .
16 int main(){
17
18   Matrix<3,4> rectangle;
19
20   Matrix<20,1> vector;
21
22   Matrix<3,3> cube;
23
24 };
```

Listing 3.19: Das allgemeine, partiell und vollständig spezialisierte Klassen-Template Matrix

In Zeile 1 wird das primäre oder auch allgemeine Klassen-Template definiert. In ihm sind keine Bedingungen an die Anzahl der Zeilen oder Spalten definiert. Das ändert sich mit dem partiell spezialisierten Klassen-Template in Zeile 6, das einen Vektor beschreibt. Dieses Klassen-Template drückt durch `class Matrix<line,1>` (Zeile 7) aus, dass die Anzahl der Spalten auf `1` gesetzt ist. Vollständig spezifiziert ist das Klassen-Template in Zeile 11, denn sowohl die Anzahl der Spalten als auch die Anzahl der Zeilen sind auf `3` gesetzt. Im Hauptprogramm werden die Klassen-Templates verwendet – zuerst die allgemeine (Zeile 18), dann die partiell spezialisierte (Zeile 20) und zuletzt die vollständig spezialisierte Form (Zeile 22). Die Regel, welches Klassen-Template zum Einsatz kommt, ist sehr einprägsam. Das am meisten spezialisierte Klassen-Template wird verwendet.

Die Template-Instanziierung ist ein Prozess, der zur Übersetzungszeit stattfindet. Genauso verhält es sich mit den folgenden konstanten Ausdrücken.

3.5 Erweiterte Datenkonzepte und Literale

Viele Datenkonzepte aus dem klassischen C++ wurden in C++11 aufgegriffen und erweitert. Diese Erweiterungen betreffen die konstanten Ausdrücke, die sogenannten Plain Old Data (POD), aber auch Enums. Neben den neuen String-Literalen R»raw string« und U»unicode string« kann der C++-Entwickler eigene Literale definieren.

EXKURS

Literale

Literale sind Zeichenfolgen, die zur Darstellung von Basistypen verwendet werden. Sie sind sogenannte Rvalues, besitzen keine Adresse und können nur auf der rechten Seite einer Zuweisung stehen. Die bekanntesten Beispiele aus klassischem C++:

Datentyp	Untertypen	Literale
Wahrheitswerte		true false
Zeichen	char wchar_t	'c' L'c'
Integer	Dezimal Oktal Hexadezimal	2 02 0x2
Fließkommazahlen	double float long double	0.123, 1.23e-1 6.7f, 6.7F 6.7l, 6.7L
Zeichenketten	char const* wchar_t const*	"Text" L"Text"

Tabelle 3.2: Einige klassische Literale

3.5.1 Konstante Ausdrücke

constexpr

In C++11 wird das Konzept von konstanten Ausdrücken (***constexpr***) erweitert. Variablen, Funktionen oder auch Objekte können, insofern sie strenge Bedingungen einhalten, als `constexpr` deklariert werden. Damit lassen sich Funktionen als konstante Ausdrücke verwenden und Objekte zur Übersetzungszeit evaluieren. In C++11 ist es erlaubt, Arrays über Funktionen zu initialisieren, die als `constexpr` deklariert wurden.

```
constexpr int getSize() {return 10;}
int someValue[getSize() + 7];
```

Der große Vorteil der konstanten Ausdrücke ist, dass sie der Compiler optimieren kann, denn sie werden schon zur Übersetzungszeit evaluiert.

Um die Optimierung geht es auch bei den Erweiterungen der Plain Old Data (POD).

3.5.2 Plain Old Data (POD)

Plain Old Data sind Datenstrukturen, die ein C-Standardlayout besitzen. Damit können sie direkt mit den effizienten C-Funktionen `memcpy`, `memmove` kopiert oder auch mit `memset` initialisiert werden. C++11 erweitert die Regeln, da nun Klassen und Strukturen als POD gelten, wenn sie drei Bedingungen erfüllen. Sie müssen trivial sein, ein Standardlayout besitzen und ihre nichtstatischen Datenelemente müssen auch PODs sein. Genauer lässt sich das im Kapitel Plain Old Data im Buchabschnitt Kernsprache nachlesen.

Ähnlich zu POD, erfahren Unions mit C++11 einige Erweiterungen.

3.5.3 Unbeschränkte Unions

Mit C++11 erfahren Unions eine Erweiterung. Sie können Elemente von Datentypen wie `std::string` mit nicht trivialen speziellen Elementfunktionen besitzen. Spezielle Elementfunktionen sind Funktionen, die der Compiler automatisch erzeugt.

Wird die Anwendung von Unions erweitert, so wird die von Enums typsicherer.

3.5.4 Streng typisierte Aufzählungstypen

Die klassische Aufzählungstypen `enum`s haben drei Probleme.

1. Sie konvertieren implizit zu int.
2. Sie führen ihre Bezeichner in dem umgebenden Bereich ein.
3. Der zugrunde liegende Typ kann nicht angegeben werden.

Mit diesen Problemen räumen die neuen, streng typisierten Aufzählungstypen auf, auf die nur über den Namen des Aufzählungstyps zugegriffen werden kann. Sie vereinen die Funktionalität der klassischen `enum`-Datenstruktur mit Aspekten von Klassen. Eine streng typisierte `enum`-Color vom zugrunde liegenden Datentyp `unsigned int` ist kompakt definiert.

```
Enum class Color: unsigned int {red, green, blue};
```

Optional kann statt `class` `struct` verwendet werden und der Datentyp `unsigned int` weggelassen werden, so dass `red`, `green` und `blue` vom Typ `int` sind.

Neben streng typisierten Aufzählungstypen gibt es auch neue String-Literale in C++11. Dies sind Raw-String-Literale und Unicode-Literale.

3.5.5 Neue String-Literale

Raw-String-Literale

Raw-String-Literale haben sich in Python als äußerst praktisch erwiesen, wenn es darum geht, den Inhalt eines Strings nicht zu interpretieren. Typische Anwendungsfälle für Raw Strings sind reguläre Ausdrücke oder auch Dateipfade unter Windows. Ein Raw String wird in C++11 durch `R"(raw string")` definiert.

Unicode-String-Literale

Der zweite neue Typ von String-Literalen sind die Unicode-String-Literale. C++11 unterstützt die drei Unicode-Kodierungen: UTF-8, UTF-16 und UTF-32. Für UTF-16 und UTF-32 wurde C++11 um zwei neue Zeichentypen `char16_6` und `char32_t` erweitert.

Benutzerdefinierte Literale

Darüber hinaus unterstützt C++11 benutzerdefinierte Suffix-Literale. Dies ist für natürliche Zahlen, Fließkommazahlen, Strings und Zeichen möglich. Damit lassen sich Literale wie 130.3km oder auch »978-3-16-148410-0«ISBN definieren. Interpretiert werden diese Literale durch die Literale-Operatoren, die die Anwendungslogik implementieren.

Von der C++11-Laufzeit wird das Literal 130.3km auf den Literale-Operator abgebildet.

```
Kilometer operator ""km(long double d){
  return Kilometer(d);
}
```

3.5.6 nullptr

Das neue Schlüsselwort `nullptr` definiert eine Nullzeigerkonstante in C++11. Damit räumt es mit der Mehrdeutigkeit der Zahl `0` in C++ und dem C-Makro `NULL` auf. Denn abhängig vom Kontext bezeichnet `0` den Nullzeiger `((void*)0)` oder die natürliche Zahl `0`. `NULL` hingegen lässt sich in der Regel nach `int` konvertieren. Der `nullptr` kann aber nur als Zeiger oder in einem booleschen Ausdruck verwendet werden.

Neben dem C++11-Literal `nullptr` gibt es noch weitere Verbesserungen in C++11, die die Sprache klarer machen. Auch der aktuelle C-Standard C99 Standard ist größtenteils in C++11 integriert.

3.6 Weitere Aufräumarbeiten und Integration von C99

3.6.1 Aufräumarbeiten

Parser-Probleme mit >>

C++98 hat Probleme, einen Ausdruck der Form `std::vector<std::vector<int>>` richtig zu parsen, denn >> wird vom ihm irrtümlich als ein Token und damit als Rechts-Shift-Operator interpretiert. Daher war es erforderlich, zwischen den zwei abschließenden >> ein Leerzeichen zu verwenden. Dies ist mit C++11 nicht mehr notwendig.

3.6.2 Integration von C99

Da der alte C++-Standard C++98 vor dem aktuell gültigen C99-Standard verabschiedet wurde, werden dessen Features in den neuen C++-Standard C++11 aufgenommen.

long long int

C++11 erbt den Datentyp `long long int` von C99, der zumindest 64 Bit (Listing 3.20, Zeile 14) groß ist.

__func__

Der Präprozessor kann den Namen `__func__` (Zeile 4 und 17) evaluieren.

c99.cpp

```
01 #include <iostream>
02
03 void showFuncName(){
04   std::cout << "__func__=  " << __func__ << std::endl;
05 }
06
07 int main(){
08
09   std::cout << std::endl;
10
11   long long int ll=10;
12   int i= 10;
13
14   std::cout << "sizeof(long long int)= " << sizeof(ll)
               << std::endl;
15   std::cout << "sizeof(int))= " << sizeof(i) << std::endl;
16
17   std::cout << "__func__=  " << __func__ << std::endl;
18   showFuncName();
19
20   std::cout << std::endl;
21
22 }
```

Listing 3.20: C99-Features in C++11

WEBSITE Tour de C++11: c99.cpp

Abbildung 3.12: C99-Features in der Anwendung zeigt die Ausgabe des Programms.

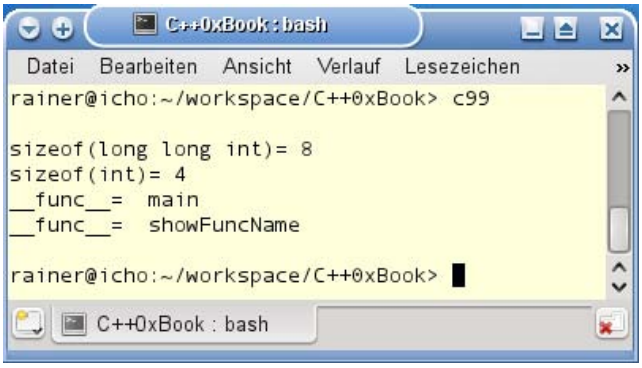


Abbildung 3.12: C99-Features in der Anwendung

Damit verlassen wir den Bereich der Kernsprache von C++11. Es folgt die neue Threading-Funktionalität von C++11, die ihre Erweiterungen insbesondere in den neuen Bibliotheken anbietet.

Copyright