Initialisierunglisten - Fortsetzung

Initialisierungslisten können kopiert werden, allerdings werden dabei nur die Positionen des Listenanfangs und -endes und ihre Länge kopiert. Damit kann auf die Listenelemente zugegriffen werden, obwohl sie nicht kopiert werden. Zusätzlich gibt es eine Komponentenfunktion `size()`, mit der die Länge der Initialisierungsliste abgefragt werden kann.

Bsp.: Konstruktor und Zuweisungsop. im selbstdef. Datentyp Vektor für Initialisierungslisten

```
                   :
#include <initializer_list>
                   :

class Vektor {
  private:
    double *ap;
    int    len;

  public:
                   :
    Vektor(initializer_list<double> il) {
      len = il.size();
      ap  = new double[len];
      auto p = ap;
      for (auto x : il)
        *p++ = x;
     }

    Vektor operator=(initializer_list<double> il) {
      delete[] ap;
      len = il.size();
      ap  = new double[len];
      auto p = ap;
      for (auto x : il)
        *p++ = x;
      return *this;
     }
                   :
};

int main()
{
  Vektor a = {1.0,3.0,5.0,7.0}, b;
  b = {1.0,2.0};
                   :
}
```

Verschiebeoperationen und rvalue-Referenzen

Bei Kopiervorgängen (Parameterübergabe, Rückgabe von Funktionswerten) und Zuweisungen werden die ursprünglichen Größen danach oft nicht mehr benötigt. Eine schnellere Übertragung kann zum Beispiel bei Vektoren, die intern mit einem Zeiger auf einen Speicherbereich arbeiten, dadurch erreicht werden, dass der Zeiger und die Länge des Vektors übergeben werden und im ursprünglichen Objekt beide auf 0 gesetzt werden. Derartige Operationen werden als Verschiebung bezeichnet und innerhalb der Klasse mit Hilfe eines Verschiebekonstruktor und eines Verschiebungszuweisungsoperator definiert.

Die Unterscheidung zwischen den verschiebenden und den kopierenden Konstruktoren bzw. Zuweisungsoperatoren wird durch die neu eingeführten rvalue-Referenzen ermöglicht. Diese ergänzen die bereits vorhandenen Referenzen, die als lvalue-Referenzen bezeichnet werden. Erstere werden bekanntlich mit `&` gekennzeichnet, letztere mit `&&`.

Jeder Ausdruck ist entweder ein lvalue oder ein rvalue, ursprünglich abhängig davon, ob er auf der linken Seite einer Zuweisung vorkommen kann oder nicht. In C++11/14 werden mit lvalue allgemeiner Größen bezeichnet, auf die mit Namen oder Zeiger zugegriffen werden kann *und* die *nicht* verschiebbar sind. Mit der etwas missverständlich bezeichneten Funktion `move` kann der Programmierer ein lvalue-Objekt als rvalue-Referenz kennzeichnen, so dass Verschiebeoperationen angewandt werden können.

Bsp.: swap

```
template <class T> inline void swap(T& a, T& b)
{
   T tmp = move(a);
   a = move(b);
   b = move(tmp);
}
```

Bsp.: Verschiebekonstruktor und -zuweisungsoperator für selbstdef. Datentyp Vektor

```
class Vektor{
           :
  public:
           :
    Vektor(Vektor&& a) {                            // Verschiebekonstruktor
      len = a.len; ap = a.ap;
      a.len = 0; a.ap = 0;
    }

    Vektor operator=(Vektor&& b) {                  // Verschiebezuweisungsoperator
      if (this!=&b) {
        delete[] ap;
        len = b.len; ap = b.ap;
        b.len = 0; b.ap = 0;
      }
      return *this;
    }
};

int main()
{
  Vektor a = {1.0,3.0,5.0,7.0}, b = {1.0,2.0};
  swap(a,b);  // Benutzt Verschiebeoperationen (wg. move)
           :
```

Bemerkungen:

1. Das Verhalten von lvalue- und rvalue-Referenzen ist nicht ganz symmetrisch: Da rvalue-Referenzen zur Implementierung der move-Semantik verwendet werden, erscheint es nicht sinnvoll, konstante rvalue-Referenzen zu betrachten. rvalue-Refererenzen können auch nur an rvalues und *nicht* konstante lvalue-Referenzen nur an lvalues gebunden werden. Allerdings können konstante lvalue-Referenzen sowohl an lvalues als auch rvalues gebunden werden. Das ermöglicht im Falle des Nichtvorhandenseins der Verschiebekonstruktoren bzw. -zuweisungsoperatoren die automatischen Verwendung der Kopierkonstruktoren und des (Kopier)zuweisungsoperators. Aus diesem Grund ist beispielsweise in C++11/14 nur noch die oben erwähnte Implementierung von `swap` vorhanden.

2. Die Argumente von return-Anweisungen in Funktionen können, sofern es sich um lokale lvalues handelt, zum Zweck der Optimierung vom Compiler als rvalues behandelt werden, d.h. für die Rückgabe von Funktionswerten wird der Verschiebekonstruktor anstelle des Kopierkonstruktors benutzt. Weitergehende Optimierungen sind in bestimmten Fällen möglich (copy/move elision).

3. Abweichende Regeln für rvalue-Referenzparameter in Funktions*templates* bewirken, dass es dort sogar möglich ist, lvalues als Argumente einzusetzen („universelle" Referenzen). Da die Typableitung bei `auto` den Regeln der Parametertypableitung bei Funktionstemplates folgt, kann `auto&&` eine rvalue-Referenz oder eine lvalue-Referenz bezeichnen.

Literatur

B. Stroustrup: *Programming: Principles and Practice Using C++. Second Edition.* Addison-Wesley 2014. [C++11]

B. Stroustrup: *The C++ Programming Language - Fourth Edition* Addison-Wesley 2013. [C++11]

B. Stroustrup: *Einführung in die Programmierung mit C++.* Pearson Studium 2010. [C++98]

B. Stroustrup: *Die C++ Programmiersprache.* Addison-Wesley 2000. [C++98]

M. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley 1995. [C++90]

U. Breymann: *Der C++ Programmierer.* Hanser 2009. [C++98]

R. Grimm: *C++11.* Addison-Wesley 2012. [C++11]

N. Josuttis: *The C++ Standard Library - A Tutorial and Reference.* Addison-Wesley 1999. [C++98]

D. Musser, G. Derge, A. Saini: *STL Tutorial and Reference Guide.* Addison-Wesley 2001. [C++98]

A. Langer, K. Kreft: *Standard C++ IOStreams and Locales.* Addison-Wesley 2000. [C++98]

D. Vandevoorde, N. Josuttis: *C++ Templates - The Complete Guide.* Addison-Wesley 2003. [C++98]

J. Barton, L. Nackman: *Scientific and Engineering C++.* Addison-Wesley 1994. [C++90]

R. Barrett et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM 1994.

W. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley 1992.

S. Meyers: *Effective C++.* Addison-Wesley 1998. [C++98]

S. Meyers: *More Effective C++.* Addison-Wesley 1997. [C++98]

S. Meyers: *Effective STL.* Addison-Wesley 2001. [C++98]

International Standard ISO/IEC 14882 Programming languages – C++. ANSI 1988. [C++98]

Online-Ressourcen

`http://www.cplusplus.com`

`http://en.cppreference.com`