10.3 Templates und die STL/Containertyp vector

10.3.1 Nachteile herkömmlicher Arrays

Das aus C und anderen Programmiersprachen bekannte Array hat einige Nachteile:

- Seine Größe muss vor dem Compilieren festgelegt werden
- Seine Größe kann während des Programmlaufs nicht geändert werden
- Es sind *buffer overflows* möglich, wenn man beim Programmieren nicht aufpasst und auf Elemente zugreift, die es nicht gibt, weil sie außerhalb der Array-Grenzen liegen

Aber es gibt Lösungen in C:

- Das VLA (*variable length array*) erlaubt es, dass man die Arraygröße zur Laufzeit festlegt
- Ein dynamisch mit `malloc()` angelegtes Array erlaubt die Festlegung und Änderung (mit `realloc()`) der Größe zur Laufzeit des Programms
- Die Bibliothek *libefence* erlaubt das sofortige Erkennen eines *buffer overflows* bei einem dynamisch angelegten Array

10.3.2 Lösung in C++

In C++ gibt es eine weitere Lösung: Die Container-Klassen der *standard template library*, kurz STL-Container-Klassen genannt. Sie stellen auf Wunsch Behälter zur Verfügung, in die man andere Daten hineinpacken kann, z. B. Strings.

Bei solch einem Behälter gibt man vorher an, was der Inhaltstyp sein soll. Und wegen der Benutzung von Templates ist trotzdem sichergestellt, dass man auf den richtigen Datentyp zugreift[1].

Es gibt verschiedene Behälterformen: Array-artige Vektoren, Listen, Maps usw. Und sie sind bereits auf maximale Leistung optimiert.

10.3.3 Containertyp vector

Der einfachste Containertyp hat den Namen `vector`. Er ähnelt dem klassischen Array. Hier ein Beispiel:

vector1.cpp

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(void)
6  {
7      vector <double> messwert(10, 47.11);
8
9      for(int i=0; i<messwert.size(); ++i)
10     {
11         cout << messwert[i] << " ";
12     }
13     cout << endl;
14     return 0;
15 }
```

[1] Es gibt auch gleichartige C-Lösungen, bei denen muss man aber immer beachten, dass man genau *den* Datentyp herausliest, den man vorher hineingeschrieben hat; oder aber man benutzt Makros wie bei GTK+. Und im Gegensatz zu den C-Lösungen liegen die C++-Lösungen netterweise bereits in der C++-Standardbibliothek STL.

- Zeile 2: Für jeden Containertyp gibt es eine eigene Header-Datei, die so heißt wie der Container selbst, hier eben `<vector>`.
- Zeile 7: In den spitzen Klammern steht der Inhalts-Typ. Wir bekommen also einen vector von double-Elementen.
- Zeile 7: Optionale Parameter des Konstruktors: Größe 10 (default: 0), Inhalt (für alle Elemente!) 47.11;
- Zeile 8: Die `size()`-Methode gibt die Anzahl der Elemente an (man braucht sich die 10 nicht zu merken)
- Zeile 10: `messwert[i]`: Der Zugriff verhält sich wie bei einem Array. Die eckige Klammer ist aber eine Methode der Klasse; der Operator wurde überladen.

Mit diesem Container kann man schon einmal die Größe zur Laufzeit festlegen.

10.3.4 Größe ändern

Nun soll die Größe zur Laufzeit geändert werden:

vector2.cpp

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(void)
6  {
7      vector <double> messwert(10, 47.11);
8      messwert.resize(12, 20.26);
9      for(int i=0; i<messwert.size(); ++i)
10     {
11         cout << messwert[i] << " ";
12     }
13     cout << endl;
14     messwert.resize(8);
15     cout << "size....: " << messwert.size() << endl;
16     cout << "capacity: " << messwert.capacity() << endl;
17     return 0;
18 }
```

- Zeile 8: `messwert.resize(12)` ändert die Größe auf 12 Elemente. Der optionale zweite Parameter initialisiert alle neuen Elemente.
- Zeile 16: Hier wird eine Reserve-Größe angegeben, bis zu der die Elemente nicht umkopiert werden müssen, wenn man das Array vergrößert.
- Es gibt auch eine anderen Möglichkeit zum Vergrößern:
  `messwert.push_back(99.9)` fügt ein Element am Ende ein und vergrößert damit das Array
- Ebenso eine Möglichkeit zum Verkleinern:
  `messwert.pop_back()` löscht das letzte Element und verkleinert das Array
- Wenn alles gelöscht wurde:
  `messwert.empty()` ist dann `true`

Die Größe eines vector-Containers kann man also zur Laufzeit ändern.

10.3.5 Zugriffskontrolle

Bei Arrays wünscht man sich manchmal eine Möglichkeit, einen Zugriff außerhalb der Grenzen zu verhindern. Auch das ist beim vector möglich:

vector3.cpp

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(void)
6  {
7      vector <double> messwert(10, 47.11);
8      cout << "Zugriff mit mw[x]...:" << messwert[40] << endl;
9      cout << "Zugriff mit mw.at(x):" << messwert.at(40) << endl;
10     return 0;
11 }
```

- Zeile 8: Der Zugriff ist möglich, verursacht aber Unsinn.
- Zeile 9: Der Zugriff bewirkt eine sogenannte *Ausnahmebehandlung*, in diesem Fall einen Programmabbruch. Man kann die Ausnahme aber abfangen und das Programm sinnvoll weiterführen.

Mit dem Containertyp vector ist also eine Zugriffskontrolle möglich. Sie kostet allerdings etwas Rechenzeit.

10.3.6 Vergleich zu anderen Lösungen

Bei Lösungen in der Informatik zählen häufig Speicherverbrauch, Zeitverbrauch und Zuverlässigkeit.

Der Containertyp vector bietet, wie ein richtiges Array, einen extrem schnellen Zugriff (intern per Adresse). Die Größe eines vectors zu ändern kostet etwas Zeit (wie bei `malloc()`. Richtig lange dauert es, wenn man Elemente mitten im vector einsortieren möchte[2]. Für diesen Zweck sind andere Containertypen besser geeignet.

[2]Dafür braucht man einen so genannten *Iterator*, dazu später