

## Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>1. Dynamische Datenstrukturen in C++</b>                  | <b>1</b>  |
| <b>1.1. Überblick: Die Standard Template Library (STL)</b>   | <b>1</b>  |
| 1.1.1. FTP-Sites / URLs                                      | 1         |
| 1.1.2. STL Programme übersetzen                              | 2         |
| 1.1.3. Bestandteile der STL                                  | 2         |
| 1.1.4. Containerklassen (Containers)                         | 3         |
| 1.1.5. Iteratoren (Iterators)                                | 3         |
| 1.1.6. Algorithmen (Algorithm)                               | 4         |
| <b>1.2. Containerklassen (Container)</b>                     | <b>5</b>  |
| 1.2.1. Die Klasse vector                                     | 6         |
| 1.2.2. Beispiel: t_vector.cpp                                | 7         |
| 1.2.3. Die Klasse list                                       | 9         |
| 1.2.4. Beispiel: t_list.cpp                                  | 10        |
| 1.2.5. Die Klasse map  | 12        |
| 1.2.6. Beispiel: t_map.cpp                                   | 12        |
| 1.2.7. Die Klasse stack                                      | 13        |
| 1.2.8. Beispiel: t_stack.cpp                                 | 13        |
| <b>1.3. Iteratoren</b>                                       | <b>14</b> |
| 1.3.1. Beispiel: t_iterator.cpp                              | 14        |
| 1.3.2. Beispiel: crossreference (map, list, iteratoren)      | 16        |
| 1.3.3. Beispiel: crossreference.cpp (m,u) crossreference.txt | 16        |
| <b>1.4. Algorithmen</b>                                      | <b>18</b> |
| 1.4.1. Beispiel: t_algorithmen.cpp                           | 19        |
| <b>1.5. Beispiel: Email Adressbuch – Die Klasse Map</b>      | <b>21</b> |
| 1.5.1. Beispiel: t_adb_map.cpp                               | 21        |

## 1. Dynamische Datenstrukturen in C++

### Inhalt:

Aufbau und Funktionalität der STL kennen lernen.

### Ziel:

STL in eigenen Programmen/Problemlösungen einsetzen können.

### Voraussetzungen:

C, C++ Kenntnisse

### 1.1. Überblick: Die Standard Template Library (STL)

STL ist die Abkürzung für Standard Template Library. Sie ist eine komplexe Sammlung von **Templateklassen** und **Templatefunktionen**, welche viele bekannte und häufig gebrauchte **Algorithmen** und **Datenstrukturen** kapseln. Man findet in ihr zum Beispiel Vektoren (das sind dynamische Arrays), Listen, Stacks, sowie Such- und Sortieralgorithmen. Die Standard Template Library ist seit 1994 ein Teil von C++.

#### 1.1.1. FTP-Sites / URLs

Die Hewlett Packard STL von Alexander Stepanov and Meng Lee:

- <ftp://butler.hpl.hp.com/pub/stl/stl.zip> für Borland C++ 4.x
- <ftp://butler.hpl.hp.com/pub/stl/sharfile.Z> für GCC

David Mussers STL-page:

- <http://www.cs.rpi.edu/~musser/stl.html>

Mumit's STL Newbie guide:

- <http://www.xraylith.wisc.edu/~khan/software/stl/STL.newbie.html>

Joseph Y. Laurino's STL page:

- [http://weber.u.washington.edu/~bytewave/bytewave\\_stl.html](http://weber.u.washington.edu/~bytewave/bytewave_stl.html)

Eine gute, kurze Zusammenfassung findet man unter Tutorials bei:  
<http://www.codeproject.com/KB/stl/stlintroduction.aspx> (kurz und gut)

### 1.1.2. STL Programme übersetzen

#### t\_compile\_stl.cpp

```
// t_compile_stl.cpp
// STL include files - include STL files first!
// g++ t_compile_stl.cpp -o t_compile_stl.exe

#include <vector>

#include <iostream>
using namespace std;

int main () {
    vector<int> v(5);

    v[0] = 4;
    cout << "First vector element: " << v[0]<<endl;

    return 0;
}
```

### 1.1.3. Bestandteile der STL

Die STL ist in drei Bereiche organisiert:

1. Containerklassen (**Containers**)
2. Iteratoren (**Iterators**)
3. Algorithmen (**Algorithms**)

Diese drei Bereiche agieren allerdings nicht alleine sondern sind mit einander verbunden.  
Einfach ausgedrückt könnte man sagen

**Algorithmen werden auf Objekte, die in Containerklassen gespeichert sind, angewendet.**

**Für den Zugriff auf die Objekte werden die Iteratoren verwendet.**

### 1.1.4. Containerklassen (Containers)

Containerklassen sind **Objekte**, die andere **Objekte** enthalten.

Containerklassen stellen den Kern der STL dar. Es gibt verschiedene Typen von Containerklassen in der STL. In der nachfolgenden Tabelle sind diese Typen aufgelistet.

| Container klassen     | Beschreibung  | Header                |
|-----------------------|---|-----------------------|
| <b>deque</b>          | Eine doppelt verkettete Liste   | <b>&lt;deque&gt;</b>  |
| <b>list</b>           | Eine lineare Liste  | <b>&lt;list&gt;</b>   |
| <b>map</b>            | Speichert Schlüssel/Wert-Paare, wobei ein Schlüssel immer mit nur einem Wert verknüpft ist.             | <b>&lt;map&gt;</b>    |
| multimap              | Speichert Schlüssel/Wert-Paare, wobei ein Schlüssel mit einem oder mehreren Werten verknüpft sein kann. | <map>                 |
| multiset              | Eine Ansammlung von Daten, in welcher ein Element mehrfach vorkommen kann.                              | <set>                 |
| <b>priority_queue</b> | Eine priorisierte Reihe   | <b>&lt;queue&gt;</b>  |
| <b>queue</b>          | Ein FIFO Speicher   | <b>&lt;queue&gt;</b>  |
| <b>set</b>            | Eine Ansammlung von Daten, in welcher ein Element nur einmal vorkommt.                                  | <b>&lt;set&gt;</b>    |
| <b>stack</b>          | Ein Stapel  | <b>&lt;stack&gt;</b>  |
| <b>vector</b>         | Ein dynamisches Array   | <b>&lt;vector&gt;</b> |

Wir unterscheiden noch zwischen:

|                                      |          |
|--------------------------------------|----------|
| <b>Sequentielle Containerklassen</b> | Vector   |
|                                      | Deque    |
|                                      | List     |
| <b>Assoziative Containerklassen</b>  | Set      |
|                                      | Multiset |
|                                      | Map      |
|                                      | Multimap |

### 1.1.5. Iteratoren (Iterators)

Iteratoren sind Objekte, die sich wie **Zeiger** verhalten.

Wie man sich mit einer Zeigervariablen über alle Elemente eines C-Arrays bewegen kann, so kann man mit Iteratoren sich über die Objekte einer Containerklasse bewegen.

Iteratoren sind Containerunabhängig. Sie werden benötigt, um die Elemente eines Containers durchzugehen. Mit ihnen kommt man an die Werte des Containers. Ein Iterator wird bei jedem Container gleich erzeugt.

Mit den Operator **++** kommen wir zum nächsten Element eines Containers. Um dann den Iterator durch einen Container wandern zu lassen müssen wir ihn als Startwert den Beginn des Containers geben. Dafür gibt es die Methode "**begin()**". Diese zeigt auf das erste Element.

Um zu erfahren wann den z.B. die Liste zu Ende ist, gibt es die Methode "**end()**". Sie zeigt **hinter** das letzte Element.

Ein Iterator kann man auf Gleichheit (**==**) oder Ungleichheit (**!=**) prüfen.

Iteratoren lassen sich dereferenzieren, d.h. mit dem **\***-Operator können wir auf den Wert zugreifen.

Es gibt kein größer oder kleiner als, darum muss die Laufbedingung einer Schleife **!= ContainerObjekt.end()** sein. Hier ein Beispiel mit einer Liste.

```
//Einen Container erzeugen, der Zahlen speichert
list<int> container;

//einen Iterator für unseren Container erzeugen
list<int>::iterator it;

//Alle werte im Container ausgeben
for(it = container.begin(); it != container.end(); ++it) {
    cout << *it << std::endl;
}
```

Wollen wir konstante Iteratoren, sprich wir wollen den Wert nicht verändern dürfen, so müssen wir einen `const_iterator` anlegen.

```
list<int>::const_iterator it;

for(it=container.begin(); it!= container.end(); it++) {
    cout << *it << std::endl;
    (*it)++;    //hier meldet der Compiler einen Fehler da Konstanten
                //nicht geändert werden dürfen
}
```

### 1.1.6. Algorithmen (Algorithm)

Algorithmen werden verwendet, um Container zu initialisieren, zu sortieren, zu durchsuchen und zu ändern.

Algorithmen sind Funktionen, die auf die Objekte in Containerklassen angewandt werden.

Hier ein Beispiel, um eine Liste rückwärts zu durchlaufen:  
Es mag vielleicht leicht klingen, aber es gibt etwas zu beachten. Da "end()" hinter das letzte Element zeigt können wir es nicht als Startwert für unseren Iterator nehmen. Des Rätsels Lösung sind die Methoden "**rbegin()**" und "**rend()**". Wir müssen aber einen neuen Iterator erzeugen der Rückwärts unterstützt.

```
//einen Rückwärtsiterator
list<int>::reverse_iterator it;
```

```
//Wichtig! rbegin() und rend()
for(it = container.rbegin(); it != container.rend(); it++) {
    cout << *it << std::endl;
}
```

## 1.2. Containerklassen (Container)

Die STL beinhaltet verschiedene Arten von Containerklassen. Einige davon sind folgende:

| Sequentielle Containerklassen |  |
|-------------------------------|--|
| vector                        | Ein dynamisches Array kann zur Laufzeit seine Größe ändern. Wie ein normales Array können die Datensätze über die eckigen Klammern angesprochen werden. Der vector hat eine Besonderheit. Die Datensätze liegen im Speicher direkt hintereinander. Zum Beispiel können wir ganz leicht alle Datensätze in eine Binärdatei schreiben ohne sie einzeln durchgehen zu müssen.<br>Als erstes müssen wir die Headerdatei <vector> einfügen.   |
| dqueue                        | Ein Vector hat einen Nachteil, er kann nur von hinten erweitert werden. Es gibt keine Möglichkeit an den Anfang einen Wert hinzuzufügen. Möchten Sie einen Vector der genau diese Eigenschaft besitzt, dann müssen sie den Datencontainer deque nehmen. Es ist ein dynamisches Array, das nach beiden Seiten wächst. Dieser bietet uns zwei zusätzliche Methoden. Einmal push_front() und pop_front(). Wie bei dem Vektor können wir mit dem []-Operator auf die Elemente zugreifen.   |
| list                          | Eine list ist ein Datencontainer, der im Hintergrund eine doppelt verkettete Liste enthält. Um Instanzen dieser Containerklasse erzeugen zu können, müssen wir die Headerdatei <list> einfügen.  |
|                               |  |
| Assoziative Containerklassen  |  |
| map                           | Eine map ist eine Tabelle die Zeilen mit einem Schlüssel und einem dazugehörigen Wert enthält. Der Schlüssel darf nur einmal vorhanden sein. Wenn ein neuer Wert mit dem selben Schlüssel hinzugefügt wird, dann wird der alte mit dem neuen überschrieben. Das besondere an diesem Container ist die Implementierung als Binärbaum. Beim hinzufügen eines Schlüssel-Wert-Paares wird dieses Paar in den Baum sortiert eingefügt. Jeder der schon mal eine Binärbaum genutzt hat weiß, dass diese ziemlich schnell sind. Um die Containerklasse nutzen zu können müssen wir erst einmal die Headerdatei <map> einbinden. |
| multimap                      | In einer multimap dürfen Schlüssel auch mehrmals vorkommen. Dadurch geht aber die Eindeutigkeit und somit einige bestimmte Funktionen verloren. Es gibt keinen []-Operator und keine Methode find() mehr. Um die multimap nutzen zu können muss die Headerdatei <map> inkludiert sein.   |
| set                           | Ein set ist ein Container, der seine Element sortiert enthält. Auch ist es nicht möglich, dass ein Element mehrmals vorkommt. Der Wert ist zugleich der Schlüssel. Implementiert ist diese Klasse als Binärbaum. Dies  |

|                          |   |
|--------------------------|---|
|                          | hat den Vorteil, dass das Einfügen eines neuen Elements ziemlich schnell geht. Erstmal müssen wir die Headerdatei <set> einbinden um dieses Container nutzen zu können.   |
| multiset                 | Der Unterschied zum normalen set ist, dass die Eindeutigkeit der Schlüssel aufgehoben ist, d.h. wir können gleiche Schlüssel hinzufügen. Die Definition dieses Containers steht in der Headerdatei <set>. Diese müssen wir einbinden.   |
|                          |   |
| <b>Container Adapter</b> | <b>Containeradapter sind Containerklassen, die mit einem anderem Container implementiert wurden. Dazu zählen stack und queue. Beide werden standardmässig mit deque implementiert.</b>  |
| stack                    | Ein stack ist ein Stapel. Auf diesen Stapel kann man oben etwas drauflegen oder entfernen. Wie bei einem Papierstapel kann man immer nur den obersten lesen. Möchte man die darunter lesen, muss man erst die darüberliegenden Blätter runternehmen und auf einen neuen Stapel legen oder in den Schredder werfen.<br>Erstmal müssen wir die Headerdatei <stack> includieren um diesen Containeradapter nutzen zu können. |
| queue                    | Eine queue ist ein erweiterter Stapel. Bei diesem kann man auf das oberste und unterste Element schauen.<br>Zu erst includieren wir die Headerdatei <queue>.  |

### 1.2.1. Die Klasse vector

Die Klasse vector ist ein dynamisches Array, d.h. die Größe des Arrays kann sich während der Laufzeit ändern. Um die STL-Klasse vector verwenden zu können muß folgender Code eingefügt werden.

```
#include <vector>
using namespace std;
```

Nach diesen zwei Zeilen steht die vector-Klasse zur Verfügung. Da die vector-Klasse eine Template-Klasse ist, wird bei der Deklaration angegeben welche Art von Objekten das dynamische Array aufnimmt. Hier einige Beispiele:

```
vector<int> iv;           // Erzeugt einen Int-Vector mit der
                          // Anfangslänge 0

vector<char> cv(20);      // Erzeugt einen Char-Vector mit der
                          // Anfangslänge 20

vector<long> lv(10, 7);   // Erzeugt einen Long-Vector der Länge 10, und
                          // initialisiert alle 10 Elemente mit 7.

vector<TADB> adbv;        // Erzeugt einen Vector, der Elemente v. Typ
                          // TADB aufnehmen kann

vector<CGegenstand> vGegenstde; // Erzeugt einen Vector, der Objekte der
                          // Klasse CGegenstand aufnehmen kann
```

Die vector-Klasse stellt alle Methoden zur Verfügung, die man braucht um auf Elemente

zuzugreifen und sie zu verändern.

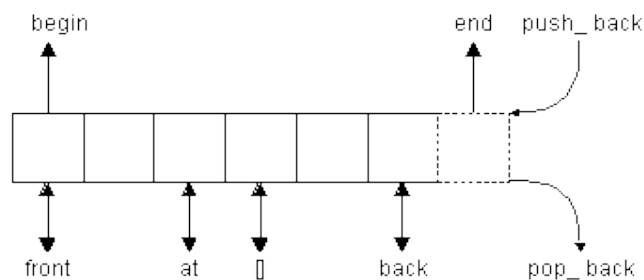
Zum einen ist der **Indexoperator []** in der vector-Klasse implementiert. Dadurch ist es möglich bei den Zugriffen auf das vector-Objekt die Standardnotation für Arrayzugriffe zu verwenden. Dies könnte z.B. so aussehen:

```
cv[10] = 'b';
if (cv[0] == 'x'){
    cv[19] = cv[10];
}
```

Neben dem Indexoperator werden noch folgende Methoden häufig verwendet:

|                          |  |
|--------------------------|--|
| <b>size()</b>            | Liefert die aktuelle Größe des vector-Objektes zurück.                           |
| <b>begin()</b>           | Liefert einen Iterator auf den Anfang des vector-Objektes zurück                 |
| <b>end()</b>             | Liefert einen Iterator auf das Ende des vector-Objektes zurück                   |
| <b>push_back(aValue)</b> | Fügt ein Element an das Ende an  |
| <b>pop_back()</b>        | Löscht das letzte Element  |
| <b>insert(aValue)</b>    | Fügt ein oder mehrere Elemente an eine beliebige Stelle in das vector-Objekt ein |
| <b>erase()</b>           | löscht Elemente aus dem vector-Objekt.   |
| <b>front()</b>           | Liefert eine Referenz auf das erste Element des Vektors zurück                   |
| <b>back()</b>            | Liefert eine Referenz auf das letzte Element des Vektors zurück                  |
| <b>at()</b>              | Liefert eine Referenz auf ein bestimmtes Element des Vektors zurück.             |

Im folgenden Bild werden die wichtigsten Methoden noch einmal graphisch veranschaulicht.



Hier folgt nun das vector-Beispielprogramm:

### 1.2.2. Beispiel: t\_vector.cpp

```
// t_vector.cpp
// g++ t_vector.cpp -o t_vector.exe

#include <vector>

#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[]){
// INT Vektor mit 10 Elementen
vector<int> v(10);
unsigned int i;

cout << "\nSTL-Demo: Vector\n";
// Die Anfangsgröße anzeigen
cout << "Size = " << v.size() << endl;
cout << "Speicherplatz = " << v.capacity() << endl;

// Zeige den aktuellen Inhalt des Vektors an
for (i = 0; i < v.size(); i++)
    cout << v[i] << " ";

cout << endl << endl;

// Den Elementen des Vektors einige Werte zuweisen
for (i = 0; i < 10; i++)
    v[i] = i*i;

// Die Size nach dem Füllen noch einmal anzeigen
cout << "Size = " << v.size() << endl;
cout << "Speicherplatz = " << v.capacity() << endl;

// Zeige den aktuellen Inhalt des Vektors an
for (i = 0; i < v.size(); i++)
    cout << v[i] << " ";

cout << endl << endl;

cout << "Den Vektor nun erweitern mittels push_back" << endl;
for (i = 0; i < 5; i++)
    v.push_back((10 + i)*(10 + i));

// Den aktuellen Inhalt anzeigen
for (i = 0; i < v.size(); i++)
    cout << v[i] << " ";
cout << endl << endl;

// Die Size nach dem Füllen noch einmal anzeigen
cout << "Size = " << v.size() << endl;
cout << "Speicherplatz = " << v.capacity() << endl;

cout << "Den Vektor nun erweitern mittels insert" << endl;
for (i = 0; i < 5; i++)
    v.insert(v.begin(), - static_cast<int>(i+1));

// Den aktuellen Inhalt anzeigen
for (i = 0; i < v.size(); i++)
    cout << v[i] << " ";
cout << endl << endl;

// Die Size nach dem Füllen noch einmal anzeigen
cout << "Size = " << v.size() << endl;
cout << "Speicherplatz = " << v.capacity() << endl;
```



```
cout << "Elemente aus dem Vektor loeschen mittels erase" << endl;
for (i = 0; i < 3; i++)
    v.erase(v.begin());

// Den aktuellen Inhalt anzeigen
for (i = 0; i < v.size(); i++)
    cout << v[i] << " ";
cout << endl << endl;

// Die Size nach dem Füllen noch einmal anzeigen
cout << "Size = " << v.size() << endl;
cout << "Speicherplatz = " << v.capacity() << endl;

return 0;
}
```

### 1.2.3. Die Klasse list

Zuerst legen wir eine list an.

```
#include <list>
using namespace std;

list<int> zahlen;
```

Für das Hinzufügen von Datensätzen gibt es zwei Möglichkeiten. Wir können unsere Datensätze an den Anfang oder ans Ende setzen. Um einen Datensatz an den Anfang der Liste einzufügen, nehmen wir die Methode "**push\_front()**". Soll der Datensatz aber an das Ende eingefügt werden, so nehmen wir die Methode "**push\_back()**". In diesem Beispiel fügen wir 3 Zahlen an das Ende und 3 Zahlen an den Anfang.

```
zahlen.push_back(911);
zahlen.push_back(4711);
zahlen.push_back(12345);
zahlen.push_front(-2000);
zahlen.push_front(2005);
zahlen.push_front(306);
```

Genau so, wie wir die Daten an den Anfang und ans Ende hinzugefügt haben, können wir sie auch wieder löschen. Dafür gibt es die Methoden **pop\_front()** und **pop\_back()**;

```
zahlen.pop_front();    //löscht das erste Element in der Liste
zahlen.pop_back();     //löscht das letzte Element in der Liste
```

Die Anzahl der Einträge der Liste bekommen wir mit der Methode **size()** heraus.

```
cout << "Anzahl der Einträge: " << zahlen.size() << std::endl;
```

Jetzt wollen wir aber auch wissen, was in der Liste drin steht. Um das erste und das letzte Element anzuzeigen gibt es die Methode "**front()**" bzw. "**back()**". Diese geben jeweils eine Referenz zurück.

```
cout << "Das erste Element: " << zahlen.front() << std::endl;
cout << "Das letzte Element: " << zahlen.back() << std::endl;
```

Um nun alle Elemente anzuzeigen, müssen wir einen Iterator für unsere Liste erzeugen.

```
//Iterator erzeugen
list<int>::iterator it;

//Die Liste durchgehen
for(it=zahlen.begin(); it!= zahlen.end(); it++) {
    cout << *it << endl;
}

//Es ist auch möglich den Wert des Iterators zu verändern
for(it=zahlen.begin(); it!= zahlen.end(); it++)
    (*it) += 100;
```

Beim Löschen eines Elements muss man etwas beachten. Wenn wir ein Element löschen, dann zeigt unser Iterator noch auf das gelöschte.

Zum Glück gibt es eine Methode zum Löschen, die uns dann einen neuen gültigen Iterator zurück gibt. Zum Löschen - oder besser gesagt - zum Entfernen eines Elements aus einem Container, wird uns die Methode "**erase()**" zur Verfügung gestellt.

In dem folgendem Beispiel werden alle Zahlen gelöscht die kleiner als 1000 sind. Um zu demonstrieren, wie der Iterator reagiert, wird der Wert vor dem Löschen und nach dem Löschen ausgegeben.

```
for(it = zahlen.begin(); it!= zahlen.end(); it++)
{
    if(*it < 1000)
    {
        cout << "Vor dem loeschen: " << *it << std::endl;
        it = zahlen.erase(it);
        cout << "Nach dem loeschen: " << *it << std::endl;
    }
}
```

### Beispiel mit einer Klasse

Jetzt folgt ein Beispiel, das zeigt, wie man eine Liste mit Objekten füllt und sie ausgibt.

Hierzu verwenden wir eine kleine Klasse, die den Namen und das Geburtsjahr speichert. Mit dem Konstruktor oder der set()-Methode können die Werte geändert werden und mit den beiden get-Methoden bekommen wir die Werte zurück.

#### 1.2.4. Beispiel: t\_list.cpp

```
// t_list.cpp
// ahofmann 2011
// stl-demo: list<Person>
```

```
// g++ t_list.cpp -o t_list.exe
// ./t_list.exe

#include <cstring>

#include <list>
#include <iostream>
using namespace std;

class Person {
public:
    Person(const char* n = NULL, int j = 1950) : gebjahr(j){
        if(n) strcpy(name, n);
    }

    void set(const char* n, int j) {
        gebjahr = j;
        strcpy(name, n);
    }

    int getGebJahr(){return gebjahr;};
    const char* getName(){return name;};

private:
    char name[128];
    int gebjahr;
};

int main() {
    //Eine Liste von Personen erzeugen
    list<Person> personen;

    //Das hinzufügen von Objekten über den Konstruktoraufruf
    personen.push_back(Person("Heinz Klein", 1986));
    personen.push_back(Person("Max Mustermann", 1900));

    //Wir können auch eine Instanz der Klasse erzeugen und diese dann
    //hinzufügen. Dabei werden die Werte aber nur kopiert und in der
    //Liste entsteht eine neue Instanz.

    Person person1("Hans Hammer", 1960);
    personen.push_back(person1);

    Person person2;
    person2.set("Bart Simpson", 1988);
    personen.push_back(person2);

    list<Person>::iterator it;
    for(it = personen.begin(); it != personen.end(); ++it) {
        cout << "Name: " << (*it).getName(); // oder: it->getName()
        cout << " Geburtsjahr: " << (*it).getGebJahr();
        cout << endl;
    }
    return 0;
}
```

output:

Name: Heinz Klein Geburtsjahr:1986  
Name: Max Mustermann Geburtsjahr:1900  
Name: Hans Hammer Geburtsjahr:1960  
Name: Bart Simpson Geburtsjahr:1988

### 1.2.5. Die Klasse map

### 1.2.6. Beispiel: t\_map.cpp

```
// t_map.cpp
// a.hofmann
// g++ t_map.cpp -o t_map.exe

#include <map>
#include <iostream>
using namespace std;

int main(){
    map <string, double, less<string> > m_varlist;
    map <string, double, less<string> >::iterator i;

    string aVar;
    double aDbl;

    cout << "\n*** Map-Demo: map <string, double, less<string>>    m_varlist;\n";
    cout << "Eingabe: string (ende mit quit)"<<endl;

    cin >>aVar;
    cout << "Eingabe: double (ende mit 0)"<<endl;
    cin >>aDbl;

    while (aVar != "quit"){
        m_varlist.insert (make_pair(aVar,aDbl));

        cout << "Eingabe: string (ende mit quit)"<<endl;
        cin >>aVar;
        cout << "Eingabe: double (ende mit 0)"<<endl;
        cin >>aDbl;
    }

    //auch das geht
    m_varlist["hofmann"]= 99.0;

    cout <<"*** Liste ausgeben\n";
    for(i= m_varlist.begin(); i != m_varlist.end(); i++)
        cout << i->first << "\t\t\t:--->"<< i->second << endl;

    cout <<"*** Suche nach einem bestimmten String\n";
    cin>>aVar;
```

```
i= m_varlist.find(aVar);
if (i == m_varlist.end())
    cout << "nicht gefunden"<<endl;
else{
    cout <<"String="<< i->first<<endl;
    cout <<"Double="<< i->second<<endl;
    //cout <<"Double="<< m_varlist[aVar]<<endl;
}
return 0;
}
```

output (Beispiel)

```
*** Liste ausgeben
hofa      :<-->123
hofmann   :<-->99
max       :<-->321
*** Suche nach einem bestimmten String
hofa
String=hofa
Double=12
```

### 1.2.7. Die Klasse stack

---

### 1.2.8. Beispiel: t\_stack.cpp

---

```
// t_stack.cpp
// a.hofmann
// stack mit stl
// g++ t_stack.cpp -o t_stack.exe

#include <stack>
#include <iostream>
using namespace std;

int main(){
    stack<int> s;

    cout << "+++push: ";
    for (int i = 0; i < 10; ++i) {
        s.push(i);
        cout << i << " ";
    }
    cout << endl;

    cout << "---pop: ";
    while (!s.empty()) {
        cout << s.top() << " "; // oberstes Stackelement lesen
        s.pop();                // oberstes Stackelement wegnehmen
    }
}
```

```
    cout << endl;  
    return 0;  
}
```

output:

+++push: 0 1 2 3 4 5 6 7 8 9

---pop: 9 8 7 6 5 4 3 2 1 0

## 1.3. Iteratoren

---

Iteratoren sind Objekte, die sich wie Zeiger verhalten. Wie man einen Pointer über ein C++-Array bewegen kann, so kann man Iteratoren über Containerklassen bewegen.

Da Iteratoren das Bindeglied zwischen Algorithmen und Containerklassen sind, ist es wichtig ihre Funktionsweise zu verstehen.

Man greift normalerweise **immer über Iteratoren auf Containerklassen** zu.

Spätestens bei den Algorithmen geht es nicht anders, da Algorithmen als Argumente Iteratoren übergeben bekommen. D.h.

**Algorithmen greifen mittels Iteratoren auf die Objekte eines Containerklassens zu.**

Um welchen Iteratortyp es sich handelt, wird über die Containerklasse festgelegt. Zum Beispiel ist ein Vector-Iterator ein Iterator mit wahlfreiem Zugriff. Die Deklaration eines solchen Iterators könnte wie folgt aussehen:

```
vector<int>::iterator p;
```

Um die Deklaration eines Vektors uns noch einmal zu vergegenwärtigen hier noch einmal ein Beispiel:

```
vector<int> v;
```

Vector und dqueue unterstützen Random-Access-Iteratoren.

Alle anderen Containerklassen (list, set, map, ...) unterstützen Bidirektionale Iteratoren

Das nachfolgende Programm soll Random-Access-Iteratoren bei einem Vektor verdeutlichen. Die Iteratoren werden verwendet, um durch das vector-Objekt zu navigieren, es auszulesen und zu verändern.

### 1.3.1. Beispiel: t\_iterator.cpp

---

```
// t_iterator.cpp  
// ahofmann  
// g++ t_iterator.cpp -o t_iterator.exe
```

```
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]){
// Vector deklarieren
vector<int> v;

// Erster Random-Access-Iterator deklarieren
vector<int>::iterator itr;

// Zweiter Random-Access-Iterator deklarieren
vector<int>::iterator itr2;

// Vektor mit Zahlen 0..9 füllen
for (int i=0; i < 10; i++)
    v.push_back(i);

// Einsatz von Iterator zur Ausgabe
// Achtung v.end() zeigt auf die Adresse (v.back()+1)
for (itr = v.begin(); itr != v.end(); itr++)
    cout << *itr << " ";

cout << endl << endl;

// Elemente über Iterator verändern
for (itr = v.begin(); itr != v.end(); itr++)
    if (*itr % 2) // Bei allen ungeraden Zahlen 10 hinzuaddieren
        *itr += 10;

// Und wieder Ausgabe des Vektors
for (itr = v.begin(); itr != v.end(); itr++)
    cout << *itr << " ";

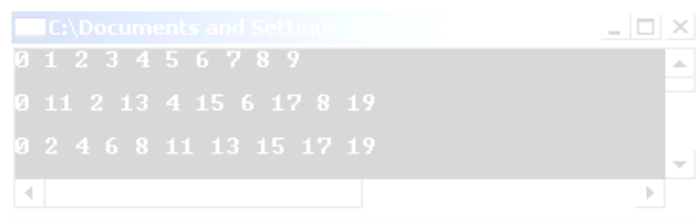
cout << endl << endl;

// Vektor-Elemente sortieren
// geht über Algorithmen besser. siehe weiter unten Kapitel Algorithmen
for (itr = v.begin(); itr != v.end()-1; itr++){
    for (itr2 = itr; itr2 != v.end(); itr2++){
        int iSwap;
        if (*itr2 < *itr){
            iSwap = *itr;
            *itr = *itr2;
            *itr2 = iSwap;
        }
    }
}

// Und wieder Ausgabe des Vektors
for (itr = v.begin(); itr != v.end(); itr++)
    cout << *itr << " ";

cout << endl << endl;
```

```
return 0;
}
```



### 1.3.2. Beispiel: crossreference (map, list, iteratoren)

Es soll eine Cross-Referenzliste erzeugt werden vergleichbar mit den Index-Seiten eines Buches. Einem Wort sind die Seitennummern oder Zeilennummer als Liste beigefügt:

Standardalgorithmen: 23, 123, 233

STL: 33, 234

...

Das folgende Programm liest die Datei: crossreference.txt und gibt die darin enthaltenen Wörter inkl. einer Liste von Zeilennummern aus.

☒ crossreference.txt

```
STL ist die Abkürzung für Standard Template Library.
Sie ist eine komplexe Sammlung von Templateklassen und
Templatefunktionen, welche viele bekannte und häufig
gebrauchte Algorithmen und Datenstrukturen kapseln.
Man findet in ihr zum Beispiel Vektoren
(das sind dynamische Arrays), Listen, Stacks,
sowie Such- und Sortieralgorithmen.
Die Standard Template Library ist seit
1994 ein Teil von C++.
```

Ausgabe:

....

sowie: 8,

und: 3,3,4,5,8,

viele: 4,

von: 3,10,

....

Bringen Sie folg. Programm zum Laufen

### 1.3.3. Beispiel: crossreference.cpp (m,u) crossreference.txt

```
// crossreference.cpp
```



```
// a.hofmann
// g++ crossreference.cpp -o crossreference.exe

#include <map>
#include <list>
#include <algorithm>
#include <string>
#include <sstream>
#include <fstream>
#include <iostream>
#include <cctype>
using namespace std;

//
class crossref{
private:
    ifstream fin;
    map<string, list<int> > m_map;

public:
    crossref(const string& filename);
    ~crossref();

    friend ostream& operator<< (ostream& o, const crossref& e);
};

crossref::crossref(const string& filename){
    int lineno= 0;
    string s, word;

    fin.open(filename.c_str()); // Datei öffnen
    if (! fin.fail()){

        // zeilenweise lesen
        lineno= 0;
        while (getline(fin,s, '\n')){
            lineno++;

            // zeile in wörter zerlegen: hier mit istringstream
            istringstream sin(s);
            while (!sin.eof()){
                sin >> word;

                if ( isalnum(word[0])) // wenns ein Wort ist
                    m_map[word].push_back(lineno);
            }
        }
    }
}

crossref::~~crossref(){
    fin.close();
}

ostream& operator<< (ostream& o, const crossref& e){
```

```
map<string, list<int> >::const_iterator it;
list<int>::const_iterator lines;

//über die map
for(it= e.m_map.begin(); it != e.m_map.end(); it++){
    o << endl << it->first << ": ";

    // über den vector
    for(lines= (it->second).begin();
        lines != (it->second).end(); lines++)

        o<< *lines << ", ";
    }

    return o;
}

int main(){
    crossref reader("crossreference.txt");

    cout << reader;

    return 0;
}
```

## 1.4. Algorithmen

---

Algorithmen sind Funktionen der STL, die über Iteratoren auf Container zugreifen, um diese zu verändern. Unter diese Kategorie fallen Kopier- und Sortieralgorithmen, welche die STL zur Verfügung stellt.

Um die Algorithmen verwenden zu können muss folgender Code eingefügt werden.

```
#include <algorithm>
using namespace std;
```

Es gibt eine Vielzahl von Algorithmen, sie alle hier zu beschreiben würde den Rahmen sprengen. Um das Prinzip zu beschreiben, wie Algorithmen auf Container angewendet werden, sollen folgende Beispiele besprochen werden:

Dies kleine Beispiel zeigt, wie man den Inhalt eines Vektors sortiert. Die Sortierfunktion erwartet zwei Iteratoren, die den Anfang und das Ende der Sortierung kennzeichnet. Alle Daten, die in dem Bereich liegen werden sortiert.

```
vector<int> vi(10);
vector<int>::iterator itr;

// Vektor mit Zufallszahlen füllen
```

```
for (itr = vi.begin(); itr != vi.end(); itr++)
    *itr = rand() % 100;

// Sortierung des Containers
sort(vi.begin(), vi.end());
```

Um die Sache noch praktischer werden zu lassen, folgt nun ein längeres Beispiel, welches die Verwendung von Sortier-, Kopier- und Tauschalgorithmen aufzeigt. Das Beispiel sollte soweit selbsterklärend sein, wenn man den Screenshot der Ausgabe noch mit zu Rate zieht.

### 1.4.1. Beispiel: t\_algorithmen.cpp

```
//t_algorithmen.cpp
//ahofmann
//g++ t_algorithmen.cpp -o t_algorithmen.exe

#include <vector>
#include <list>
#include <algorithm>

#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
    vector<int> vi(10);
    vector<int> vi2(10);
    vector<int> vi3(10);
    vector<int>::iterator itr;
    vector<int>::iterator itr2;

    cout << "Vektor 1 mit Zufallszahlen fuellen" << endl;

    // Vektor mit Zufallszahlen füllen
    for (itr = vi.begin(); itr != vi.end(); itr++)
        *itr = rand() % 100;

    // Inhalt des Vektors ausgeben
    for (itr = vi.begin(); itr != vi.end(); itr++)
        cout << *itr << " ";
    cout << endl << endl;

    cout << "Sortierung von Vektor 1." << endl;
    sort(vi.begin(), vi.end());
    // Inhalt des Vektors ausgeben
    for (itr = vi.begin(); itr != vi.end(); itr++)
        cout << *itr << " ";
    cout << endl;

    int iCount = 1;
    cout << "Vektor 2 mit Zahlenreihe fuellen." << endl;
    // Zweiter Vektor mit Zahlenreihe füllen
    for (itr = vi2.begin(); itr != vi2.end(); itr++)
        *itr = iCount++;
```

```
// Inhalt des zweiten Vektors ausgeben
for (itr = vi2.begin(); itr != vi2.end(); itr++)
    cout << *itr << " ";
cout << endl<< endl;

cout << "Inhalt von Vektor 3 ausgeben." << endl;
// Inhalt des dritten Vektors ausgeben
for (itr = vi3.begin(); itr != vi3.end(); itr++)
    cout << *itr << " ";
cout << endl << endl;

cout << "Kopiere die ersten 4 Werte von Vektor 2 in Vektor 3.";
cout << endl;
copy(vi2.begin(), vi2.begin()+4, vi3.begin());

// Inhalt des dritten Vektors ausgeben
for (itr = vi3.begin(); itr != vi3.end(); itr++)
    cout << *itr << " ";
cout << endl << endl;

cout << "Fuege die ersten zwei Elemente aus Vektor 1" << endl;
cout << "an der fuenften Stelle von Vektor 3 ein." << endl;
vi3.insert(vi3.begin()+5, vi.begin(), vi.begin()+2);

// Inhalt des dritten Vektors ausgeben
for (itr = vi3.begin(); itr != vi3.end(); itr++)
    cout << *itr << " ";
cout << endl << endl;

cout << "Tausche den Inhalt von Vektor 1 und Vektor 2" << endl;
swap_ranges(vi.begin(), vi.end(), vi2.begin());

cout << "Inhalt von Vektor 1 ausgeben." << endl;
// Inhalt des dritten Vektors ausgeben
for (itr = vi.begin(); itr != vi.end(); itr++)
    cout << *itr << " ";
cout << endl;
cout << "Inhalt von Vektor 2 ausgeben." << endl;
// Inhalt des dritten Vektors ausgeben
for (itr = vi2.begin(); itr != vi2.end(); itr++)
    cout << *itr << " ";
cout << endl;

return 0;
}
```

```

C:\Documents and Settings\Administrator\My Documents>
Vektor 1 mit Zufallszahlen fuellen
41 67 34 0 69 24 78 58 62 64

Sortierung von Vektor 1.
0 24 34 41 58 62 64 67 69 78
Vektor 2 mit Zahlenreihe fuellen.
1 2 3 4 5 6 7 8 9 10

Inhalt von Vektor 3 ausgeben.
0 0 0 0 0 0 0 0 0 0

Kopiere die ersten 4 Werte von Vektor 2 in Vektor 3.
1 2 3 4 0 0 0 0 0 0

Fuege die ersten zwei Elemente aus Vektor 1
an der fuenften Stelle von Vektor 3 ein.
1 2 3 4 0 0 24 0 0 0 0

Tausche den Inhalt von Vektor 1 und Vektor 2
Inhalt von Vektor 1 ausgeben.
1 2 3 4 5 6 7 8 9 10
Inhalt von Vektor 2 ausgeben.
0 24 34 41 58 62 64 67 69 78

```

## 1.5. Beispiel:Email Adressbuch – Die Klasse Map

"Associative container" eignen sich für schnelles Abfragen von Daten, die mit Schlüsseln arbeiten. Intern werden spezielle Datenstrukturen (Bäume mit keys zur Identifikation der Daten,...) eingesetzt, um den Datenzugriff zu beschleunigen.

**set**, **map** verwenden sog. **unique keys**, d.h. für jeden Schlüssel gibt es **nur einen** Datensatz.

Das Email Adressbuch könnte folg. Aufbau haben:

### 1.5.1. Beispiel: t\_adb\_map.cpp

```

// t_adb_map.cpp
// ahofmann
// Email-Adressbuch mit STL map.h
// g++ t_adb_map.cpp -o t_adb_map.exe

#include <map>

#include <iostream>
#include <string>
using namespace std;

class adb_record {
public:
    adb_record() : nickname (""),email(""), comment("") {}
    adb_record(string n, string e, string c) : nickname (n),
        email (e), comment (c) {}
    adb_record(const adb_record& a) : nickname (a.nickname),
        email (a.email), comment (a.comment) {}

    friend ostream& operator<< (ostream& os, const adb_record& r);

```

```

    bool operator< (const adb_record& e) const {
        return nickname < e.nickname;
    }

private:
    string nickname;
    string email;
    string comment;
};

// friend Funktion
ostream& operator<< (ostream& os, const adb_record& r) {
    os << r.nickname << endl << r.email << endl << r.comment << endl;
    return os;
}

```

```

// -----
// MAIN
// -----

int main(){
// 1. Datensätze erstellen/lesen
//-----
adb_record record1("toni", "anton.hofmann@unix.at", "ich bin es");
adb_record record2("priska", "priska.hofmann@unix.at", "meine tochter");
adb_record record3 ("diana", "diana.hofmann@unix.at", "meine tochter");

// 2. Map definieren(=assoziativer Speicher): <MAP: key, value, compare>
//-----
map <string, adb_record, less<string> > adb_map;

cout << "\nDemo: Email-Adressbuch und STL: map()!\n";

// 3. Einfuegen
//-----
adb_map.insert (make_pair (string("toni"), record1));
adb_map.insert (make_pair (string("priska"), record2));
adb_map.insert (make_pair (string("diana"), record3));

// 4. Suchen mittels Iterator, MAP-Iterator
//-----
map <string, adb_record, less<string> >::iterator i;

cout << "\nSuche nach diana !\n";

i= adb_map.find (string("diana"));
if (i == adb_map.end() )
    cout << "ADB_RECORD: diana not found";
else
{
    cout << "\n<KEY>\n" << (*i).first << "\n</KEY>\n";
    cout << "\n<VALUE>\n" << (*i).second << "</VALUE>\n";
}
}

```

```
// 5. Zugriff mittels [] moeglich
//-----

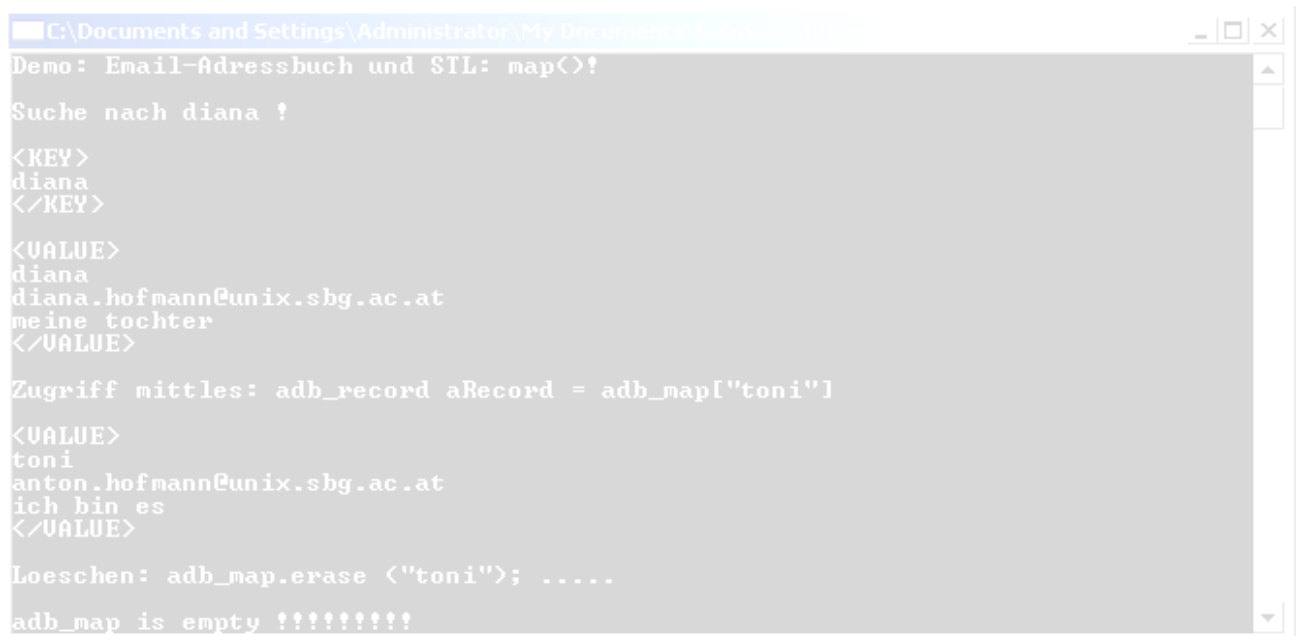
cout << "\nZugriff mittles: adb_record aRecord = adb_map[\"toni\"]\n";
adb_record aRecord = adb_map["toni"];
cout << "\n<VALUE>\n" << aRecord << "</VALUE>\n";

// 6. Loeschen
//-----

cout << "\nLoeschen: adb_map.erase (\"toni\"); ..... \n";
adb_map.erase ("toni");
adb_map.erase (adb_map.begin() );
adb_map.erase (adb_map.begin(), adb_map.end() );
if (adb_map.empty() )
    cout << "\nadb_map is empty !!!!!!!!!";

return 0;
}
```

Folgende Bildschirmausgabe auf der Basis des obigen Programmes:



```
C:\Documents and Settings\Administrator\My Documents\hbf\cpp\02-cpp-stl.odt
Demo: Email-Adressbuch und STL: map<>!

Suche nach diana ?

<KEY>
diana
</KEY>

<VALUE>
diana
diana.hofmann@unix.sbg.ac.at
meine tochter
</VALUE>

Zugriff mittles: adb_record aRecord = adb_map["toni"]

<VALUE>
toni
anton.hofmann@unix.sbg.ac.at
ich bin es
</VALUE>

Loeschen: adb_map.erase ("toni"); .....

adb_map is empty !!!!!!!!!
```