21. Die STL (Standard Template Library)

21.1 Allgemeines

Um nicht Standard-Algorithmen für **Sequenzen** (Listen, Vektoren, ...) von Objekten immer wieder neu implementieren zu müssen (und dies dann noch pro Datentyp einmal) hat man sich etwas einfallen lassen:

a) Man schreibt einen **Container**, d.h. eine Klasse, die die gesamte Funktionalität für eine Sequenz beinhaltet. Als **sequentiellen Container** bezeichnet man einen solchen, der auf einer Listenstruktur basiert (Bsp.: `list`). **Assoziative Container** bestehen aus 2 Listen: Die Liste der Schlüsselfelder (keys) und die Liste der Wertefelder (values). Wenn man einen Wert (value) hineinschreibt, dann muß man angeben, unter welchem Schlüssel (key) dieser gespeichert werden soll (Bsp.: `map`).

b) In einem weiteren Schritt sorgt man dafür, dass dieser Container **beliebige Elemente** aufnehmen kann, wozu man das **Template**-Konzept (Vorlagen-Konzept) einführt:

Beispiel für die Definition eines Templates:

```
template<class T>
class SmartPtr
{
      public:
            SmartPtr(T* pT = NULL) : m_pT(pT) {}
            SmartPtr(SmartPtr<T>& Obj) : m_pT(Obj.GetPtr())
            { Obj.Release(); }
            ~SmartPtr() { delete m_pT; }
            void Release() { m_pT = NULL; }
            T* GetPtr() { return m_pT; }
            template<class C>
            bool Transform(SmartPtr<C>& Obj)
            {
                  T* pT = dynamic_cast<T*>(Obj.GetPtr());
                  if(!pT)
                        return false;
                  m_pT = pT;
                  return true;
            }
            T* operator->() { return m_pT; }
            T& operator*() { return *m_pT; }
            ...
      private:
            T* m_pT;
};
```

Instanziierung dieser Vorlage für den Typ `MyClass`:

```
SmartPtr<MyClass> pMyClass(new MyClass);
```

c) Als letztes schafft man sich dann noch ein **Zugriffsobjekt**, über welches man auf jedes Element der Sequenz zugreifen kann, einen sogenannten **Iterator**:

Beispiel:

```
list<int> listInteger;
list<int>::iterator it;

for(it = listInteger.begin();it != listInteger.end();++it)
     (*it).push_back(5); //Zugriff auf die Methode push_back()
for(it = listInteger.begin();it != listInteger.end();++it)
     printf("%d\n",(*it)); //Ausgabe des gespeicherten Wertes
```

Diese Dinge haben Informatiker schon seit langem erledigt und optimiert, so dass **1994** Alexander **Stepanov** und Meng **Lee** das Ergebnis ihrer langjährigen Forschungsarbeiten im Bereich der Standard-Algorithmen (bei der Firma HP) erfolgreich als **STL** (**Standard-Template-Library**) in das ISO/ANSI-C++-Werk einbringen konnten.

Es gibt **verschiedene Implementierungen** der STL. Hier ein paar Beispiele für portable Implementierungen:

- STL von STLport **http://www.stlport.org/**
- STL von Rogue Wave **http://www.ccd.bnl.gov/bcf/cluster/pgi/pgC++_lib/stdlib.htm**
- STL von SGI (Silicon Graphics Inc.) **http://www.sgi.com/tech/stl/**
- STL von HP (Hewlett-Packard) bzw. D.R. Musser **ftp://ftp.cs.rpi.edu/pub/stl/**

Beispiele für Container-Klassen der STL:

`vector`	→	Eindimensionales Feld
`list`	→	Doppelt verkettete Liste
`queue`	→	Schlange
`deque`	→	Schlange mit 2 Enden
`stack`	→	Stack
`set`	→	Sortierte Menge
`bitset`	→	Sortierte Menge von booleschen Werten
`map`	→	Sortierte Menge (Schlüsselfelder), die mit anderer Menge (Wertefelder) assoziiert ist

Besonderheiten:

- **Token:**

  Die **Token** (Steuerzeichen) < und > sind etwas problematisch: Bei **Verschachtelungen von Templates** kann eine **Verwechslung der doppelten Klammerung mit << oder >>** passieren. Deshalb muss man bei Verschachtelungen ggf. ein **SPACE** einfügen.

Beispiel:

```
Falsch:     set<long,less<long>> setMyObjects;

Richtig:    set<long,less<long> > setMyObjects;
                                ↑
```

- **Iteratoren:**

  Iterator-Objekte arbeiten wie Zeiger und entkoppeln die Algorithmen von den Daten, so dass diese typunabhängig werden. Sie sind praktisch Schnittstellen-Objekte.

  Innerhalb von **Read-Only**-Member-Funktionen (`const`) muss man mit

  ```
  const_iterator
  ```

  statt `iterator` arbeiten.

- **Effektivität:**

  Die STL bietet ein **optimiertes dynamisches Heap-Speicher-Management** mit **generischem Code**, was kaum zu überbieten sein dürfte. So sollte man sich intensiv der STL bedienen, wenn man etwas **"Dynamisches" auf dem Heap** benötigt. Kann man die zu lösende Aufgabe jedoch mittels nicht- dynamischem Array lösen (keine dynamische Länge des Arrays; kein Code, sondern nur Daten im Array) dann sollte man prüfen, ob die Nutzung eines Arrays auf dem Stack nicht doch effektiver ist.

  Beispiel:

  ```
  vector<int> vectValues;
  ```
  → Container auf dem Stack, der über die Methode `push_back()` intern Heap allokiert, um Daten zu speichern

  ```
  int aValues[100];
  ```
  → Array auf dem Stack ("purer" Stack-Speicher)

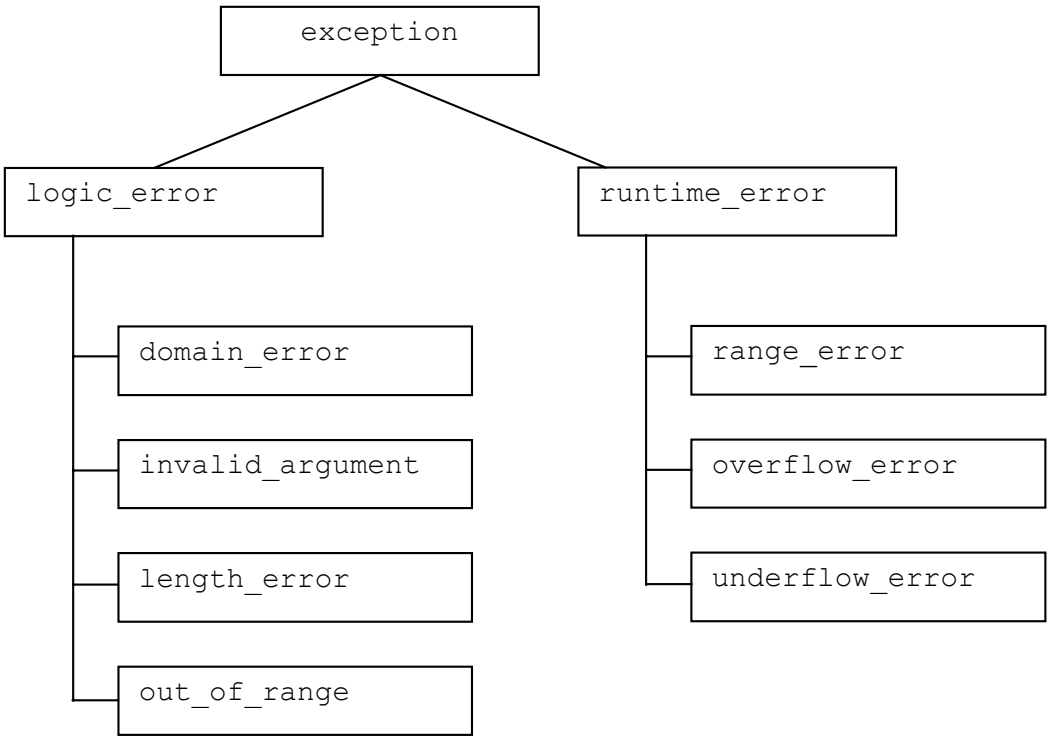
- **Güte der Implementierung und `hash_map`:**

  Möchte man eine Implementierung der STL testen, dann sollte man erst mal einen Blick auf die `hash_map` werfen. Besonders schwach ist die Implementierung, wenn es gar keine `hash_map` gibt. Die Untersuchung der Schnelligkeit der `hash_map` sagt viel aus → wenn so etwas Kompliziertes wie eine `hash_map` gut funktioniert, dann ist das schon mal ein sehr gutes Zeichen. Zum Testen kann man zunächst als Schlüssel (key) den Typ `string` verwenden.

Exceptions der STL:

Die STL definiert in der Header-Datei **`stdexcept`** folgende Exceptions:

```
namespace std
{
      class logic_error;
            class domain_error;
            class invalid_argument;
            class length_error;
            class out_of_range;
      class runtime_error;
            class range_error;
            class overflow_error;
            class underflow_error;
};
```



Die Exceptions sind alle von `exception` abgeleitet (**`what()`** → Erfragen der Fehlermeldung):

```
class exception
{
      public:
            exception() throw();
            exception(const exception& rhs) throw();
            exception& operator=(const exception& rhs) throw();
            virtual ~exception() throw();
            virtual const char *what() const throw();
};
```

21.2 Nutzung der STL von STLport

21.2.1 Allgemeines

Um die STL-Implementierung 'STLport' von der Firma 'STLport' (Nomen est Omen) nutzen zu können, muss man sich zunächst die erforderlichen Dateien von **http://www.stlport.org** herunterladen (Freeware). Diese STL liegt zunächst nur als Quell-Code vor.

21.2.2 STLport mit GNU unter Linux

Die Nutzung von STLport mit GNU unter Linux wird durch folgende Maßnahmen vorbereitet:

- Die Dateien in das private Entwicklungs-Verzeichnis (hier: `/Peter/Dev/`) kopieren:

  ```
  mkdir /home/Peter/Dev/STLport_STL
  cp * /home/Peter/Dev/STLport_STL -r
  ```

- Ins `src`-Verzeichnis wechseln:

  ```
  cd /home/Peter/Dev/STLport_STL/src
  ```

- Vorbereiten (nicht bauen):

  ```
  make -f gcc-linux.mak prepare
  ```

- Ins `stlport`-Verzeichnis wechseln:

  ```
  cd /home/Peter/Dev/STLport_STL/stlport
  ```

- Die Datei `stl_user_config.h` editieren:

  ```
  vi stl_user_config.h
  ```

  Kommentarzeichen `//` vor folgende Anweisung:

  ```
  // #define _STLP_NO_OWN_IOSTREAMS 1
  ```

  Dadurch wird die Nutzung der `STLport` iostreams abgeschaltet und es werden stattdessen die bereits vorhandenen iostreams gewrapped.

Nun kann STLport bei der Programmierung genutzt werden:

In einem neuen Projekt

```
/home/Peter/Dev/NewProject
```

nutzt man als **`include`** Pfad

```
../STLport_STL/stlport
```

und als **`lib`** Pfad

```
../STLport_STL/lib
```

Man würde also die Datei `test.cpp` wie folgt bauen:

```
g++ -I../STLport_STL/stlport test.cpp -L..STLport_STL/lib/
```

Beispiel:

`test.cpp`:

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
      string strTest("Hello World");
      printf("%s\n",strTest.c_str());
      return 0;
}
```

`makefile`:

```
#
# Linker:
#

test: test.o
      g++ test.o -L..STLport_STL/lib/ -otest.exe

#
# Compiler:
#

test.o: test.cpp #dependencies
      g++ -c test.cpp -I../STLport_STL/stlport
```

21.2.3 STLport mit Visual C++ unter Windows

Die Nutzung von STLport unter Visual C++ wird durch folgende Maßnahmen vorbereitet:

- In der Datei **"\stlport\stl_site_config.h"** sind folgende Einstellungen **vor dem Bau** der STL vorzunehmen:
    - **Multithreading** aktivieren:

```
//#define _NOTHREADS //aktiviert
```

    - STL zwingen, einfaches (nicht optimiertes) **new** zum Allokieren zu benutzen:

```
#define _STLP_USE_NEWALLOC 1
```

- Make-Datei (makefile) im Unterverzeichnis **/src** für den Compiler **umbenennen**:

  "vc6.mak" → "makefile"

- Auf der **Kommandozeile** (Console) im Verzeichnis **/src bauen**:

  "nmake all"

- Dem **Compiler/Linker die Pfade der Dateien mitteilen**. Sie sollten **vor den Pfaden der Compiler/Linker-Bibliotheken** in der Liste auftauchen → Menu **"Tools.Options.Directories"**:

  Include: L:\...**stlport**
  Library Files: L:\...**lib**
  Source Files: L:\...**src**

  wobei **"L:\..."** für den Hauptpfad steht, z.B. **"D:\STLport-4.0"**

- In den Projekt-Settings sind unter **C++.Code Generation** die zu verwendenden **Runtime-Libraries** anzugeben:

DEBUG:	Debug Multithreaded DLL
RELEASE:	Multithreaded DLL

- Im Code kann man nun folgende warnings abschalten

```
//Keine Warn. 'unrefer.inline function has been removed':
#pragma warning(disable:4514)
//Keine Warnung 'truncated identifier ...':
#pragma warning(disable:4786)
```

21.3 STL-Header-Dateien

21.3.1 Aufbau: Die Endung ".h" fehlt

Bei den **Header-Dateien** der STL gibt es einige **Besonderheiten**:

- Sie haben keine Endung, also auch **nicht `".h"`**
- Der Code befindet sich im **`namespace std`**

  Beispiel:

```
//*****************************************
// file: 'list'
//*****************************************

namespace std
{
      ...
}
```

- Die **normalen C-Header** werden auch im **`namespace std`** angeboten, wobei jedoch der **Name um das Präfix `"c"` erweitert** wird und die **Endung `".h"` weggelassen** wird

  Beispiel:

```
//*****************************************
// file: 'cmath'
//*****************************************

namespace std
{
      #include <math.h>
}
```

21.3.2 Nutzung: "using namespace std"

Wenn man also **Header der STL nutzt**, dann muss man grundsätzlich den **`namespace std`** mit in den global scope aufnehmen oder alle Zugriffe über **`std::`** durchführen.

Beispiel:

```
#include <list>
using namespace std;
int main()
{
      list<int> listValues;
      listValues.push_back(3);
      listValues.push_back(4);
      int i = listValues.front();
      return 0;
}
```

21.4 Wichtige STL-Member-Variablen und Methoden

Für alle Container der STL gibt es einen **gemeinsamen** Satz von Member-Variablen und Methoden:

Wichtige STL-Member-Variablen:

value_type
→ Datentyp der Elemente (values)
key_type
→ Datentyp der Schlüssel (keys) bei assoziativen Containern (wie map)
size_type
→ Einheit der Längen- bzw. Größenangaben
difference_type
→ Einheit des Iterierens zum nächsten Element

iterator
→ Zeiger **`value_type*`**, der bei ++ nach vorne dreht
const_iterator
→ Zeiger **`const value_type*`**, der bei ++ nach vorne dreht

reverse_iterator
→ Zeiger **`value_type*`**, der bei ++ zurück dreht
reverse_const_iterator
→ Zeiger **`const value_type*`**, der bei ++ zurück dreht

reference
→ Referenz **`value_type&`**
const_reference
→ Referenz **`const value_type&`**

Wichtige STL-Methoden:

Allgemeiner Zugriff:

empty()
→ Prüft, ob der Container leer ist
size()
→ Anzahl der Elemente
max_size()
→ Maximale mögliche Anzahl von Elementen

for_each()
→ for-Schleife über alle Elemente
begin()
→ Zeiger auf das erste Container-Element (für iterator)
end()
→ Zeiger hinter das letzte Container-Element (kein gültiges Element)
rbegin()
→ Zeiger auf das letzte Container-Element (für reverse_iterator)
rend()
→ Zeiger vor das erste Container-Element (kein gültiges Element)

front()
→ Erstes Element
last()
→ Letztes Element
[index]
→ Element an der Stelle **index** (ungeprüfter Zugriff)
at(index)
→ Element an der Stelle **index** (geprüfter Zugriff)

key_compare()
→ Vergleich der Schlüsselfelder (keys) assoziativer Container

Manipulationen:

clear()
→ Alle Elemente eines Containers löschen

insert()
→ Element richtig sortiert einfügen (`set`, `map`)
erase()
→ Element in einer sortierten Sequenz (`set`, `map`) löschen

push_back()
→ Element an den Schluss einer unsortierten Sequenz anhängen
pop_back()
→ Element vom Schluss einer unsortierten Sequenz entfernen
push_front()
→ Element vor den Anfang einer unsortierten Sequenz stellen
pop_front()
→ Element vom Anfang einer unsortierten Sequenz entfernen

sort()
→ Elemente in unsortierter Sequenz sortieren
stable_sort()
→ Elemente sortieren, wobei Reihenfolge gleicher Elemente bleibt
partial_sort()
→ Den ersten Teil einer Sequenz sortieren
partial_sort_copy()
→ Elemente kopieren und den ersten Teil der Sequenz sortieren
nth_element()
→ Das n-te Element an die richtige Stelle sortieren
merge()
→ 2 sortierte Sequenzen verschmelzen
inplace_merge()
→ 2 sortierte Teilsequenzen einer Sequenz verschmelzen
unique()
→ Aufeinanderfolgende Duplikate entfernen (vorher: **`sort()`**)
unique_copy()
→ Elemente kopieren und aufeinanderfolgende Duplikate entfernen
remove()
→ Element (alle Vorkommnisse) in unsortierter Sequenz löschen
remove_if()
→ Elemente mit bestimmtem Inhalt entfernen
remove_copy()
→ Elemente kopieren und die mit bestimmten Inhalt entfernen
remove_copy_if()
→ Elemente kopieren und die mit bestimmten Inhalt entfernen
replace()
→ Inhalt der Elemente durch anderen Inhalt ersetzen
replace_if()
→ Inhalt der Elemente durch anderen Inhalt ersetzen (mit Bedingung)
replace_copy()
→ Elemente kopieren und dabei Inhalt ersetzen
replace_copy_if()
→ Elemente kopieren und dabei Inhalt ersetzen (mit Bedingung)

copy()
→ Alle Elemente in gegebener Reihenfolge kopieren
copy_backwards()
→ Alle Elemente in umgekehrter Reihenfolge kopieren
reverse()
→ Umkehrung der Reihenfolge der Elemente
reverse_copy()
→ Elemente kopieren und ihre Reihenfolge umkehren

swap()
→ 2 Elemente vertauschen
iter_swap()
→ 2 Elemente, auf die die Iteratoren zeigen, vertauschen
swap_ranges()
→ 2 Bereiche von Elementen vertauschen
rotate()
→ Elemente rotieren
rotate_copy()
→ Elemente kopieren und rotieren
random_shuffle()
→ Elemente zufällig mischen
partition()
→ Bestimmte Elemente nach vorne platzieren
stable_partition()
→ Bestimmte Elemente unter Beibehaltung ihrer Reihenfolge . . .

fill()
→ Alle Container-Elemente füllen
fill_n()
→ n Container-Elemente füllen
generate()
→ Alle Elemente mit dem Ergebnis einer Operation füllen
generate_n()
→ n Elemente mit dem Ergebnis einer Operation füllen

set_union()
→ Sortierte **Vereinigungsmenge** zweier Sequenzen erzeugen
set_intersection()
→ Sortierte **Schnittmenge** zweier Sequenzen erzeugen
set_difference()
→ Andere Menge ausschließen
set_symmetric_difference()
→ Schnittmenge ausschließen

make_heap()
→ Sequenz als Heap einrichten
push_heap()
→ Element auf die als Heap eingerichtete Sequenz drauflegen
pop_heap()
→ Element von der als Heap eingerichteten Sequenz wegnehmen
sort_heap()
→ Als Heap eingerichtete Sequenz sortieren

Analysen:

min()
→ Das Element von zweien mit dem kleineren Inhalt bestimmen
max()
→ Das Element von zweien mit dem größeren Inhalt bestimmen

find()
→ Bestimmtes Element finden
find_if()
→ Bestimmtes Element finden (Bedingung)
find_first_of()
→ Bestimmtes Element finden (erstes Vorkommnis)
adjacent_find()
→ Benachbartes Elemente-Paar finden
min_element()
→ Das Element einer Sequenz mit dem kleinsten Inhalt finden
max_element()
→ Das Element einer Sequenz mit dem größten Inhalt finden
lower_bound()
→ Erstes Element mit bestimmtem Inhalt in sortierter Sequenz finden
upper_bound()
→ Letztes Element m.bestimmtem Inhalt in sortierter Sequenz finden
search()
→ Nach einer Teilsequenz mit bestimmtem Inhalt suchen
search_n()
→ Nach einer Teilsequenz mit n übereinstimmenden Inhalten suchen
find_end()
→ Letztes Auftreten einer Teilsequenz suchen
equal_range()
→ Teilsequenz mit bestimmtem Inhalt in sortierter Sequenz finden
mismatch()
→ Erstes unterschiedliches Element in 2 Sequenzen finden
next_permutation()
→ nächste Permutation in lexikographischer Reihenfolge finden
prev_permutation()
→ vorherig.Permutation in lexikographischer Reihenfolge finden

count()
→ Anzahl der Elemente mit bestimmtem Inhalt finden
count_if()
→ Anzahl der Elemente mit bestimmtem Inhalt finden
binary_search()
→ Prüfen, ob Element mit best. Inhalt in sortierter Sequenz enthalten

equal()
→ Prüfen, ob 2 Sequenzen gleich sind
includes()
→ Prüfen, ob Sequenz eine Teilsequenz einer anderen Sequenz ist
lexicographical_compare()
→ Prüfen, ob 2 Sequenzen lexikographisch gleich sind

21.5 Generierung von Sequenzen über STL-Algorithmen

21.5.1 back_inserter()

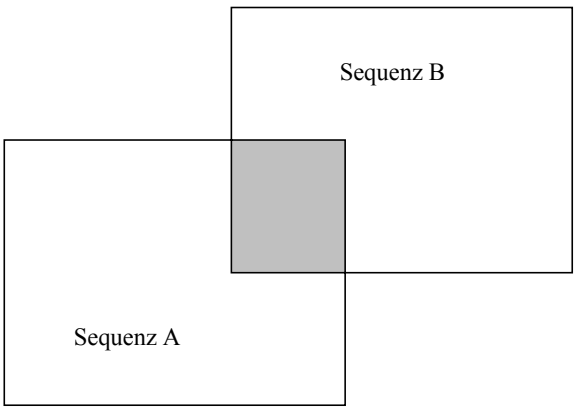
Die Funktion `back_inserter()` dient dazu, die von einem STL-Algorithmus generierte lokale Sequenz in einen vom Aufrufer bereitgestellten Container zu kopieren. Die Anwendung von back_inserter() entspricht also in etwa der Anwendung eines Referenz-Parameters beim Aufruf einer Funktion.

Beispiel:

```
list<MyClass> listDiff;
set_difference(      listA.begin(),listA.end(),
                     listB.begin(),listB.end(),
                     back_inserter(listDiff));
```

21.5.2 Schnittmenge (set_intersection)

Die STL kann eine Schnittmenge zweier Sequenzen ermitteln:

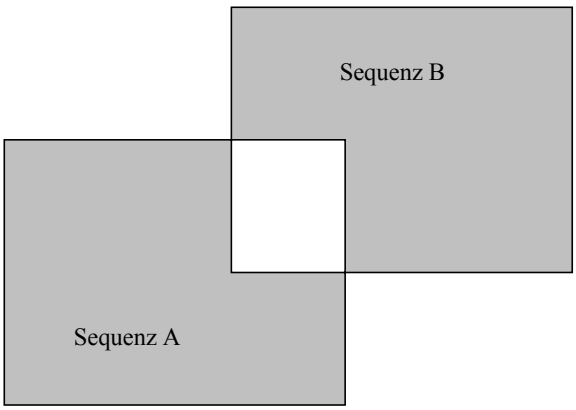



```
listA.sort();
listB.sort();

list<MyClass> listIntersection;
set_intersection(    listA.begin(),listA.end(),
                     listB.begin(),listB.end(),
                     back_inserter(listIntersection));
```

21.5.3 Schnittmenge ausschließen (set_symmetric_difference)

Die STL kann die Schnittmenge zweier Sequenzen von der Gesamtmenge ausschließen:

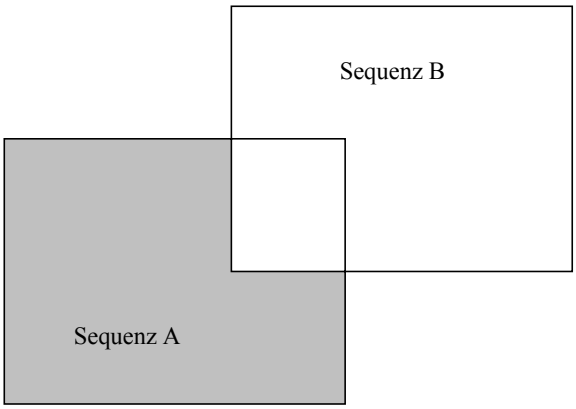



```
listA.sort();
listB.sort();

list<MyClass> listSymDiff;
set_symmetric_difference(  listA.begin(),listA.end(),
                           listB.begin(),listB.end(),
                           back_inserter(listSymDiff));
```

21.5.4 Sequenz ausschließen (set_difference)

Die STL kann eine Sequenz von der Gesamtmenge ausschließen:

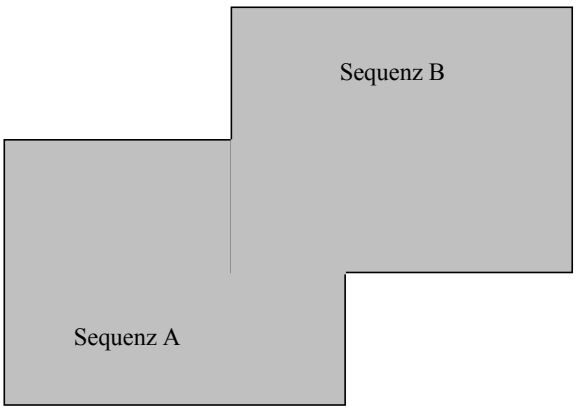



```
listA.sort();
listB.sort();

list<MyClass> listDiff;
set_difference( listA.begin(),listA.end(),
                listB.begin(),listB.end(),
                back_inserter(listDiff));
```

21.5.5 Vereinigungsmenge bilden (set_union)

Die STL kann eine Vereinigungsmenge zweier Sequenzen bilden:

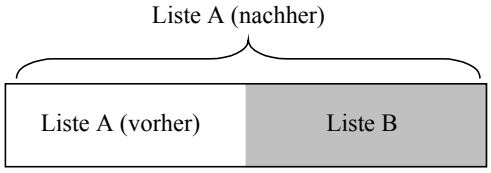



```
listA.sort();
listB.sort();

list<MyClass> listUnion;
set_difference( listA.begin(),listA.end(),
                listB.begin(),listB.end(),
                back_inserter(listUnion));
```

21.5.6 Liste an eine andere Liste anhängen (list::insert)

Die STL kann eine Liste an eine andere Liste anhängen:




```
listA.insert(   listA.begin(),listA.end(),
                listB.begin(),listB.end());
```

21.6 Wichtige Regeln

21.6.1 Einbinden der STL

Es gibt 2 Besonderheiten beim Bau einer Software mit der STL:

- Lange Namen der STL

Die STL bildet durch Verschachtelungen sehr lange Namen. Microsoft's Visual C++ hat dann Probleme diese Namen in die Debug-Informations-Datei einzutragen, da dort eine maximale Länge von 255 Zeichen erlaubt ist. Deshalb kürzt der Compiler diese und bringt eine Warnung:

→ warning: identifier was truncated to '255' characters in the debug information.

- Nichtreferenzierte `inline`-Funktionen der STL

Normalerweise werden nicht alle `inline`-Funktionen der STL durch den Code des Nutzers referenziert. Visual C++ wirft diese Funktionen beim Bau weg und warnt den Nutzer:

→ warning: unreferenced inline function has been removed.

Um diese lästigen Warnungen bei der Kompilierung mit Visual C++ abzuschalten, benutzt man folgende `#pragma`-Anweisungen:

```
#pragma warning(disable:4786)
#pragma warning(disable:4514)
```

Die Aufnahme der STL in den Quell-Code geschieht in einer zentralen Header-Datei (`Settings.h`), wobei nach dem Abschalten der Warnungen die Algorithmen der STL über **`#include <algorithm>`** und **danach** die **benötigten STL-Container** einzubinden sind. **Zum Schluss** nimmt man **`std` in den global namespace** auf, d.h. alle Namen im namespace `std` werden zu globalen Namen.

Beispiel:

```
//Keine Warnung wg. verkürzten Namen oder beseitigten
//inline-Funktionen bei MS VC++:
#if  defined(_MSC_VER) &&
     !defined(__MWERKS__) &&
     !defined(__ICL) &&
     !defined(__COMO__)
          #pragma warning(disable:4786)
          #pragma warning(disable:4514)
#endif

//STL-Algorithmen:
#include <algorithm>
//STL-Container:
#include <list>
#include <set>
//std in global space aufnehmen:
using namespace std;
```

21.6.2 Die benötigten Operatoren implementieren

Die Container der STL benötigen Objekte, für die ein Vergleich möglich ist, d.h. es muss auf Gleichheit und auf Kleiner geprüft werden können. Wenn man eine eigene Klasse für die Objekte eines Containers schreibt, dann müssen `operator==()` und `operator<()` implementiert werden. Außerdem wird in manchen Fällen der Default-Konstruktor gebraucht.

Beispiele:

Minimal-Implementierung:

```
class MyClass
{
      public:
            explicit MyClass(int nID = 0) //Default
                  : m_nID(nID)
            {
            }
            int GetID() const { return m_nID; }
            bool operator==(const MyClass& Obj) const
            {
                  if(Obj.GetID() == m_nID)
                        return true;
                  return false;
            }
            bool operator<(const MyClass& Obj) const
            {
                  if(m_nID < Obj.GetID())
                        return true;
                  return false;
            }
      private:
            int m_nID;
};
```

Implementierung mit globalen Operatoren und einem Zuweisungsoperator:

```
class MyClass
{
      public:
            MyClass(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            void SetID(int nID) { m_nID = nID; }
            MyClass& operator=(const MyClass& Obj)
            {
                  if(this == &Obj)
                        return *this;
                  SetID(Obj.GetID());
                  return *this;
            }
      private:
            int m_nID;
};
```

```
//Globale Operatoren:

inline bool operator==(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs.GetID() != rhs.GetID())
		return false;
	return true;
}
inline bool operator!=(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs == rhs)
		return false;
	return true;
}
inline bool operator<(const MyClass& lhs,const MyClass& rhs)
{
	if(lhs.GetID() < rhs.GetID())
		return true;
	return false;
}
```

21.6.3 Iterator: ++it statt it++ benutzen

Man sollte den **Präfix-Operator `++it`** immer dem Postfix-Operator `it++` vorziehen (Bsp.: im Kopf einer `for`-Schleife), da dessen **Performance besser** ist.

Beispiel:

```
int main()
{
	list<MyClass> listObjs;
	listObjs.push_back(2);
	listObjs.push_back(1);
	listObjs.push_back(1);
	listObjs.push_back(3);
	listObjs.sort();
	listObjs.unique();
	list<MyClass>::iterator it;
	for(it = listObjs.begin();it != listObjs.end();++it)
	{
		printf("%d\n",(*it));
	}
	return 0;
}
```

Grund:

Der Postfix-Operator benötigt ein **zusätzliches temporäres Objekt**, welches **konstruiert**, für die Rückgabe **kopiert** (Wert-Rückgabe) und dann noch **destruiert** werden muss:

```
template<class T>
class iterator
{
        public:
                ...
                iterator& operator++();       //Präfix (++it)
                iterator operator++(int);     //Postfix (it++)
        protected:
                T& container;
                T::iterator iter;
                ...
};

iterator& iterator::operator++()     //Präfix (++it)
{
        ++iter;
        return *this;
}

iterator iterator::operator++(int)   //Postfix (it++)
{
        iterator temp = *this;
        ++(*this);
        return temp; //der Wert vor der Operation wird zurückgegeben
}
```

*21.6.4 Löschen nach find(): Immer über Iterator (it) statt über den Wert (*it)*

Problem:

Das **Löschen über einen Wert** (**`*it`**) erfordert zunächst immer ein **internes `find()`** zum Suchen der Speicherstelle, die diesen Wert beinhaltet. Wenn man gerade erst `find()` ausgeführt hat, um herauszufinden, ob sich das Element überhaupt in der Sequenz befindet, dann verschwendet man beim Löschen über den Wert nochmal die Zeit für ein `find()`.

Abhilfe:

Man merkt sich den **Iterator** (**`*it`**), welchen `find()` zurückliefert, und operiert direkt über diesen auf der Speicherstelle.

Beispiel:

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <list>
#include <set>
using namespace std;

int main()
{
	MyClass Obj1(1);
	MyClass Obj2(2);
	MyClass Obj3(3);
	MyClass Obj3(4);

	set<MyClass> setObjs;
	setObjs.insert(Obj1);
	setObjs.insert(Obj2);
	setObjs.insert(Obj3);
	setObjs.insert(Obj4);
	int i = 0;
	set<MyClass>::iterator it = setObjs.find(Obj3);
	if(it != setObjs.end())
	{
		i = (*it).GetID();
		setObjs.erase(it); //nicht setObjs.erase(*it)!!!
	}

	return 0;
}
```

Anmerkung:

`erase()` kann natürlich ohne vorhergehendes `find()` angewendet werden!

21.6.5 map: Nie indizierten Zugriff [ ] nach find() durchführen

Problem:

Der **indizierte Zugriff** auf eine map mit dem []-Operator erfordert zunächst immer ein **internes `find()`** zum Suchen der Speicherstelle, die diesen Wert beinhaltet. Wenn man gerade erst `find()` ausgeführt hat, um zu wissen, ob sich das Element überhaupt in der Sequenz befindet, dann verschwendet man nochmal die Zeit für ein `find()`.

Abhilfe:

Man merkt sich den Iterator, welchen find() zurückliefert, und operiert direkt über diesen auf der Speicherstelle.

Beispiel:

```
class MyClass
{
      public:
            explicit MyClass(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            bool operator==(const MyClass& Obj) const
            {
                  if(Obj.GetID() == m_nID)
                        return true;
                  return false;
            }
            bool operator<(const MyClass& Obj) const
            {
                  if(m_nID < Obj.GetID())
                        return true;
                  return false;
            }
      private:
            int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
      MyClass Obj1(1);
      MyClass Obj2(2);
      MyClass Obj3(3);

      map<long,MyClass> mapHandleToObj;

      mapHandleToObj[1L] = Obj1;
      mapHandleToObj[10L] = Obj1;
      mapHandleToObj[100L] = Obj1;
      mapHandleToObj[200L] = Obj2;
      mapHandleToObj[300L] = Obj3;
```

```
	long lDeletedKey = 0L;
	MyClass DeletedObj(0);

	//key suchen:
	map<long,MyClass>::iterator it = mapHandleToObj.find(10L);
	if(it != mapHandleToObj.end())
	{
		lDeletedKey = (*it).first;
		DeletedObj = (*it).second;
				//nicht DeletedObj = mapHandleToObj[10L]!!!
		mapHandleToObj.erase(it);
				//nicht mapHandleToObj.erase(*it)!!!
	}

	return 0;
}
```

21.7 Beispiele für die Verwendung der Container

21.7.1 list: Auflistung von Objekten mit möglichen Mehrfachvorkommnissen

```
class MyClass
{
      public:
            explicit MyClass(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            bool operator==(const MyClass& Obj) const
            {
                  if(Obj.GetID() == m_nID)
                        return true;
                  return false;
            }
            bool operator<(const MyClass& Obj) const
            {
                  if(m_nID < Obj.GetID())
                        return true;
                  return false;
            }
      private:
            int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <list>
using namespace std;

int main()
{
      MyClass    Obj1(1);
      MyClass    Obj2(2);
      MyClass    Obj3(3);

      list<MyClass> listObjs;
      listObjs.push_front(Obj3); //Element an den Anfang einfügen
      listObjs.push_front(Obj2); //Element an den Anfang einfügen
      listObjs.push_back(Obj1);  //Element ans Ende anhängen
      listObjs.push_back(Obj2);  //Element ans Ende anhängen

      listObjs.sort();           //Liste sortieren
      listObjs.unique();         //Benachbarte Mehrfach-
                                 //vorkommnisse eliminieren
      MyClass Obj(0);
      Obj = listObjs.front();    //erstes Element lesen
      Obj = listObjs.back();     //letztes Element lesen

      listObjs.pop_front();      //erstes Element löschen
      listObjs.pop_back();       //letztes Element löschen

      listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
      listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
      listObjs.push_back(Obj1);  //-> Mehrfachvorkommnis von Obj1
```

```
    int nFirstDeletedID = 0;
    list<MyClass>::iterator it;
    it = find(listObjs.begin(),listObjs.end(),Obj1);
    if(it != listObjs.end())   //erstes Vorkommnis von Obj1
    {
        nFirstDeletedID = (*it).GetID();
        listObjs.erase(it);   //dieses Vorkommnis von Obj1 löschen
    }

    listObjs.remove(Obj1);     //alle Vorkommnisse von Obj1 löschen

    for(it = listObjs.begin();it != listObjs.end();++it)
    {
        if((*it).GetID() == 1)
            break;
    }

    return 0;
}
```

Besonderheiten bei `list`:

- **Sortierung (`sort()`) und Eindeutigkeit (`unique()`)**
  Eine `list` kann nur von Mehrfachvorkommnissen von Objekten befreit werden (`unique()`), wenn sie sortiert (`sort()`) vorliegt:
  ```
  listObjs.sort();
  listObjs.unique();
  ```

- **Keine Member-Funktion `find()`**
  Zum Suchen ist der allgemeine Algorithmus `find()` anzuwenden, wobei zu beachten ist, dass hierbei nur das erste Vorkommnis in der `list` gefunden wird:

  ```
  it = find(listObjs.begin(),listObjs.end(),Obj1);
  ```

  Man kann natürlich die `list` vorher sortieren und unique machen. In dem Fall ist es jedoch effektiver, von vornherein eine `set` zu benutzen, da dort bereits beim Einfügen sortiert wird, was schneller geht. Außerdem ist die `set` a priori unique.

- **Nur `remove()` löscht sicher alle Vorkommnisse eines Wertes**
  Wenn man alle Vorkommnisse eines Wertes löschen will, ist **nicht** `erase()` zu verwenden, sondern **`remove()`**.

- **Die Reihenfolge der Objekte in der `list` ist fix**
  Nur Algorithmen wie `sort()` können etwas an der Reihenfolge der Objekte in der `list` ändern.

21.7.2 set: Aufsteigend sortierte Menge von Objekten (unique)

```
class MyClass
{
	public:
		explicit MyClass(int nID = 0) : m_nID(nID) {}
		int GetID() const { return m_nID; }
		bool operator==(const MyClass& Obj) const
		{
			if(Obj.GetID() == m_nID)
				return true;
			return false;
		}
		bool operator<(const MyClass& Obj) const
		{
			if(m_nID < Obj.GetID())
				return true;
			return false;
		}
	private:
		int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
	MyClass	Obj1(1);
	MyClass	Obj2(2);
	MyClass	Obj3(3);

	set<MyClass> setObjs;
	setObjs.insert(Obj3); //Element einfügen
	setObjs.insert(Obj1); //Element einfügen
	setObjs.insert(Obj2); //Element einfügen

	set<MyClass>::iterator it;

	it = setObjs.find(Obj1);
	if(it != setObjs.end())
		setObjs.erase(it);	//Element löschen

	for(it = setObjs.begin();it != setObjs.end();++it)
	{
		if((*it).GetID() == 1)
			break;
	}
	return 0;
}
```

21.7.3 map: Zuordnung von Objekten zu eindeutigen Handles

```
class MyClass
{
      public:
            explicit MyClass(int nID = 0) : m_nID(nID) {}
            int GetID() const { return m_nID; }
            bool operator==(const MyClass& Obj) const
            {
                  if(Obj.GetID() == m_nID)
                        return true;
                  return false;
            }
            bool operator<(const MyClass& Obj) const
            {
                  if(m_nID < Obj.GetID())
                        return true;
                  return false;
            }
      private:
            int m_nID;
};

#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
      MyClass    Obj1(1);
      MyClass    Obj2(2);
      MyClass    Obj3(3);

      map<long,MyClass> mapHandleToObjs;

      //Elemente einfügen:
      mapHandleToObjs[11L] = Obj3;     //Handle 11 -> Obj3
      mapHandleToObjs[102L] = Obj1;    //Handle 102 -> Obj1
      mapHandleToObjs[1026L] = Obj2;   //Handle 1026 -> Obj2

      map<long,MyClass>::iterator it;

      it = mapHandleToObjs.find(102L);
      if(it != mapHandleToObjs.end())
      {
            long lkey = (*it).first;         //key
            MyClass Obj = (*it).second;      //value
            mapHandleToObjs.erase(it);       //Element löschen
      }
```

```
      for(it = mapHandleToObjs.begin();
            it != mapHandleToObjs.end();
            ++it)
      {
            if((*it).second.GetID() == 1)
                  break;
      }

      return 0;
}
```

Besonderheiten bei map:

- **Indizierter Zugriff führt internes `find()` aus**

  → langsam → keine Alternative zu `vector`!

- **Vor dem Lesen muss man nach dem key-Eintrag suchen**

  Wenn man vor dem Lesen nicht sucht, wird **bei nichtvorhandenem key ein neuer Eintrag generiert** (Default-Konstruktor) und dessen Wert zurückgeliefert, was katastrophale Folgen haben kann.

  Nach dem Suchen sollte man keinen indizierten Zugriff ([]-Operator) mehr durchführen, da dieser wieder ein internes `find()` durchführt:

```
MyClass Obj;
it = mapHandleToObjs.find(102L);
if(it != mapHandleToObjs.end())
      Obj = (*it).second; //nicht Obj = mapIntAndObjs[102L]!!!
```

21.7.4 map: Mehrdimensionaler Schlüssel

Um einen mehrdimensionalen Schlüssel zu verwenden, definiert man am besten eine eigene Klasse, die die notwendigen Operatoren für die `map` enthält:

Beispiel:

```
struct MyTriple
{
      unsigned short x;
      unsigned short y;
      unsigned short z;

      //Initializing (c'tor):
      MyTriple() : x(0),y(0),z(0) {}
      //Assignment:
      MyTriple& operator=(const MyTriple& Obj)
      {
            if(this == &Obj)
                  return *this;
            x = Obj.x;
            y = Obj.y;
            z = Obj.z;
            return *this;
      }
};

//Global operators:

inline bool operator==(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs.x != rhs.x)
            return false;
      if(lhs.y != rhs.y)
            return false;
      if(lhs.z != rhs.z)
            return false;
      return true;
}
inline bool operator!=(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs == rhs)
            return false;
      return true;
}
```

```
inline bool operator<(const MyTriple& lhs,const MyTriple& rhs)
{
      if(lhs.x < rhs.x)
            return true;
      if(lhs.x > rhs.x)
            return false;
      if(lhs.y < rhs.y)
            return true;
      if(lhs.y > rhs.y)
            return false;
      if(lhs.z < rhs.z)
            return true;
      return false;
}

#include <stdio.h>
#include <map>
using namespace std;
int main()
{
      map<MyTriple,unsigned long>     mapTriple2Long;

      //----- Store: ------

      MyTriple t;
      t.x = 5;
      t.y = 2;
      t.z = 4;

      unsigned long lNumber = 0x44d533a1L;

      mapTriple2Long[t] = lNumber;

      //----- Search (Read): ------

      t.x = 5;
      t.y = 2;
      t.z = 4;

      lNumber = 0L;

      map<MyTriple,unsigned long>::const_iterator it
                                                = mapTriple2Long.find(t);
      if(it != mapTriple2Long.end())
            lNumber = (*it).second;

      return 0;
}
```

21.7.5 vector: Schneller indizierter Zugriff

Der indizierte Zugriff bei `vector` ist effektiver als bei einer `map`, da es sich beim **Index** nicht um ein beliebiges key-Feld handelt, sondern um einen **Integer**, der direkt zur Berechnung der Adresse des Wertes benutzt wird. `vector` ist einem normalen Array vorzuziehen, wenn man die Anzahl der Elemente erst zur Laufzeit kennt (vollkommen dynamische Länge, Heap-Management).

Beispiel:

```
#include <stdio.h>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;

void Fill(vector& vectFill)
{
      vectFill.clear();
      vectFill.push_back("1");
      vectFill.push_back("2");
      vectFill.push_back("3");
}
int main()
{
      //Anzahl der Elemente bekannt:
      vector<string> vectEntries(2);
      vectEntries[0] = "Hello";
      vectEntries[1] = " World!";
      string szTest("");
      szTest = vectEntries[0];
      szTest = vectEntries[1];

      //Anzahl der Elemente unbekannt:
      Fill(vectEntries);
      for(int i = 0;i < vectEntries.size();++i)
            szTest = vectEntries[i];

      return 0;
}
```

Besonderheit beim `vector`:

`push_back()` ist immer dem indizierten Schreibzugriff `[]` vorzuziehen

Der indizierte Schreibzugriff erfordert zuvor das Allokieren von Speicher. Wenn also nicht dem Konstruktor als Parameter die Anzahl der Elemente mitgeteilt wurde oder nicht schon durch `push_back()` oder `push_front()` der `vector` auf eine entsprechende Größe gebracht wurde, führt der indizierte Schreibzugriff in einen nicht definierten Speicherbereich (→ '`access violation`').

21.7.6 pair und make_pair(): Wertepaare abspeichern

Möchte man Wertepaare abspeichern, dann kann **`pair`** die richtige Unterstützung bieten. **`make_pair()`** erzeugt effektiv die passenden Wertepaare zum Einfügen.

Beispiel:

```
#include <stdio.h>
#include <algorithm>
#include <list>
using namespace std;

int main()
{
      list<pair<int,int> > listPoints;

      int x = 11;
      int y = 345;

      listPoints.push_back(make_pair(x,y));

      list<pair<int,int> >::iterator it;
      for(it = listPoints.begin();it != listPoints.end();++it)
      {
            int xTest = (*it).first;
            int yTest = (*it).second;
      }
      return 0;
}
```

21.8 hash_map

21.8.1 hash_map für Nutzer von Visual C++

Da nicht alle Compiler, wie z.B. Microsoft Visual C++, eine STL mit **`hash_map`** anbieten, muss man gegebenenfalls eine STL-Implementierung eines Drittanbieters benutzen (z.B. STLport).

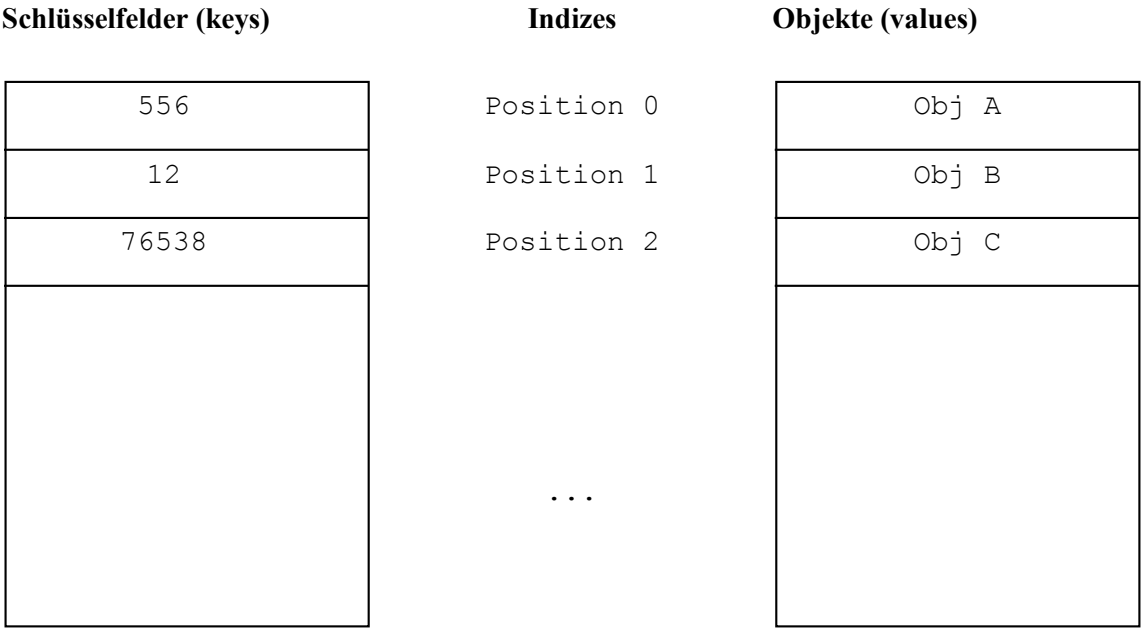
21.8.2 Prinzip von hash_map

`hash_map` ist eine Tabelle, in der Objekte abhängig von ihrem **key** abgespeichert werden. Der key bestimmt die Position eines Objektes. Die Funktionsweise einer `hash_map` lässt sich am besten durch einen Vergleich mit einer (normalen) `map` erklären:

`map`:

Liest man über einen key ein Objekt aus, dann wird zunächst in einer Schlüsselfeld-Liste das Schlüsselfeld gesucht (**internes `find()`**), das diesen key enthält. Hat man das Feld gefunden, dann entspricht seine Position (Index) der Position des Objektes in einer weiteren Liste, wo das Objekt dann ausgelesen wird. Das **interne `find()`** macht die Sache insgesamt **langsam**.

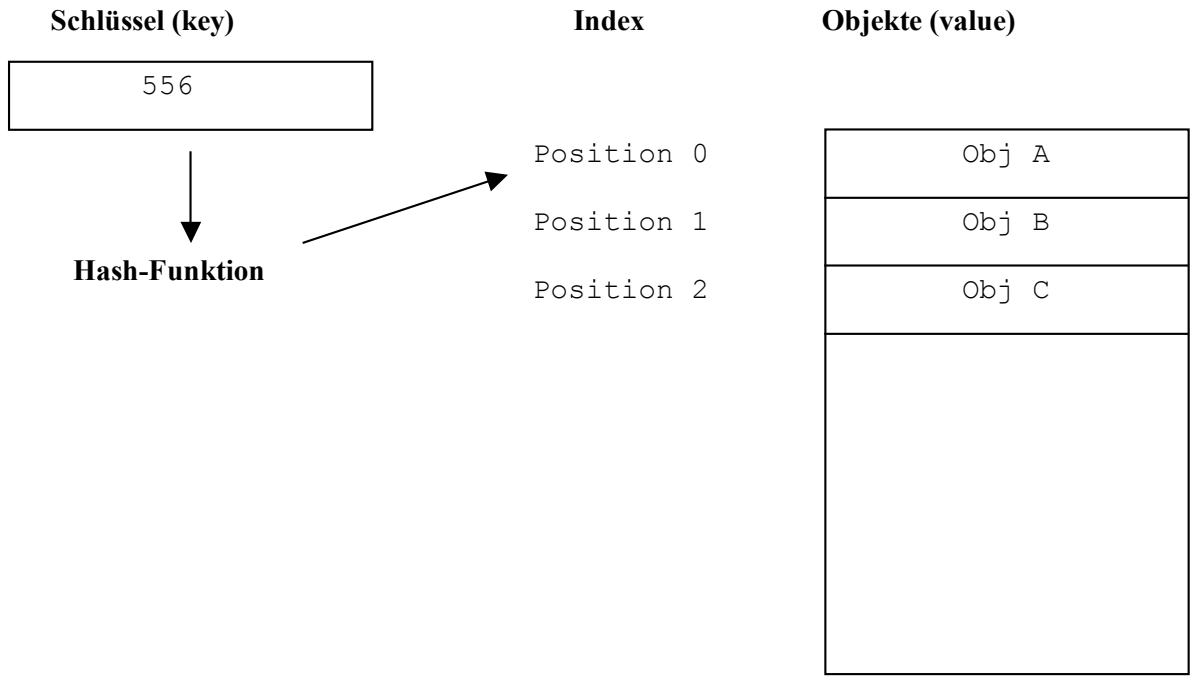
Prinzip der `map`:



hash_map:

Im Unterschied zu einer (normalen) `map` muss man **nicht erst ein Schlüsselfeld suchen**, dessen Position dann den Index für den Objekt-Zugriff liefert, sondern man **errechnet sich den Index direkt über eine sogenannte Hash-Funktion aus dem key** (hash = zerhacken).

Prinzip der `hash_map`:



Die **Hash-Funktion** (`hash<key>`) **sollte einen eindeutigen Index liefern**, d.h. zwei unterschiedliche keys sollten nicht zum gleichen Index (`equal_to<key>`) führen. Ist dies nicht der Fall, wird die Berechnung noch mit einer lokalen Suche gekoppelt. Man hat **keine lückenlos aufgefüllte Liste** wie bei der `map`. Wenn die `hash_map` zu voll wird (ab ca. 75% Füllung), dann wird das Verfahren langsam.

Die Syntax sieht wie folgt aus:

```
hash_map<key,value,hash<key>,equal_to<key> > hmapKeyToValue;
```

Das **`hash<key>`**-Template ist die Hash-Funktion. Das **`equal_to<key>`**-Template sorgt für die Eindeutigkeit der `hash_map`. Im Gegensatz zur `map` kann die `hash_map` **nicht sortieren**, da sie keinen "ist-kleiner"-Vergleich, wie z. B. die `map` mit `less<key>`, durchführt. Man findet lediglich schnell den Eintrag (`value`) zu einem Schlüssel (`key`).

Vorteil:

- Eine `hash_map` ermöglicht einen **schnellen** Objekt-Zugriff, wenn die Hash-Funktion effektiv ist.

Nachteile:

- **Performance-Verlust**, wenn die `hash_map` **zu voll** ist (ab ca. 75%)
  → Abhilfe durch automatische Vergrößerung

- Im Gegensatz zur `map` ist **keine Sortierung** der Schlüssel möglich

- **hash-Funktion** muss **für eigene Schlüssel-Klassen** bereitgestellt werden
  → Benutzt man eine selbst geschriebene Klasse als Schlüssel, dann muss man auch die hash-Funktion selbst programmieren, was nicht zu unterschätzen ist.

21.8.3 Nutzung von hash_map der STL

Es gibt **4 Arten**, die `hash_map` zu nutzen:

- **Keine eigene `hash`- oder `equal_to`-Funktion zur Verfügung stellen**:

  ```
  hash_map<key,value> hmapKeyToValue
  ```

  Diese Methode ist auf jeden Fall zu empfehlen, wenn **`key`** ein **Standard-Typ** ist, denn dann bietet die STL sowohl die `hash`- als auch die `equal_to`-Funktion an.

  Beispiel:

```
int main()
{
      hash_map<const char*,int> hmapMonthToDays;

      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf("September: %d days\n",
             hmapMonthToDays["September"]);

      return 0;
}
```

- **Eigene `hash`-Funktion zur Verfügung stellen**:

```
hash_map<key,value,MyHash> hmapKeyToValue
```

Diese Methode ist auf jeden Fall erforderlich, wenn man eine selbst geschriebene Klasse als `key` benutzt!

Beispiel:

Der `key` wird **aus den ersten 4 Zeichen eines Strings** ermittelt. Man verwendet die 8-Bit-ASCII-Codes der Zeichen zum Errechnen eines Indexes:

"ABCD…" → $0xD_1D_0C_1C_0B_1B_0A_1A_0$

```
struct MyHash
{
      size_t operator()(const char* key) const
      {
            size_t ret = 0;
            for(int i = 0;i < 4;++i)
            {
                  if(*key == 0)
                        break;
                  ret += (1 << (i * 8)) * (*key++);
            }
            return ret;
      }
};

int main()
{
      hash_map<const char*,int> hmapMonthToDays;

      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf("September: %d days\n",
             hmapMonthToDays["September"]);

      return 0;
}
```

- **Eigene `equal_to`-Funktion zur Verfügung stellen**:

```
hash_map<key,value,hash<key>,MyEqualTo> hmapKeyToValue
```

Beispiel:

Die Schlüssel sind Strings (`char`-Arrays), die zwar von der STL gehashed, aber über eine eigene Funktion auf Gleichheit überprüft werden.

```
struct MyEqualTo
{
      bool operator()(const char* s1,const char* s2) const
      {
            return strcmp(s1,s2) == 0;
      }
};

int main()
{
      hash_map<const char*,int,hash<const char*>,MyEqualTo>
                                                hmapMonthToDays;
      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf("September: %d days\n",
             hmapMonthToDays["September"]);

      return 0;
}
```

- **Eigene `hash`- und `equal_to`-Funktion zur Verfügung stellen**:

```
hash_map<key,value,MyHash,MyEqualTo> hmapKeyToValue
```

Beispiel:

Der `key` wird sowohl selbst gehashed (**aus den ersten 4 Zeichen**), als auch auf Gleichheit überprüft.

```
struct MyEqualTo
{
      bool operator()(const char* s1,const char* s2) const
      {
            return strcmp(s1,s2) == 0;
      }
};
struct MyHash
{
      size_t operator()(const char* key) const
      {
            size_t ret = 0;
            for(int i = 0;i < 4;++i)
            {
                  if(*key == 0)
                        break;
                  ret += (1 << (i * 8)) * (*key++);
            }
            return ret;
      }
};
int main()
{
      hash_map<const char*,int,MyHash,MyEqualTo>
                                                hmapMonthToDays;

      hmapMonthToDays["January"] = 31;
      hmapMonthToDays["February"] = 28;
      hmapMonthToDays["March"] = 31;
      hmapMonthToDays["April"] = 30;
      hmapMonthToDays["May"] = 31;
      hmapMonthToDays["June"] = 30;
      hmapMonthToDays["July"] = 31;
      hmapMonthToDays["August"] = 31;
      hmapMonthToDays["September"] = 30;
      hmapMonthToDays["October"] = 31;
      hmapMonthToDays["November"] = 30;
      hmapMonthToDays["December"] = 31;

      printf("September: %d days\n",
             hmapMonthToDays["September"]);

      return 0;
}
```

21.9 Lokalisierung mit der STL (streams und locales)

Die STL bietet eine C++-Schnittstelle zur Lokalisierung an. Das bedeutet, dass sprach- und länderspezifische Merkmale (z.B. Datum und Preise/Währungen) automatisch für verschiedene Sprach/Land-Kombinationen (locale) richtig formatiert werden.

Hierbei gibt es 3 wesentliche Begriffe, die man kennen sollte:

facet:

Eine bestimmte sprach- und landesspezifische Sache.
Beispiele:

Eingeben der Uhrzeit (`time_get`)
Auslesen und Darstellen der Uhrzeit (`time_put`)

category:

Zusammenfassung von facets, die thematisch zusammengehören.
Beispiele:

`all`	`(LC_ALL)`	→	Alle Kategorien
`collate`	`(LC_COLLATE)`	→	Stringvergleich
`ctype`	`(LC_CTYPE)`	→	Zeichen-Handling
`monetary`	`(LC_MONETARY)`	→	Preisformatierung
`numeric`	`(LC_NUMERIC)`	→	Dezimalpunktformatierung
`time`	`(LC_TIME)`	→	Zeitformatierung
`messages`	`(LC_MESSAGES)`	→	Nachrichtenformatierung

locale:

Zusammenfassung aller facets einer Sprach/Land-Kombination.
Beispiele:

`de_DE@euro`	→	Deutsch/Deutschland + Euro
`en_US`	→	Englisch/USA

Das Prinzip der STL-Lokalisierung sieht so aus, dass die Operationen, die auf einen **Stream** angewendet werden, durch ein **`locale`**-Objekt beeinflusst werden (**imbuing** = durchdringen). Das bedeutet also, dass man auf jeden Fall immer einen **stream** und ein **`locale`-Objekt** benötigt, um diese Lokalisierung auszuführen. Da man bei der Programmierung von Anwendungen mit graphischer Benutzeroberfläche nicht die **iostreams** `cout` und `cin` benutzen wird, kommt bei solchen Anwendungen hauptsächlich der **stringstream** zum Einsatz.

Für einen STL-Hersteller ist es möglich, die STL so zu implementieren, dass sie lediglich die C-Funktion `localeconv()` benutzt, um sich beim Betriebssystem über die Parameter der verschiedenen locales zu informieren. In dem Fall beinhaltet die STL keine eigene locale-Implementierung.

Vorteile der Nutzung von `locale`-Objekten:

- Sprach- und landesspezifische Formatierungen geschehen automatisch.
- Man muss die Implementierung für die Lokalisierung nicht selbst pflegen.
- Wenn die STL-Implementierung `localeconv()` benutzt, um sich beim Betriebssystem über die Parameter der verschiedenen locales zu informieren, dann bleibt die gebaute Software zeitlos.

Nachteile:

- Wenn die STL-Implementierung lediglich `localeconv()` benutzt, also keine eigene locale-Implementierung hat, dann...
  - ...ist in der Regel auch der String-Parameter des Konstruktors von `locale()` betriebssystemabhängig.
  - ...sind die Formatierungen systemabhängig und können je nach System unterschiedlich aussehen.

- Wenn die STL-Implementierung **nicht** `localeconv()` benutzt, sondern eine eigene Implementierung der kompletten Lokalisierung beinhaltet, dann...
  - ...ist die gebaute Software nicht zeitlos, wohl aber der eigene Quell-Code, der lediglich mit einer neuen STL-Implementierung neu gebaut werden muss (bspw. nach der Umstellung einer Währung, wie z.B. "DM" → "EURO")

Beispiel für die locales "**de_DE**" und "**en_US**":

```
#include <stdio.h>
#include <time.h>

#include <locale> //STL-Lokalisierung
#include <sstream> //string streams
using namespace std;

int main()
{
     printf("Locales:\r\n");

     //Default locale:

     locale localeDefault;
     printf("   Default locale: '%s'\r\n",localeDefault.name());


     //Die category "time" durch diejenige in "de_DE" ersetzen:

     locale localeCustomized(   localeDefault,
                          locale("de_DE"),
                          locale::time);
     printf("   Customized locale: '%s'\r\n",
            localeCustomized.name());
```

```
//Facet 'time_put' der category 'time'
//('struct tm' buffer wird gebraucht):

printf("time_put:\r\n");

timeSystem = time(NULL);
ostringstream ossTime;
ossTime.imbue(locale("de_DE"));
const time_put<char>& tp
          = use_facet<time_put<char> >(ossTime.getloc());

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'x');
ossTime << " ";

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'A');
ossTime << " ";

tp.put(ossTime.rdbuf(),
       ossTime,
       ossTime.fill(),
       localtime(&timeSystem),
       'B');
string strTime(ossTime.str());
printf("   Time in '%s': [%s]\r\n",
       ossTime.getloc().name().c_str(),
       strTime.c_str());

//Facet 'ctype' der category 'ctype' (Zeichen-Handling):

printf("ctype:\r\n");
locale locCType("de_DE");
const ctype<char>& ct = use_facet<ctype<char> >(locCType);
ct.tolower(const_cast<char*>(
           strTime.c_str()),strTime.c_str()+strTime.length());
printf("   Time in '%s' - 'lowercase': [%s]\r\n",
       locCType.name().c_str(),
       strTime.c_str());

//Facet 'money_put' der category 'monetary' (Preise/Währungen):

printf("money_put:\r\n");

const string strPriceText("720000000000");
ostringstream ossPriceText;
ossPriceText.imbue(locale("en_US"));
bool bInternational = false;
ossPriceText.setf(ios_base::showbase); //basefield (Währung)
const money_put<char>& mp
     = use_facet<money_put<char> >(ossPriceText.getloc());
```

```
mp.put(ossPriceText.rdbuf(),
       bInternational,
       ossPriceText,
       ' ',
       strPriceText);
string strPrice(ossPriceText.str());
printf("   Price in '%s': [%s]\r\n",
       ossPriceText.getloc().name().c_str(),
       strPrice.c_str());

//Facet 'num_put' der category 'numeric':

printf("num_put:\r\n");

locale locNum("de_DE");
const num_put<char>& np = use_facet<num_put<char> >(locNum);

double dNum = 3827298.77;
ostringstream ossFloatNum;
ossFloatNum.imbue(locNum);
ossFloatNum.setf(ios_base::fixed | ios_base::internal);
ossFloatNum.precision(2); //Anzahl der Stellen nach dem Komma
np.put(ossFloatNum.rdbuf(),ossFloatNum,' ',dNum);
string strNum(ossFloatNum.str());
printf("   Floatingpoint in '%s': [%s]\r\n",
       ossFloatNum.getloc().name().c_str(),
       strNum.c_str());

ostringstream ossScientificNum;
ossScientificNum.imbue(locNum);
ossScientificNum.setf(
                ios_base::scientific | ios_base::floatfield);
ossScientificNum.precision(6); //Anzahl d.Stellen nach d.Komma
np.put(ossScientificNum.rdbuf(),ossScientificNum,' ',dNum);
strNum = ossScientificNum.str();
printf("   Scientific in '%s': [%s]\r\n",
       ossScientificNum.getloc().name().c_str(),
       strNum.c_str());

bool bFlag = false;
ostringstream ossBooleanNum;
ossBooleanNum.imbue(locNum);
ossBooleanNum.setf(ios_base::boolalpha); //bool: "true"/"false"
np.put(ossBooleanNum.rdbuf(),ossBooleanNum,' ',bFlag);
strNum = ossBooleanNum.str();
printf("   Boolean in '%s': [%s]\r\n",
       ossBooleanNum.getloc().name().c_str(),
       strNum.c_str());
```

```
      unsigned long dwHexNum = 0x12F5418E;
      ostringstream ossHexNum;
      ossHexNum.imbue(locNum);
      ossHexNum.setf(ios_base::uppercase); //uppercase, 'f' -> 'F'
      ossHexNum.setf(
            ios_base::hex,ios_base::basefield); //basefield in HEX
      ossHexNum.width(15); //Länge -> 15 Stellen
      np.put(ossHexNum.rdbuf(),ossHexNum,'-',dwHexNum); //Füllz.='-'
      strNum = ossHexNum.str();
      printf("   HEX number in '%s': [%s]\r\n",
             ossHexNum.getloc().name().c_str(),
             strNum.c_str());

      return 0;
}
```

Beispiel für die Implementierung einer "besser lesbaren" Zahlendarstellung:

```
#include <locale>
#include <sstream>
using namespace std;

string GetLocalizedStyleNumber(
      const double& dNum,unsigned long dwDigitsAfterDecPoint=0)
{
      locale locNum("en_US");
      const num_put<char>& np = use_facet<num_put<char> >(locNum);
      ostringstream ossFloatNum;
      ossFloatNum.imbue(locNum);
      ossFloatNum.setf(ios_base::fixed | ios_base::internal);
      ossFloatNum.precision(dwDigitsAfterDecPoint);

      np.put(ossFloatNum.rdbuf(),ossFloatNum,' ',dNum);
      return ossFloatNum.str();
}

int main()
{
      string strNum = GetLocalizedStyleNumber(10000000,2);
      printf("Num = %s\r\n",strNum.c_str());
      return 0;
}
```

Hierdurch wird die Zahl 10000000 in der Form

```
      10,000,000.00
```

dargestellt, was wesentlich besser lesbar ist.

http://www.springer.com/978-3-540-01058-6

Notizen zu C++
Thömmes, P.
2004, XIII, 312 S., Hardcover
ISBN: 978-3-540-01058-6