1. SVN und STL

AlDaBi Praktikum

David Weese
WS 2010/11

Inhalt

- Einführung
- Subversion (SVN)
- Standard Template Library (STL)

EINFÜHRUNG

Praktikum

Das Praktikum umfasst:

- *diese* Praktikumsvorlesung
  - alle 2 Wochen, freitags 10-12 c.t.
- Praktikumsaufgaben
  - stehen auf jedem zweiten Übungszettel
  - **Abgabe**: montags, 15 Uhr in den Wochen der P-VL
  - Tutorien: dienstags, 12-14 und 14-16 in den Wochen zwischen den P-VLs
  - 75% der Praktikumsaufgaben lösen
  - Code-Reviews in den Wochen von Review und Examination

- P-VL Themen:
  - Subversion, C++ Standard Template Library
  - Debugging, Profiling, Zeitmessung
  - Rechnerarchitektur, Parallele Programmierung
  - OpenMP
  - Dateizugriff
  - SeqAn, TBB, OpenMPI, Boost

Abgabe der Lösungen

- Abgabe per Subversion:
  - jede Gruppe erhält ein SVN-Verzeichnis https://svn.mi.fu-berlin.de/agbio/aldabi/ws10/gruppeX
  - die Lösung jeder Aufgabe muss als Datei mit dem Titel **aufgabeY.cpp** (Kleinschreibung beachten, keine Unterordner!) bis zum Abgabetermin eingecheckt werden
  - ihr Programm muss u.a. auf einem Linux-Poolrechner kompilieren mit
    g++ -pedantic -Wall -ansi -o aufgabeY aufgabeY.cpp
  - beachten Sie genau die Hinweise zum jeweiligen Ein- und Ausgabeformat auf den Aufgabenblättern und auf den Wiki-Seiten
- Erstmaliges Auschecken:
  - Eintragen als Gruppe unter https://www.mi.fu-berlin.de/w/ABI/AlDaBiWS10Groups
  - svn checkout https://svn.mi.fu-berlin.de/agbio/aldabi/ws10/gruppeX
- Lösung abgeben:
  - cd gruppeX
  - svn add aufgabeY.cpp
  - svn commit aufgabeY.cpp

Literatur

- C++ für Einsteiger
  - *„Der C++ Programmierer“*, Breymann, Hanser Verlag
- C++ für Fortgeschrittene
  - *„C++ Templates“*, Vandevoorde und Josuttis, Addison-Wesley
  - *„Essential C++“*, Lippman, Addison-Wesley
  - *„Moderne C++ Programmierung”*, Schneeweiß, Springer
- STL
  - *„Effective STL“*, Meyers, Addison-Wesley
  - *„Generic Programming and the STL“*, Austern, Addison-Wesley
- Algorithmen
  - *„Algorithmen in C++“,* Sedgewick, Pearson Studium
  - *„Algorithmen und Datenstrukturen“*, Pomberger und Dobler, Pearson Studium

Web

- C++ für Einsteiger
  - Häufige Fragen
    - http://www.parashift.com/c++-faq-lite/
  - Tutorials
    - http://code.google.com/intl/de-DE/edu/languages/cpp/basics/
    - http://www.inf.fu-berlin.de/en/groups/abi/lectures_past/SS2010/K_CPP_Advanced/
  - Referenzen
    - http://www.cppreference.com/
    - http://www.sgi.com/tech/stl/
    - http://gcc.gnu.org/onlinedocs/gcc-4.2.2/gcc/
    - http://msdn.microsoft.com/en-us/library/ms864102
- C++ für Fortgeschrittene
  - Diverse spezifische Artikel
    - http://www.gotw.ca/publications/index.htm
  - Forum für Entwickler
    - http://stackoverflow.com/

Subversion (SVN)

Kollaboration ...

- Bei großen Projekten treten folgende Probleme auf:
  - Wie koordiniert und synchronisiert man Code zwischen verschiedenen Entwicklern?
  - Wie sichert man Code gegen versehentliches Löschen oder archiviert alte Stände?

- Per Mail? Dateien online ablegen? Auf Netzlaufwerken arbeiten?

- Besser: Source Code Management Tool (SCM) bspw.:
  - CVS
  - **Subversion**
  - Bazaar
  - Mercurial
  - Git

... mit einem SCM

- SCM besteht aus:
  - Repository
    - Code gespeichert auf zentralem Server oder verteilt bei Usern
  - Working Copy
    - Entwickler checkt eine Kopie des Repositories auf seinem Computer aus
  - Revision History
    - jede Änderung einer Datei wird gespeichert
    - kann auch wieder rückgängig gemacht werden
  - Conflict Handling
    - Was passiert wenn zwei Entwickler dieselbe Datei verändern?
    - ... in der selben Zeile?

Abläufe

- Typischer Entwicklungszyklus:
  - Entwickler checkt **Repository** initial aus
  - Editiert seine lokale **Working Copy**
  - Aktualisiert (*update*) seine **Working Copy** mit den Änderungen der anderen Entwickler aus dem **Repository**
  - Überträgt (*commit*) seine Änderungen in das **Repository**

- Diffs und Patches
  - Zwischen zwischen **Repository** und **Working Copies** werden nur Datei-Differenzen (*diffs*) hin- und hergeschickt
  - Differenzen (*patches*) zweier Dateien liefert das Tool diff
    - diff –u Datei1 Datei2 > MeinBugFix
  - Anwenden kann man diese mit patch
    - patch –p0 < MeinBugFix

Subversion - SVN

- Kommandos für Kommunikation mit dem Server:
  - svn checkout <URL>
  - svn update
  - svn commit
- Offline Kommandos:
  - svn add <Datei>
  - svn delete <Datei>
  - svn diff
  - svn rename
  - svn move
  - …
  - svn help <Kommando>

- TortioseSVN (Windows Client) - http://tortoisesvn.net/ und Tutorial
- SVN Homepage - http://subversion.tigris.org/ und Tutorial

STANDARD TEMPLATE LIBRARY (STL)

Warum die STL benutzen?

- Feld *f* von `int` der Größe *n* erzeugen:
  - malloc/free (klassisches C, stdlib):

```
int *f = (int *)malloc(sizeof(int) * n);
...
free(f);
```

  - new/delete (C++):

```
int *f = new int[n];
...
delete[] f;
```

  - vector (C++, STL):

```
std::vector<int> f(n);
```

Warum die STL benutzen? (II)

- An *f* eine 23 anfügen:
  - malloc/free (klassisches C, stdlib):

```
int * f_new = (int *)realloc(sizeof(int) * (n+1));
if (f_new != NULL) f = f_new;
f[n] = 23;
```

  - new/delete (C++):

```
int *f_new = new int[n+1];
for (int i = 0; i < n; ++i)
    f_new[i] = f[i];
delete[] f;
f = f_new;
f[n] = 23;
```

  - vector (C++, STL):

```
f.push_back(23);
```

Warum die STL benutzen? (III)

- Enthält Datenstrukturen (Container), die:
  - die (fehlerprovozierende) Verwaltung des Speichers abnehmen
    - Speicherfreigabe vergessen
    - delete statt delete[] verwendet
    - doppelte Speicherfreigabe
  - auch kompliziertere Klassen speichern können
    - malloc, realloc, free (C) konstruiert/destruiert Objekte nicht
  - **generisch** sind (templates)
    - Elementtyp ist template-Parameter
    - Vergleichsoperator von sortierten Containern lässt sich frei definieren
  - **effizient** implementiert sind
    - Container-Funktionen haben Laufzeitgarantien

Die Standard Library – Überblick

Überblick: Die Header-Dateien

C++ Library Header Files

<algorithm>	<iomanip>	<list>	<ostream>	<streambuf>
<bitset>	<ios>	<locale>	<queue>	<string>
<complex>	<iosfwd>	<map>	<set>	<typeinfo>
<deque>	<iostream>	<memory>	<sstream>	<utility>
<exception>	<istream>	<new>	<stack>	<valarray>
<fstream>	<iterator>	<numeric>	<stdexcept>	<vector>
<functional>	<limits>			

C Library Header Files

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<clocale>	<cstdarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<cstddef>		

STL = „Standard Template Library“

Programmbibliothek:

Datenstrukturen & Algorithmen.

„Generisch“:

- Formuliert über „Concepts“
- Hochgradige Typisierung (über Template-Argumente)
- Container und Algorithmen sind voneinander „entkoppelt“

Concepts - Motivation

Aufgabe: Programmiere eine Ausgabefunktion...

```
void printSequence(char * start, char * end)
{
 while (start != end)
 {
     cout << *start;
     ++start;
 }
}
```

... aber bitte so, dass sie nicht nur für `char *` funktioniert!

Concepts - Motivation (II)

Lösung: Templates

```
template <typename T>
void printSequence(T start, T end)
{
 while (start != end)
 {
     cout << *start;
     ++start;
 }
}
```

Frage: Für welche Typen **T** funktioniert das?

Concepts

Concept = Menge von Spezifikationen für Typen:

- Interface (Funktionen, Operatoren):
  z.B.: »muss **`operator*`** definieren.«
- Vorgaben für Ressourcenverbrauch:
  z.B.: »**`operator*`** muss O(1) Zugriffszeit haben.«
- Semantik:
  z.B.: »nach Anwendung von **`operator++`** zeigt der Iterator auf das *nächste* Element im Container.«

STL Varianten

- Spezifikation lässt Implementierungsvarianten zu.
  z.B.: **`allocator`** der SGI-STL implementiert für kleine Speicherblöcke einen Pool Allocator.

- Gewisse nicht-standard Container & Algorithmen sind weit verbreitet.
  z.B.:
  - hash_map
  - hash_multimap
  - hash_set
  - rope

STL - Überblick

Header (aus dem Standard) der STL:

- Stringklasse und Container:
  `<string>`, `<vector>`, `<deque>`, `<list>`, `<map>`, `<set>`, `<queue>`, `<stack>`, `<bitset>`
- Iteratoren:
  `<iterator>`
- Algorithmen:
  `<algorithm>`
- Funktionsobjekte:
  `<functional>`
- Sonstiges:
  `<utility>`, `<sstream>`

Container kopieren Objekte

Problem: STL Container von Objekten kopieren diese Objekte häufig.

Lösungsansätze:

1. Schnelle Copy-C'toren schreiben
2. Reference-Counting
3. Container von Pointern

Quiz: Wo ist der Unterschied?

Frage: Was unterscheidet folgende Ausdrücke voneinander? (**`obj`** sei ein Containerobjekt)

```
if ( obj.size() == 0 ) ...
```

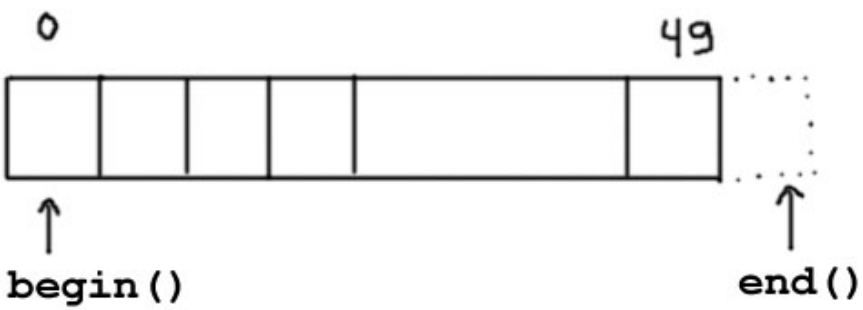
```
if ( obj.empty() ) ...
```

Iteratoren

Iteratoren = verallgemeinerter Pointer

```
vector<char> vec(50);
vector<char>::iterator anfang = vec.begin();
vector<char>::iterator ende = vec.end();

cout << ende - anfang;   //Ausgabe: 50
```



Warum Iteratoren benutzen?

- summiere alle Elemente von Feld *f*

```
int sum = 0;
for (int i = 0; i < n; ++i)
   sum += f[i];
```

```
        ...
.L8:
        movl    -4(%rbp), %eax
        cltq
        salq    $2, %rax
        addq    -16(%rbp), %rax
        movl    (%rax), %eax
        addl    %eax, -8(%rbp)
        addl    $1, -4(%rbp)
.L7:
        movl    -4(%rbp), %eax
        cmpl    -20(%rbp), %eax
        jl      .L8
        ...
```

- wird vom Compiler genauso übersetzt wie:

```
int sum = 0;
for (int i = 0; i < n; ++i)
   sum += *(f + i);
```

- effizienter ist (ohne Optimierung):

```
int sum = 0;
int *f_end = f + n;
for (int *p = f; p < f_end; ++p)
   sum += *p;
```

```
        ...
.L8:
        movq    -8(%rbp), %rax
        movl    (%rax), %eax
        addl    %eax, -20(%rbp)
        addq    $4, -8(%rbp)
.L7:
        movq    -8(%rbp), %rax
        cmpq    -16(%rbp), %rax
        jb      .L8
        ...
```

Warum Iteratoren benutzen? (II)

- Warum sorgen, wenn der Compiler optimiert?
  - die Optimierung funktioniert nur bis zu einer gewissen Komplexität
- Was passiert, wenn *f* eine verkettete Liste ist?
  - `f[i]` hat dann eine Laufzeit von O(i)
  - die ganze Schleife hat dann also $O(n^2)$ Laufzeit

```
std::list<int> f(n);
for (int i = 0; i < n; ++i)
    sum += f[i];
```

- Iteratoren sind optimal für sequentielle Zugriffe
  - Schleife mit Iteratoren hat O(n) Laufzeit

```
std::list<int> f(n);
std::list<int>::iterator i = f.begin();
std::list<int>::iterator i_end = f.end();
for (; i != i_end; ++i)
    sum += *i;
```

Iteratoren (II)

Mit Iteratoren kann man durch Container traversieren.

```
template<typename TIn, typename TOut>
inline TOut copy(TIn first, TIn last, TOut dest)
{
 while ( first != last )
 {
    *dest = *first;
    ++dest; ++first;
 }
 return (dest);
}
```

Quiz: Was läuft hier schief?

Frage: Wieso produziert dieses Programm einen Laufzeitfehler?

```
vector<int> vec1;
vector<int> vec2;

for (int i = 0; i < 10; ++i) vec1.push_back(i);

copy(vec1.begin(), vec1.end(), vec2.begin());
```

Quiz: Und hier?

Frage: Auch hier geht etwas schief. Was?

```
vector<int> vec;

for (int i = 0; i < 10; ++i) vec.push_back(i);

copy(  vec.begin(),
       vec.end(),
       back_insert_iterator< vector<int> >(vec));
```

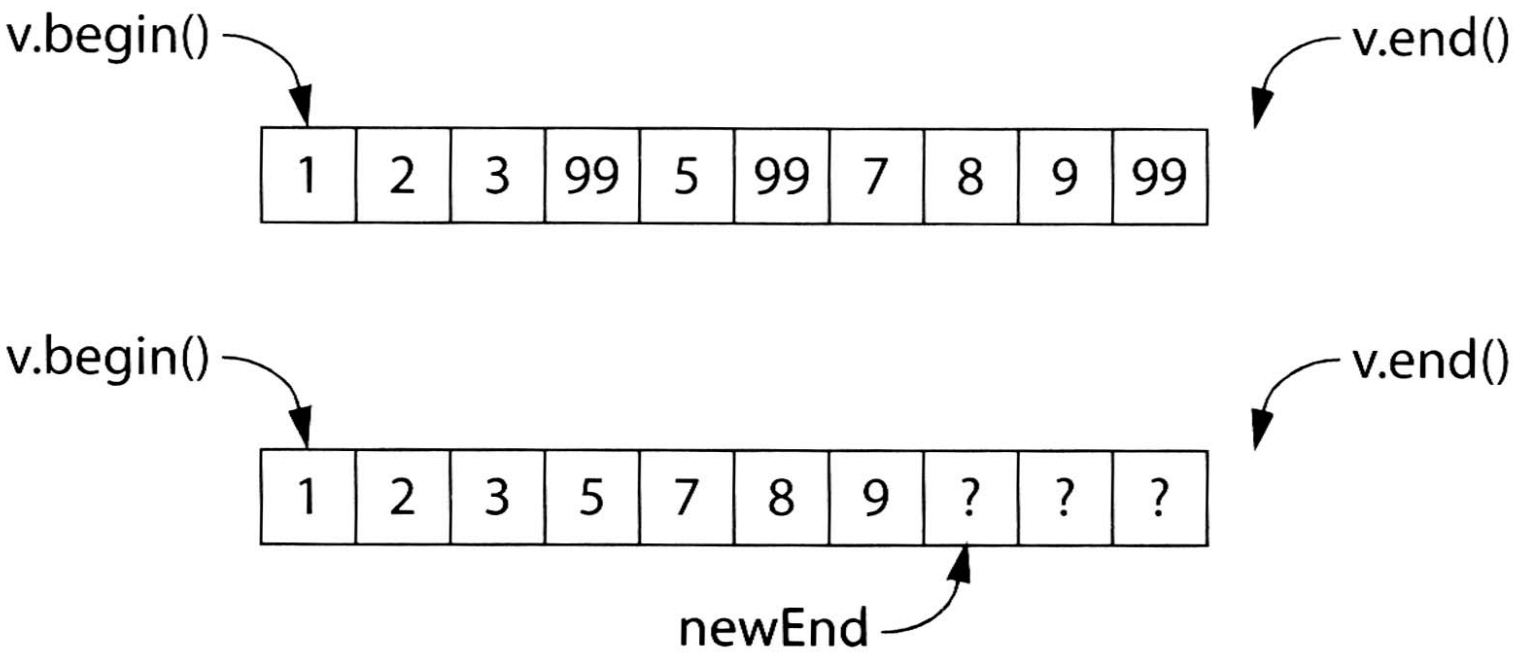
Algorithmen - Überblick

1. Durchlaufen und Setzen
   `for_each`, `transform`, `fill`, `generate`, …
2. Kopieren und Löschen
   `copy`, `replace`, `remove`, …
3. Suchen, Vergleichen und Zählen
   `find`, `search`, `equal`, `count`, `min`, …
4. Sortieren und Anordnen
   `sort`, `merge`, `random_shuffle`, …

Beispiel 1: `remove`

remove durchläuft eine Range, entfernt alle Elemente einer bestimmten Art daraus und gibt einen Iterator auf hinter das neue Ende zurück.

```
remove(v.begin(), v.end(), 99);
```



Beispiel 1: remove (II)

Problem:

```
vector<int> v;
v.reserve(10);
for (int i=1; i <= 10; ++i) v.push_back(i);
v[3] = v[5] = v[9] = 99;

remove(v.begin(), v.end(), 99);

cout << v.size();  //Ausgabe: 10!
```

Lösung:

```
v.erase( remove(v.begin(), v.end(), 99), v.end() );
```

Beispiel 2: `sort`

`sort` sortiert eine Range aufsteigend.

```
vector<int> vec(5);
vec[0] = vec[3] = 4;
vec[1] = 9;
vec[2] = 0;
vec[4] = 3;

sort( vec.begin(), vec.end() );
```

Beispiel 2: `sort` (II)

Frage: Was, wenn man absteigend sortieren will?

```
bool compareFunc(int a, int b)
{
 return (a > b);
}

sort( vec.begin(), vec.end(), compareFunc );
```

```
sort( vec.begin(), vec.end(), greater<int>() );
```

Funktionsobjekte (*functors*)

Ein Funktionsobjekt definiert **`operator ()`**:

```
template <typename T>
struct greater
{
 inline bool operator () ( T a, T b)
 {
     return (a > b);
 }
};
```

Funktionsobjekte (*functors*) (II)

1. Übliche Funktionen:
   `plus`, `multiplies`, `equal_to`, `logical_and`, …
2. Binders:
   `bind1st`, `bind2nd`, …
3. Adaptoren:
   `ptr_fun`, `mem_fun`, …