3. Generische Programmierung in C++

```
#include <iostream>
#include <string>
#include <map>

using namespace std ;

int main ( ) {
  string buf ;
  map < string , int > m ;
  while ( cin >> buf ) m [ buf ] ++ ;
  multimap < int , string > n ;
  for ( map < string , int > :: iterator p = m . begin ( ) ;
                p !=  m . end ( ) ; ++ p )
    n . insert ( multimap < int , string > :: value_type ( p -> second , p ->
    first ) ) ;
  for ( multimap < int , string > :: iterator p = n . begin ( ) ;
                p !=  n . end ( ) ; ++ p )
    cout <<  p -> first  << "\t" << p -> second << endl ;
}
```

```
$ wc < wc.cpp
1       "\t"
1       <iostream>
…..
6       string
10      (
10      )
10      p
11      ;
```

Input - & Output – Iteratoren

```
#include <string>
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
#include <sstream>

int main()
{
    std::istringstream s("I need your help!");

    std::vector<std::string> v( ( std::istream_iterator<std::string>(s) ) ,
                                std::istream_iterator<std::string>());
    std::copy(v.begin(), v.end(),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

```
I
need
your
help!
```

Scott Meyers „Effective STL“ (Item 6:) Be alert for C++'s most vexing parse

3. Generische Programmierung in C++

	vector	deque	list	set	multiset	map	multimap
interne Datenstruktur	dynamisches Array	Menge von Arrays	doppelt verkettete Liste	Binärbaum	Binärbaum	Binärbaum	Binärbaum
Elemente-Art	Wert	Wert	Wert	Wert	Wert	Wertepaar	Wertepaar
Duplikate erlaubt	ja	ja	ja	nein	ja	nein (Schlüssel)	ja (Schlüssel)
wahlfreier Zugriff	ja	ja	nein	nein	nein	über Schlüssel	nein
Iterator-Kategorie	Random-Access	Random-Access	Bidirectional	Bidirectional (Wert konstant)	Bidirectional (Wert konstant)	Bidirectional (Schlüssel konstant)	Bidirectional (Schlüssel konstant)
Suchen/Finden von Elementen	langsam	langsam	sehr langsam	sehr schnell	sehr schnell	sehr schnell für Schlüssel	sehr schnell für Schlüssel
Einfügen/Löschen schnell	am Ende	am Anfang und am Ende	überall konstant	überall logarithmisch	überall logarithmisch	überall logarithmisch	überall logarithmisch
Verweise werden ungültig	bei Reallokierung	potenziell immer	nein	nein	nein	nein	nein
Speicher wird freigegeben	nie	manchmal	immer	immer	immer	immer	immer
Speicherreservierung möglich	ja	nein	-	-	-	-	-

Container-Adaptoren

neben den (primären) Containern gibt es einige sog. Container-Adaptoren, es handelt sich dabei um Anpassungen der Container für spezielle Anwendungen

Queues `(#include <queue>)`

FIFO-Warteschlangen (auch mittels `list` instantiierbar)

```
namespace std {
  template < class T,
             class Container = deque<T>
  >
  class queue;
}
```

Priority Queues (#include <queue>)

Warteschlangen mit Prioritäten (auch mittels **deque** instantiierbar)

```
namespace std {
  template < class T, class Container = vector<T>,
    class Compare = less<typename Container::value_type>
  >
  class priority_queue;
}
```

Stacks (#include <stack>)

Kellerspeicher (auch mittels **list** und **vector** instantiierbar)

```
namespace std {
   template < class T, class Container = deque<T> >
   class stack;
 }
```

Strings `(#include <string>)` (vgl. z.B. Josuttis, Kapitel 10, S.357 ff.)

```
namespace std {
  template < class charT,
     class traits = char_traits<charT>
     class allocator = allocator <charT> >
  class basic_string;
  // noch nicht auf Zeichentyp festgelegt
  typedef basic_string<char>    string;  // ASCII
  typedef basic_string<wchar_t> wstring; // Unicode
}
```

mit Einführung von `string` wurde auch die `iostream`-Bibliothek erheblich überarbeitet, um mit strings zusammenarbeiten zu können, ohne dass sich die Nutzerschnittstelle wesentlich verändert hat, ggf. ist wichtig

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

Numerische Klassen

Komplexe Zahlen (#include <complex>)

```
namespace std {
  template<class T> class complex;
  template<> class complex<float>;
  template<> class complex<double>;
  template<> class complex<long double>;
}
```

komplexe Zahlen mit allem "drum und dran" (Arithmetik und transzendente Funktionen) für float, double, long double bereits vordefiniert !

Valarrays (#include <valarray>)

```
namespace std {
  template<class T> class valarray;
}
```

Vektoren und Matrizen für numerische Operationen mit gutem Zeitverhalten (keine temporären Zwischenergebnisse) und kompakter Notation (z.B. Ausführung von Operationen auf allen Elementen, Bildung von sog. Slices, ...)

Bitsets (#include <bitset>)

```
namespace std {
  template< size_t bits > class bitset;
}
```

Bitvektoren (konstanter Länge)

vector<bool> - kein Container von bool's

```
std::vector<bool> v;

bool *pb = & v[0]; // nicht übersetzbar ☹
```

Scott Meyers „Effective STL“ (Item 18:) Avoid using vector<bool>.

Allokatoren `(#include <memory>)`

- Separation der Speicherverwaltung für dynamische Objekte (Listen- und Baum-Knoten etc.) von den abstrakten Containern: Container enthalten als Typparameter eine Klasse, die die Speicherverwaltung komplett übernimmt
- Standardmäßig stellt jede Implementation einen *default allocator* in der Klasse `std::allocator` bereit, dieser verwaltet geeigntet Memory-Pools
- alternative Allokatoren (z.B. mit garbage collection, oder auf verschiedenen Speichermodellen ...) sind möglich und beeinflussen die eigentliche Funktionalität der Container in keiner Weise !!

auto_ptr (#include <memory>)

```
namespace std {
  template<class X> class auto_ptr {
    template <class Y> struct auto_ptr_ref {};
  public:
    typedef X element_type;
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y>
       auto_ptr& operator=(auto_ptr<Y>&) throw();
   ~auto_ptr() throw();
```

auto_ptr (#include <memory>)

```
    // ...
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
    X* release() throw();
    void reset(X* p =0) throw();

    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref<Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
  };
}
```

auto_ptr (#include <memory>)

owning pointers - ein auto_ptr -Objekt kapselt einen Zeiger, bei Zuweisungen (Initialisierungen) von auto_ptr -Objekten wird die Zuständigkeit weitergereicht, im Quellobjekt wird 0 hinterlegt, auto_ptr ist daher NICHT in Containern benutzbar !

std::tr1::shared_ptr/weak_ptr z.B. hier

auto_ptr -Objekte verhalten sich selbst wie Zeiger (operator->)
damit lassen sich auch dynamisch erzeugte Objekte *exception safe* verwalten:

```
void foo() {                void foo() {
  X* p = new X;                 auto_ptr<X> p (new X);
  p->bar();                   p->bar();
  // leak ???                 // never leaks:
  delete p;                     // ~auto_ptr<X>() !!!
}                             }
```

Erweiterungen (über den derzeit standardisierten Umfang hinaus)

in konkreten Implementationen meist nicht als solche ausgewiesen, sondern unterschiedslos mit den standardisierten Klassen/Funktionen realisiert !

maßgeblicher Vertreter (und guter Kandidiat für 2nd revision des C++ -Standards) ist die SGI - STL, die folgende wichtige Zusätze implementiert (z.B. ab libg++2.8.1):

– neue Algorithmen: "Kleinigkeiten", die sich aber nicht allgemein und effizient mit den vorhandenen generischen Algorithmen realisieren lassen: `iota` (Bereich mit aufsteigenden Zahlen initialisieren) , `random_sample` (eine zufällige Stichprobe ohne Änderung des Originalcontainers [statt `random_shuffle`]), etc.

– ein verbesserter `sort`-Algorithmus (aka introsort) der sich auch bei nahezu vorsortierten Containern optimal [O(n log n)] verhält

Erweiterungen (über den derzeit standardisierten Umfang hinaus)

- neue Container: die assoziativen Container `[multi]set`, `[multi]map` sind immer sortiert, dies ist nicht immer nötig (kostet aber *overhead*) bzw. manchmal sogar unerwünscht ---> TR1 `unordered_set`, `unordered_map`, `hash_multiset`, `hash_multimap` in der Benutzung wie zugehörige Container, aber effizienter
- nicht immer braucht man doppelt verkettete Listen ---> TR1 `slist`
- eine spezielle Stringklasse `rope` (*Ropes are a scalable string implementation: they are designed for efficient operation that involve the string as a whole. Operations such as assignment, concatenation, and substring take time that is nearly independent of the length of the string. Unlike C strings, ropes are a reasonable representation for very long strings such as edit buffers or mail messages*) – nicht in TR1
- *thread safety* - derzeitige Container sind nicht notwendig *thread safe* !!!

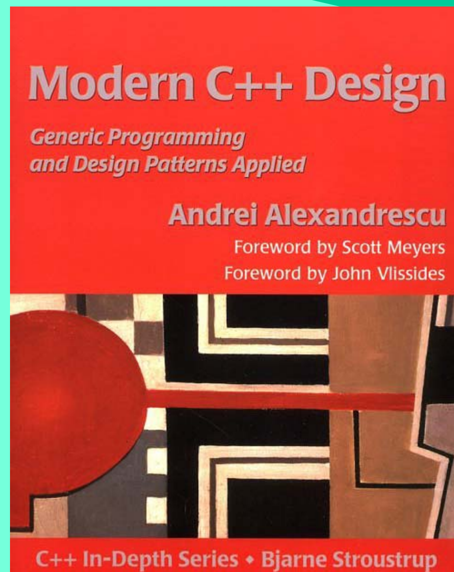
0. Einführung

Der Einsatz des Konzeptes der generischen Typen (Templates) verändert heute die Art und Weise der Programmierung in C++ weit mehr, als dies ursprünglich von den Erfindern der Sprache vorauszuahnen war. Die STL war erst der Anfang. Es ist vor allem das Verdienst von Andrei Alexandrescu mit der radikalen Anwendung dieser Technik völlig neue, bislang ungeahnte Perspektiven für den Entwurf flexibler und hochgradig wiederverwendbarer Bibliotheken in C++ eröffnet zu haben.

Die Vorlesung befasst sich mit einigen dieser Techniken in Theorie und Praxis.
Voraussetzung sind fundierte und komplette Kenntnisse der Sprache C++. Vertrautheit mit klassischen Entwurfsmustern ist nützlich, aber nicht zwingend.

1. Quellen

www.moderncppdesign.com
sourceforge.net/projects/loki-lib/



ISBN 0-201-70431-5

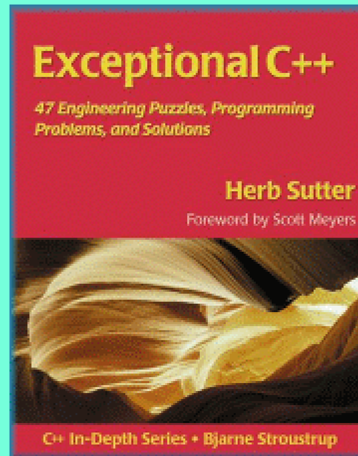
ISBN 3-8266-1347-3

Humboldt Universität Informatik

1. Quellen

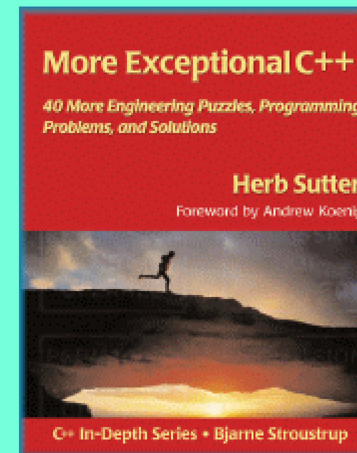
Bücher von Herb Sutter

www.gotw.ca/publications/xc++



ISBN 0-201-61562-2

www.gotw.ca/publications/mxc++

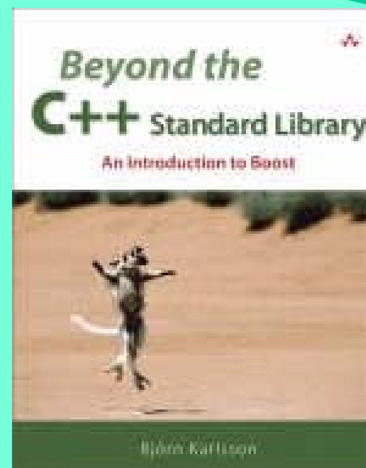


ISBN 0-201-70434-X

1. Quellen

Boost.org

www.boost.org



ISBN 0-321-13354-4

Humboldt Universität Informatik

1. Quellen

C++ Users Journal

www.cuj.com

Usenet news

comp.lang.c++

comp.lang.c++.moderated

comp.std.c++

Artikelsammlung von Bjarne Stroustrup

www.research.att.com/~bs/papers.html

Humboldt Universität Informatik

2. Compile-Time-Polymorphism

H. Sutter: More Exceptional C++

Item1: Switching Streams (Difficulty: 2 [of 10])

‚What is the best way to dynamically use different stream sources and targets, including the standard console stream and files?'

1. What are the types of std::cin and std::cout ?
2. Write an ECHO program that simply echoes ist input and that can be invoked equivalently in the two following ways:

```
ECHO <infile >outfile
ECHO infile >outfile
ECHO infile outfile
```

Humboldt Universität Informatik

2. Compile-Time-Polymorphism

ad 1.

```
namespace std { // <iostream>
        extern istream cin;   // ? istream ?
        extern ostream cout; // ? ostream ?
        //...
}

namespace std { // <istream>
   template<class Elem, class Tr = char_traits<Elem> >
   class basic_istream;
   typedef basic_istream<char, char_traits<char> > istream;
}
```

Humboldt Universität Informatik

2. Compile-Time-Polymorphism

```
namespace std { // <ostream>
  template<class Elem, class Tr = char_traits<Elem> >
  class basic_ostream;
  typedef basic_ostream<char, char_traits<char> > ostream;
  // ...
}
```

`char_traits ?`

trait [trei] *s* (*pl* **~s** [~z]) Zug, Merkmal

17.1.18 traits class [ISO/IEC 14882:1998(E)] **[defns.traits]**

a class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated. Traits classes defined in clauses 21, 22 and 27 are *chararacter traits*, which provide the character handling support needed by the string and iostream classes.

Humboldt Universität Informatik

2. Compile-Time-Polymorphism

H. Sutter:

„The idea is that traits classes are instances of templates, and are used to carry extra information – esp. information that other templates can use – about the types on which the traits template is instantiated. The nice thing is that the traits class T<C> lets us record such extra information about a class C, without requiring any change at all to C.“

(and even if C isn't a class at all !)

Humboldt Universität Informatik

2. Compile-Time-Polymorphism

ad 2.

```
// The tersest solution: A one-statement wonder
#include <fstream>
#include <iostream>
int main( int argc, char* argv[] )
{
  using namespace std;
  (argc > 2 ?
    ofstream(argv[2], ios::out | ios::binary) : cout)
  <<
  (argc > 1 ?
    ifstream(argv[1], ios::in  | ios::binary) : cin)
    .rdbuf();
}
```

2. Compile-Time-Polymorphism

Aber:

Prefer readability!

Avoid writing terse code
(brief, but difficult to understand and maintain)!

Eschew obfuscation!

Prefer extensibility!

Prefer encapsulation!

Separate concerns !