STL

Standard Template Library

Mathias Linke, IN00, HTWK-Leipzig, 6. April 2004

Gliederung

- Einführung in die Standardbibliothek:
  Motivation, Bezugsquellen, Namespace, Headerdateien
- Grundkurs zu Templates:
  Motivation, Deklaration, Beispiele
- Standard Template Library:
  Container, Iterator, Algorithmen, Funktoren

Einführung in die Standardbibliothek

Einführung — Motivation

- Mit der Standardisierung der Sprache C++ entstand der Wunsch auch die vorhanden Bibliotheken zu standardisieren. Vorhandene I/O- und String-Funktionen wurden zusammengefasst und durch Container-Klassen und Algorithmen ergänzt.
- Die Standardbibliothek ist ein **effizientes**, **getestetes** und **portables** Programmierwerkzeug.
- Die STL ist ein bedeutender Bestandteil und bietet fertige Lösungen für immer wiederkehrende Programmieraufgaben.

Einführung — Infos

- Literatur:

  Nicolai Josuttis
  Die C++ Standardbiliothek
  Addision-Weslay, 1996

  David R. Musser, Atul Saini
  STL Tutorial and Reference Guide
  Addision-Weslay, 1996

- Bezugs- und Informationsquellen:
  www.sgi.com/tech/stl/
  www.boost.org/libs/libraries.htm
  www.codeproject.com/vcpp/stl/

Einführung — Compileranforderungen (1)

- Templates in allen Varianten
- Exception-Handling:
  Die Standardbibliothek definiert mehrere Exception-Klasse, die alle von `class exception` abgeleitet sind.
  `logic_error, runtime_error, out_of_range, ...`
- Namespaces:
  Alle Identifier sind normalerweise im Namespace `std` definiert.
- Datentype `bool`:
  Dadurch lassen sich Funktionen speziell für diesen Typ überladen.

Einführung — Compileranforderungen (2)

- Schlüsselwort `explicit`:
  Verhindert, dass ein Konstruktor, der nur einem Argument hat, automatisch eine Typumwandlung definiert.

```
class Stack{
  explicit Stack( int size ); // Stack mit Startgroesse
  };
```

  Ohne `explicit` gäbe es eine automatische Typumwandlung von `int` nach `Stack`.

  ```
  Stack s; ... s = 40;
  ```

  Die `40` würde einen leeren Stack mit 40 Elementen erzeugen, der dann `s` zugewiesen wird.

Einführung — Compileranforderungen (3)

- Neue Operatoren zur Typumwandlung:
  - `const_cast`, entfernt nur die Konstantheit von Objekten
  - `static_cast`, wandelt Datentyp logisch um (wenn definiert)
  - `dynamic_cast`, bei Einsatz von Polymorphie lassen sich Objekte, wenn möglich, in ihren tatsächlichen Typ umwandeln.
  - `reinterpret_cast`, interpretiert Bits neu (ohne jede Prüfung!)
- Initialisierung von konstanten statischen Komponenten:

```
class BspKlasse{
  static const int anzahl = 100;
  int elements[anzahl]; // Konstante sofort nutzbar
};
```

Einführung — Erste Schritte

- Namespace `std`:
  Alle Identifier sind normalerweise im Namespace `std` definiert. Bei Verwendung der Standard-Stream-Klassen für die Ein- und Ausgabe ist also statt `ostream` nun `std::ostream` zu verwenden.

  ```
  std::cout << std::hex << 3.4 << std::endl;
  ```

  Oft wird mit

  ```
  using namespace std;
  ```

  der Namespace an geeigneter Stelle aufgelöst.

- Header-Dateien:
  Die Standard-Header werden ohne jede Endung eingebunden.
  `#include <iostream>` statt `#include <iostream.h>`

Grundkurs zu Templates

Grundkurs zu Templates — Motivation

Die Funktion `swap` soll zwei per Referenz übergebene Werte vertauschen. Für den Datentyp `int` sieht die Funktion so aus.

```
void swap( int& lhv, int& rhv ){
  int temp = lhv;
  lhv = rhv;
  rhv = temp;
}
```

Für alle anderen Daten-Typen ist der Algorithmus gleich. Trotzdem ist `swap` für jeden einzelnen neu zu überladen!

Bei Listen würde es reichen, statt mit echten Element-Typen zu arbeiten, `void`-pointer zu nutzen. Dadurch geht jedoch die **Typsicherheit** verloren.

Grundkurs zu Templates — Deklaration

Die Deklaration beginnt mit dem Schlüsselwort `template`. In den spitzen Klammern dahinter, werden die Parameter angegeben. Sie lassen sich in der Funktion an der Stelle von Typen oder Konstanten verwenden.

```
template< typname T >
void swap( T& lhv, T& rhv ){
  T temp = lhv;
  lhv = rhv;
  rhv = temp;
}
```

Dieses Template funktioniert nur mit Typen, für die ein **Zuweisungsoperator** existiert!

Notfalls ist er für eigene Klassen selbst zu schreiben.

Grundkurs zu Templates — Instanzenbildung

Die genaue Typisierung der `swap`-Funktion erfolgt bei ihrer Verwendung weitestgehend automatisch. Der Compiler überprüft bei der Übersetzung, ob es schon eine entsprechende Instanz gibt. Falls nicht, wird der komplette Programmteil für den entsprechenden Typ erneut eingefügt.

Es ist jedoch zu beachten, dass bei der automatischen Instanzenbildung **keine implizite Typumwandlung** stattfindet.

```
int a=3, b=5;
double c=1.4, d=4.1;
swap( a, b ); // ok a = 5 und b = 3 Typ int, int
swap( c, d ); // ok c = 4.1 und d = 1.4 Typ double, double
swap( a, d ); // FEHLER, keine Schablone fuer int, double
```

Grundkurs zu Templates — Templateklassen (1)

Templates lassen sich auch mit Klassen nutzen. Die Definition ähnelt der von Funktionen.

```
template< typename T, int SIZE = 10 >  // 10 ist Default
class MyArray{
private:
  T _array[SIZE];
public:
  Array( T value); // initialisiert _array mit value
  T Get( int index ); // liest Werte
  void Set( int index, T value ); // setzt Werte
};
```

Es lassen sich auch einzelne Methoden und innere Klassen als Template definieren. (⇒ Iteratoren bei Listen)

Grundkurs zu Templates — Templateklassen (2)

Die Deklaration ist etwas kompliziert und zeigt, dass die Typinformation untrennbar mit der Klasse verbunden ist.

```
template< typename T, int SIZE = 10 >
MyArray< T, SIZE >::MyArray( T value ){
  for( int x = 0; x < SIZE; x++ )
    _array[x] = value;
}
template< typename T, int SIZE = 10 >
T MyArray< T, SIZE >::Get( int index ){
  return _array[index];  /*Bereichspruefung fehlt noch*/
}
template< typename T, int SIZE = 10 >
void MyArray< T, SIZE >::Set( int index, T value ){
  _array[index] = value;  /*Bereichspruefung fehlt noch*/
};
```

Grundkurs zu Templates — Templateklassen (3)

Bei der Verwendung der Klasse sind in spitzen Klammern die gewünschten Ersetzungen für Typen und Konstanten anzugeben.

```
MyArray<int> a1( 0 );        // Platz fuer 10 int
MyArray<double> a2( 0.0 ); // Platz fuer 10 double
MyArray<int,20> a3( 0 );    // Platz fuer 20 int
a1.Set( 1, 5 )    // ok, 5 ist int
a1.Set( 1, 5.2 ) // ok, aber Verlust der Nachkommastelle
a2.Set( 1, 2.5 ) // ok, 2.5 ist double
a2.Set( 1, 6 )    // ok, int 6 wird zu double
```

Bemerkung: Obwohl bei der Verwendung der Set-Methode für die Werte eine automatische Typumwandlung stattfindet, lassen sich die Arrays nicht untereinander zuweisen.

Jede dieser MyArray-Instanzen hat einen anderen Typ!

Grundkurs zu Templates — Zusammenfassung

Templates sind ein mächtiges und vielseitiges Sprachmittel.

- Doppelte Schreibarbeit vermeiden
- Typsichere generische Programmierung

Problem der Programmaufblähung:

- Oft sind in Klassen nicht alle Methoden oder Membervariablen vom gewählten Typ abhängig. Trotzdem werden sie mehrfach in den Code eingefügt.
- Typunabhängige Programmteile sind außerhalb von Templates zu platzieren. Dies lässt sich durch Basisklassen realisieren.

STL Überblick

STL Überblick — Das Konzept (1)

- Datenverarbeitung ist meist Mengenverarbeitung.
- Die STL bietet Lösungen für beliebige Arten von Mengen mit frei definierbarem Element-Typ.
- Entgegen einem strengen OO-Design, sind in der STL **Daten und Algorithmen getrennt**.
- Die STL basiert auf den Komponenten:
  - Container
  - Iteratoren
  - Algorithmen

STL Überblick — Das Konzept (2)

- Container
  - verwalten eine Menge von Objekten eines bestimmten Typs.
  - Unterschiedliche Container spiegeln unterschiedliche Techniken bei der Implementierung wieder.
  - Sie haben entsprechende Vor- und Nachteile.
- Iteratoren
  - dienen dazu, über die Elemente eines Containers zu iterieren.
  - haben eine einheitliche Schnittstelle, die die Struktur des Containers verbirgt.
  - lassen sich im allgemeinen wie einfache Zeiger benutzen.

STL Überblick — Das Konzept (3)

- Algorithmen
  - Sie dienen dazu die Elemente eines Container-Bereiches zu bearbeiten.
  - Dabei stützen sie sich auf die Iteratoren, welche der Container zur Verfügung stellt.
  - Für die Algorithmen kann der Programmierer oft Hilfsfunktionen definieren, um speziellen Anforderungen gerecht zu werden.

STL Überblick — Container (1)

- Sequentielle Container
  - geordnete Mengen deren Elemente ein Position haben, die durch den Ort und den Zeitpunkt des Einfügens bestimmt ist.
  - Vordefiniert: `vector`, `deques`, `list`
- Assoziative Container
  - sortierte Mengen, bei denen die Position der Elemente durch ein Sortierkriterium definiert ist.
  - Die Sortierfunktion ist frei wählbar. Standard ist $<$.
  - meist als Binär-Baum implementiert
  - Vordefiniert: `set`, `map`, `multiset`, `multimap`

STL Überblick — Container (2)

- `vector<>`
  - dynamisches Array, random access, Einfügen / Löschen am Ende optimal schnell
- `deque<>`
  - dynamisches Array, kann in beide Richtungen wachsen, random access, Einfügen / Löschen am Anfang und Ende optimal schnell
- `list<>`
  - doppelt verkettete Liste, kein random access, Einfügen / Löschen überall optimal schnell

STL Überblick — Container (3)

- `set<>`
  - keine doppelten Elemente, automatisch nach ihrem Wert sortiert
- `map<>`
  - speichert Key/Value-Elemente, Key darf nur einmal vorkommen, assoziatives Array
- `multiset<>`
  - wie `set<>`, doppelte Elemente sind zulässig
- `multimap<>`
  - wie `map<>`, doppelte Schlüsselwerte sind zulässig

STL Überblick — Container (4)

- Container-Adapter
  - bilden die fundamentalen Container auf spezielle Anforderungen ab
- Stack
  - Stapel, Keller, LIFO
- Queue
  - Puffer, FIFO
- Priority-Queue
  - Puffer, beim Auslesen der Elemente spielt die Priorität ein Rolle

STL Überblick — Container (5)

- Anforderung an Containerelemente
  - **Copy-Konstruktor**
    Container legen grundsätzlich Kopien der Elemente an und geben Kopien zurück!
  - **Zuweisungsoperatror** $=$
    Container und Algorithmen benutzen ihn um Elemente an Position zu kopieren, wo schon Elemente sind
  - **Destruktor** Container zerstören 'gelöschte' Elemente
  - **Vergleichsoperator** $==$ für die Elementsuche
  - Sequentielle Container erfordern einen **Default-Konstruktor**
  - Assoziative Container erfordern den **Operator** $<$

STL Überblick — Container (6)

- gemeinsame Operationen aller Container

```
size()   // akt. Elementanzahl
empty()  // prueft, ob Container leer ist
==, !=, <, >, <=, >= // Vergleiche
=                    // Zuweisung ganzer Container
begin()   // Iterator fuer das erste Element
end()     // Iterator hinter dem letzten Element
rbegin()  // Reverse-Iterator fuer das letzte Element
rend()    // Reverse-Iterator vor dem ersten Element
insert( pos, elem ) // fuegt Kopie von elem an pos ein
erase( pos )        // loescht das Element an pos
erase( anf, end )   // loescht den Bereich [anf,end)
clear()             // loescht den Container

pos, anf, end sind Iteratoren!
```

STL Überblick — Iteratoren (1)

- sind Objekte, die über die Elemente von Containern wandern können.
- Jedes Iterator-Objekt repräsentiert eine Position in einem Container.
- Iterator-Objekte werden durch die Container bereitgestellt.
- Grundfunktionen sind:

```
operator*()   // liefert das Element an der
    Iteratorposition
operator++()  // setzt den Iterator auf das naechste
    Element
operator==()  // true, wenn zwei Iteratoren dasselbe
    Element repraesentieren
```

STL Überblick — Iteratoren (2)

- In Abhängigkeit von den Containern können die Iteratoren weitere Fähigkeiten besitzen.
- **Bidirectional-Iteratoren**
  können auch in einer Menge zurückgehen, dazu dient der
  `operator--()`
- **Random-Access-Iterator**
  sind erweiterte Bidirectional-Iteratoren. Sie beherrschen zusätzlich Adress-Arithmetik. Differenzen, Addition bzw. Subtraktion von Offsets und Vergleiche $<$ und $>$

STL Überblick — Container/Iterator Beispiel (1)

- Container stellen mit den Funktionen `begin()` und `end()` zwei Iteratoren zur Verfügung, die zusammen ein **halboffene** Menge gültiger Positionen bestimmen.

```
#include <list>
using namespace std;
...
  list<char> m; // List-Container fuer char
  for( char c='a'; c<='z'; c++ )// a bis z einfuegen
    m.push_back( c );

  // Alle Elemente ausgeben
  list<char>::iterator pos;
  for( pos = m.begin(); pos != m.end(); ++pos )
    cout << *pos << endl;
```

STL Überblick — Container/Iterator Beispiel (2)

Die Elemente einer `map< T1, T2 >` sind vom Typ `pair< T1, T2 >`!

```
#include <map>                          // template< T1, T2 >
using namespace std;                    // struct pair{
...                                     //   T1 first;
  typedef map<string,float> SFMap;      //   T2 second;
  SFMap m;                              // };

  m["Pi"] = 3.1415;  // Wenn der Key nicht existiert
  float x = m["MW"]; // wird ein passender erzeugt x=0.0

  SFMap::iterator pos;
  for( pos = m.begin(); pos != m.end(); ++pos )
    cout << pos->first << " / " << pos->second << endl;
```

STL Überblick — Algorithmen (1)

- In der STL sind die Algorithmen globale Funktionen, welche mit Iteratoren arbeiten.
- Vorteil: Wiederverwendbarkeit auch mit gemischten und selbst definierten Containern.
- Nachteil: Für bestimmte Anwendungen besitzen sie ein schlechtes Zeitverhalten.
- **Nicht-Modifizierende Algorithmen**
  Lese-Zugriff, Suchen und Finden, Min/Max, Zählen, Vergleichen
- **Modifizierende Algorithmen**
  Schreib-Zugriff, Kopieren, Vertauschen, Löschen

STL Überblick — Algorithmen (2)

- Beispiel: Algorithmen mit einem Container

```
vector< int > v;

v.push_back( 2 );v.push_back( 4 );v.push_back( 3 );
v.push_back( 1 );v.push_back( 5 );v.push_back( 4 );

vector< int >::iterator pos;         // random access
pos = min_element( v.begin(), v.end() );
cout << "Min=" << *pos << endl;

pos = max_element( v.begin(), v.end() );
cout << "Max=" << *pos << endl;

sort( v.begin(), v.end() );
reverse( v.begin()+1, v.end()-2 ); // Teilbereich!
```

STL Überblick — Algorithmen (3)

- Beispiel: Algorithmen mit zwei Containern
- **Achtung: Der Zielbereich muss genügend groß sein!**

```
list< int > m1;
vector< int > m2;

for( int i=1; i<=9; i++)
  m1.push_back( i );

m2.resize( m1.size() ) // !!!

copy( m1.begin(), m1.end(), m2.begin() )
```

- ohne `resize()` währe `m2` zu klein und das Programm würde mit einem Fehler abbrechen.

STL Überblick — Algorithmen (4)

- Algorithmen sich durch Funktionen und Funktionsobjekte parametrisierbar.

```
void print_elem( int elem ){
  cout << elem << " ";
}

class PrintInt{
  public: void operator()( int elem ){
            cout << elem << " ";
          }
};

for_each( m.begin(); m.end(); print_elem );
for_each( m.begin(); m.end(); PrintInt() );
```

STL Überblick — Algorithmen (5)

- Es sind schon sehr viele Funktionsobjekte vordefiniert.

```
negate<typ>()                          // -param
plus<typ>()                            // param1 + param2
  auch - * / \%
equal_to<typ>()                        // param1 == param2
  auch !=, <, >, <=, >=
logical_not<typ>                       // !param
  auch logical_and<typ> und logical_or<typ>

// quadrieren der Werte aller Elemente in m
transform( m.begin(), m.end()  // erster Quellbereich
           m.begin(),          // zweiter Quellbereich
           m.begin(),          // Zielbereich
           multiples<int>() ); // Operation
```

STL Überblick — Algorithmen (6)

- Funktionsobjekte sind durch Funktions-Adapter erweiterbar.
- Dadurch lassen sie sich kombinieren und mit Werten versehen.

```
// multiplizieren aller Elemente in m mit 7
transform( m.begin(), m.end() // erster Quellbereich
           m.begin(),         // zweiter Quellbereich
           bind2nd( multiples<int>(), 7 );// Operation

// finde das erste Element, das kleiner als 17 ist
find_if( m.begin(), m.end(),              // Bereich
         bind2nd( less< int >, 17 ) );    // Operation

// suche das erste gerade Element aus einem Bereich
find_if( m.begin(), m.end(),
         not1( bind2nd( modulus<int>(), 2 )));
```

STL Überblick — Algorithmen (7)

- Wichtige Hinweise
- Container und Algorithmen haben **keine Bereichsprüfung**!
- Der Programmierer muss sicher stellen, dass
  1. der Iterator für den Bereichsanfang **vor** dem Iterator für das Bereichsende liegt
  2. das Bereichsende vom Bereichsanfang aus **erreichbar** ist.
- Auf assoziative Container sind manipulierende Algorithmen nicht anwendbar. Deshalb bieten diese Container selbst entsprechende Funktionen an.

STL Überblick — Zusammenfassung

- Die STL ist ein umfangreiches und mächtiges Werkzeug, welches Programmierer von Routinearbeit befreit.
- Der Umgang mit Containern und Iteratoren ist leicht erlernbar und einheitlich anwendbar.
- Die Definition von Hilfsfunktionen und Funktionsobjekten ist nicht immer gewünscht. Oft werden diese Funktionen nur einmal gebraucht und daher in ein `for`-Schleife gepackt.
- Obwohl die Funktionsadapter sehr mächtig sind, ist ihre Anwendung oft schwer nachvollziehbar.
- **Die STL sollte jeder C++ Programmierer kennen.**