

Systemnahe Programmierung in C/C++

Peter Pehler

`peter.pehler@symetrion.com`

Lehrstuhl für Datenbanken und Informationssysteme
Fakultät für Mathematik und Informatik

2006-12-13

Agenda

- 1 Function Templates
- 2 Class Templates
- 3 Nontype Template Parameters
- 4 STL Container
- 5 Aufgaben

- 1 Function Templates
- 2 Class Templates
- 3 Nontype Template Parameters
- 4 STL Container
- 5 Aufgaben

Definition von Function Templates

- Funktion-Templates bieten die Möglichkeit, typunabhängige Implementierungen zu realisieren
- ein Funktion-Template repräsentiert eine Familie von Funktionen
- Funktion-Templates werden wie gewöhnliche Funktionen realisiert, mit der Ausnahme, dass manche Elemente unbestimmt sind
- Definition eines Beispiel-Templates:

```
template< typename T >  
bool before( const T& lhs, const T& rhs )  
{  
    return( lhs < rhs );  
}
```

- diese Templatedefinition spezifiziert eine Familie von Funktionen, welche ermitteln, ob eine Instanz vor einer anderen sortiert wird
- die Typen der Funktionsparameter sind freigelassen als *Templateparameter T*

Verwendung von Function Templates

- Der folgende Programmausschnitt zeigt die Benutzung des `before` Funktionstemplates:

```
int i = 5;
before( i, 5 );

double d = -5.5;
before( 3.4, d );

std::string str1 = "Informatik";
std::string str2 = "inf.";
before( str1, str2 );
```

- `before` wird drei mal gerufen, einmal für `int`, einmal für `double` und einmal für `std::string`
- der Prozess der Ersetzung der Templateparameter durch konkrete Typen wird *Instanziierung* genannt.
- leider ist die Verwendung des Begriffs *Instanziierung* nicht mehr eindeutig (historisch als Begriff für die Erschaffung eines konkreten Objektes einer Klasse)

- bei einem Aufruf einer Templatefunktion, wie `before`, werden die Templateparameter (Typen) durch die übergebenen Argumente ermittelt
- für die automatische Typerkennung müssen die Typen eindeutig ermittelbar sein:

```
before( 5, 5.5 ); //ERROR: erstes Argument int, zweites double
```

- Es gibt zwei Möglichkeiten:
 - 1 explizite Typkonvertierung der Parameter

```
before( static_cast<double>(5), 5.5 );
```

- 2 Explizite Spezifizierung der Templateparameter

```
before<double>( 5, 5.5 );
```

- Funktion-Templates besitzen zwei Arten von Parametern:
 - 1 *Templateparameter*, diese werden in spitzen Klammern eingeschlossen und vor dem Funktionsnamen deklariert

```
template< typename T > //T ist ein Templateparameter
```

- 2 *Aufrufparameter*, diese werden in Klammern gesetzt und stehen nach dem Funktionsnamen

```
before( const T& lhs, const T& rhs ); //lhs und rhs sind Aufrufparameter
```

- um eine Templatefunktion zu schreiben, welche zwei unterschiedliche Typen akzeptiert, schreibt man:

```
template< typename TL, typename TR  
bool before( const TL& lhs, const TR& rhs );  
{ return(lhs < rhs); }
```

- der Return-Typ kann festgelegt werden durch:

```
template< typename RT, typename TL, typename TR
RT before( const TL& lhs, const TR& rhs );
{ return(lhs < rhs); }
```

und verwendet werden durch:

```
before< bool >( 5, 5.5 ); //Retrun-Typ ist bool,
                        //erster Aufrufparameter ist int,
                        //zweiter Aufrufparameter ist double
```

- 1 Function Templates
- 2 Class Templates**
- 3 Nontype Template Parameters
- 4 STL Container
- 5 Aufgaben

Deklaration von Klassen-Templates

- die Deklaration von Klassen-Templates ist gleich der Deklaration von Funktionstemplates:

```
template< typename T >  
class Stack {  
    ...  
};
```

- das Schlüsselwort `class` kann an Stelle von `typename` verwendet werden (veraltet)

```
template< class T >  
class Stack {  
    ...  
};
```

Deklaration von Klassen-Templates

- innerhalb der Klasse kann T wie jeder bekannte Datentyp verwendet werden

```
template< typename T >
class Stack {
    private:
        std::vector< T > elems;
    public:
        Stack(); //Defaultkonstruktor
        void push( const T&); //pop Element
        void pop() throw(std::out_of_range); //pop Element
        T top() const throw(std::out_of_range); //Kopie des obersten Elementes
};
```

- Der Typ dieser Templateklasse ist Stack<T>. Diese Schreibweise ist zu benutzen, immer wenn der eigene Typ gemeint ist:

```
template< typename T >
class Stack {
    ...
    Stack( const Stack<T>& ); //Copy-Konstruktor
    Stack<T>& operator=( const Stack<T>&); //Zuweisungsoperator
    ...
};
```

- die Definition der Methoden erfordert die Angabe der Templateparameter:

```
template< typename T >
T Stack<T>::top() const throw(std::out_of_range)
{
    if( elems.empty() )
        throw std::out_of_range("Stack<>::top():_empty_stack");

    return elems.back();
};
```

- die Verwendung von Klassentemplates erfolgt wie folgt:

```
Stack<int> intStack;  
intStack.push( 5 );  
intStack.pop();  
  
typedef Stack< std::string > StringStack;  
StringStack stringStack;  
stringStack.push( "hello" );
```

Spezialisierung von Klassentemplates

- es ist möglich, für bestimmte Templateargumente ein Klassentemplate zu spezialisieren
- spezialisierte Klassentemplates ermöglichen:
 - eine optimierte Implementation für einzelne Typen
 - eine Behandlung von Fehlverhalten bei der Verwendung spezieller Typen
- die Spezialisierung erfolgt, indem der Typ, für den eine gesonderte Implementierung erfolgen soll, angegeben wird:

```
template<>
class Stack< std::string >
{
    private:
        std::deque<std::string> elems;
    public:
        ...
        //Kopie des obersten Elementes
        std::string top() const throw(std::out_of_range);
};
```

- die Definition eines Methodes eines spezialisierten Klassentemplates:

```
std::string Stack< std::string >::top() const throw(std::out_of_range)
{
    if( elems.empty() )
        throw std::out_of_range("Stack<_std::string_>::top():_empty_stack");

    return elems.back();
}
```

partielle Spezialisierung von Klassentemplates

- für das folgende Klassentemplate

```
template< typename T1, typename T2 >  
class Stack{  
    ...  
};
```

sind unterschiedliche partielle Spezialisierungen möglich:

- 1 beide Templateparameter sind vom gleichen Typ:

```
template< typename T >  
class Stack<T,T>{  
    ...  
};
```

- 2 zweiter Typ ist int

```
template< typename T >  
class Stack<T,int>{  
    ...  
};
```

- 3 beide Typen sind Zeiger

```
template< typename T1, typename T2 >  
class Stack<T1*,T2*>{  
    ...  
};
```

4 erster Typ ist eine Referenz

```
template< typename T1, typename T2>
class Stack<T1&,T2>{
    ...
};
```

5 erster Typ ist ein basic_string-Template

```
template< typename TChar, typename T2>
class Stack< std::basic_string<TChar> ,T2>{
    ...
};
```

default Template Arguments

- für Klassentemplates (nicht für Funktionstemplates) können default-Werte für die Templateargumente angegeben werden
- wird kein Templateargument angegeben, wird der default-Wert angenommen
- Deklaration

```
template< typename T, typename TContainer = std::vector<T> >
class Stack{
private:
    TContainer elements;
public:
    ...
    T top() const throw(std::out_of_range);
};
```

- Definition:

```
template< typename T, typename TContainer >
T Stack<T, TContainer >::top() const throw(std::out_of_range)
{
    if( elems.empty() )
        throw std::out_of_range("Stack<_std::string_>::top():_empty_stack");

    return elems.back();
}
```

- 1 Function Templates
- 2 Class Templates
- 3 Nontype Template Parameters**
- 4 STL Container
- 5 Aufgaben

Nontype Class Template Parameters

- im Gegensatz zu den bisherigen Beispielen ist auch möglich Werte als Templateparameter zu übernehmen
- Das Stack-Beispiel mit Verwendung eines Arrays, dessen Dimension als Templateparameter übergeben wird:

```
template< typename T, int TCapacity >
class Stack{
    private:
        T elements[TCapacity];
        ...
    public:
        ...
};

template< typename T, int TCapacity >
T Stack<T,TCapacity>::top() const throw(std::out_of_range)
{
    if( space <= 0 )
        throw std::out_of_range("Stack<_std::string_>::top():_empty_stack");

    return elems[ space -1 ];
}
```

Nontype Function Template Parameters

- wie es möglich ist, Werte als Templateparameter für Klassentemplates zu verwendet, ist dies auch bei Funktionstemplates möglich:

```
template< typename T, int TValue >  
T addValue( const T& x )  
{  
    return x + TValue;  
}
```

- 1 Function Templates
- 2 Class Templates
- 3 Nontype Template Parameters
- 4 STL Container**
- 5 Aufgaben

- Container sind Behälter, die Objekte des gleichen Typs aufnehmen und verwalten
- Container sind als Klassentemplates realisiert
- als ein Templateparameter wird der Typ übergeben, von dem Objekte verwaltet werden sollen, Bsp:

```
#include <vector>

std::vector< std::string > stringVector;
```

- an die, von einem Container zu verwaltenden Typen werden zwei Anforderungen gestellt:
 - 1 sie müssen *copy-constructible* sein
d.h. es muß ein Copy-Konstruktor vorhanden sein
 - 2 sie müssen *assignable* sein
d.h. es muß ein Zuweisungsoperator existieren

- bei der Aufnahme eines Objektes in einen Container wird dieses Kopiert
- die Kopie befindet sich im Besitz des Container
- sobald der Container zerstört wird, werden alle von ihm verwalteten Objekte abgeräumt
- die Container besitzen drei wesentliche Eigenschaften:
 - ① sie automatisieren die Speicherverwaltung
 - ② sind Typsicher (kein Typecast!)
 - ③ sie zwingen den Benutzer in keine Vererbungshierarchie
- zu beachten ist:
 - Container nehmen meist keine Prüfung vor, ob Methodenaufrufe sinnvoll sind.
 - Objekte werden nicht im 'Original' im Container abgelegt, sondern kopiert. Dies kann bei großen Objekten aufwendig sein.
 - Es kann sinnvoll sein, Zeiger oder Referenzen durch Container zu verwalten

- Container sind eine *Datenstruktur zur Verwaltung von Objekten*
- sie besitzen die dafür notwendige Funktionalitäten
- Algorithmen sind in der STL gesondert implementiert und lassen sich auf STL-konforme Container anwenden
- gemeinsam entfalten Container und Algorithmen ihre volle Kraft

- die Container der STL lassen sich in drei Gruppen gliedern:
 - ① sequenzielle Container (vector, deque, list)
 - ② assoziative Container (set, multiset, map, multimap)
 - ③ Containderadapter (stack, queue, priority_queue)

Container, Allgemeine Anforderungen

- alle Container müssen einen Satz von *Typdefinition* zur Verfügung stellen, der von anderen Komponenten benutzt werden kann
- es muß gültig sein: `container< ... >::typ-name`
- für `typ-name` muß in jedem Container definierte sein:
 - `value_type` Typ der Elemente, die der Container verwaltet
 - `allocator_type` Typ des Allokators, der für die Speicherverwaltung zuständig ist
 - `reference` Referenz-Typ der Elemente, die der Container verwaltet.
Meist `value_type&`.
 - `const_reference` Konstanter Referenz-Typ der Elemente die der Container verwaltet. Meist `const value_type&`.
 - `pointer` Zeiger-Typ der Elemente, die der Container verwaltet.
Meist `value_type*`.
 - `const_pointer` Konstanter Zeiger-Typ, der Elemente die der Container verwaltet. Meist `value_type*`.

iterator Iteratortyp, der auf ein Objekt des Typs `value_type` verweist. Iteratoren werden gesondert behandelt.

const_iterator Iteratortyp mit dem ausschließlich lesend auf Elemente zugegriffen wird.

difference_type implementierungsabhängiger Abstandstyp, der Werte der Subtraktion von Iteratoren aufnimmt.

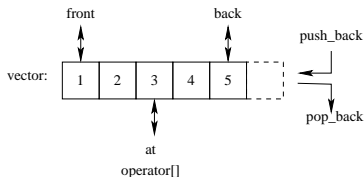
size_type implementierungsabhängiger, vorzeichenloser ganzzahliger Typ, der alle nicht negativen Werte von `difference_type` aufnehmen kann und als Index auf Containerelemente benutzbar ist.

..gekürzt..

- beherbergen eine endliche Anzahl von Objekten
- alle Objekte sind vom gleichen Typ
- die konkrete Position der Objekte in einem Container hängt vom *Zeitpunkt* und vom *Ort* der Einfügeoperation, aber *nicht vom Wert* ab.
- sequenzielle Container sind: vector, deque, list

sequenzielle Container - vector

- kann man sich als dynamisches Feld/Array vorstellen
- auf Elemente kann mit konstantem Aufwand zugegriffen werden (`vector[]`)
- Einfügen und Entfernen von Elementen am Ende ist effizient (konstanter Aufwand)
- Einfügen und Entfernen von Elementen am Anfang und im Container ist aufwendig (linearer Aufwand)



- effizienter Zugriff über Schlüssel
- meisten Operationen sind mit logarithmischem Aufwand verbunden
- assoziative Container sind: set, multiset, map

```
template< typename TKey,
          typename TValue,
          typename TCompare = less<TKey>,
          typename TAllocator = allocator< pair<const TKey, TValue> > >
class map{
    ...
};
```

- beinhaltet Schlüssel-Wert Paare
- die Schlüssel müssen eindeutig sein
- TKey ist der Schlüssel-Typ
- TValue ist der Wert-Typ
- TCompare ist ein Funktionsobjekt, um Schlüsselwerte zu vergleichen
- TAllocator Typ des Allocator für die Speicherverwaltung

assoziative Container - map

- das folgende Beispiel definiert ein Funktionsobjekt (ltstr) und verwendet es in einer map
- die map wird durch Verwendung des Indexoperators gefüllt

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

int main()
{
    map<const char*, int, ltstr> months;

    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;
}
```

- Zugriff auf die verwalteten Elemente über einen Iterator

```
cout << "june->" << months["june"] << endl;
map<const char*, int, ltstr>::iterator cur = months.find("june");
map<const char*, int, ltstr>::iterator prev = cur;
map<const char*, int, ltstr>::iterator next = cur;
++next;
--prev;
cout << "Previous_(in_alphabetical_order)_is_" << (*prev).first << endl;
cout << "Next_(in_alphabetical_order)_is_" << (*next).first << endl;
}
```

1

¹Quelle: <http://www.sgi.com/tech/stl/Map.html>

- der im Beispiel verwendete Indexoperator wird für map und multimap, nicht jedoch für die anderen assoziativen Container, bereitgestellt
- der Indexoperator ist deklariert als `mapped_type& operator[](const key_type& k);` mit `mapped_type` als Typdefinition des zum Schlüssel gehörenden Werts
- es wird also eine Referenz auf den zum Schlüssel `k` gehörenden Wert geliefert
- dadurch kann der Wert gelesen und modifiziert werden
- ist `k` nicht in der map enthalten, wird es eingefügt
- das Ergebnis entspricht `*((insert(make_pair(k, T()))).first)).second`

- hinzufügen von Elementen durch `pair<iteratort, bool> insert(const value_type& x);`

```
months.insert( pair<const char*, int>("february",29) );  
months.insert( make_pair("february",29));  
typedef map<const char*, int, ltstr> MonthMap;  
months.insert( MonthMap::value_type("february",29) );
```

- 1 Function Templates
- 2 Class Templates
- 3 Nontype Template Parameters
- 4 STL Container
- 5 Aufgaben**

Aufgabe 1 (Klassentemplates)

Problemstellung:

```
typedef TTypeCapsule< uint32_t, 0> EmployeeId_t;
typedef TTypeCapsule< uint32_t, 1> ProjectId_t;
uint32_t employeeId( 10 );
uint32_t projectId( 12 );
employeeId = projectId; //logical failure but no compiletime error
EmployeeId_t employeeId_( 10 );
ProjectId_t projectId_( 12 );
employeeId_ = projectId_; //logical failure AND compiletime error
```

Schreiben Sie das Klassentemplate TTypeCapsule mit der dargestellten Eigenschaft und implementieren Sie die arithmetischen Operatoren für TTypeCapsule

Aufgabe 2 (Container)

Schreiben Sie eine Klasse `TCPServiceRegister` welche:

- `TCP-servent` Strukturen verwaltet (siehe `man getservent`)
- eine performante Suche über die Portnummer implementiert hat
- eine performante Suche über den Dienstnamen implementiert hat
- STL-Container für die Realisierung verwendet
- halten Sie jede `servent`-Instanz nur einmal im Speicher

`getservbyname` und `getservbyport` sind für die Realisierung der Suche nicht zu verwenden.