



Praxis des Programmierens

Clemens Gröpl

Wintersemester 2009

15. C++ -- Templates Teil II

Bioinformatik

Universität Greifswald

An dieser Stelle ein herzlicher Dank an Oliver Kohlbacher
für die Überlassung der Präsentation! - Clemens Gröpl

Informatik II

Oliver Kohlbacher

Sommersemester 2006

15. C++ -- *Templates Teil II*

Abt. Simulation biologischer Systeme
WSI/ZBIT, Eberhard Karls Universität Tübingen

Templates für Fortgeschrittene

- Defaultparameter
- Nichttyp-Parameter
- Template-Methoden
- Partielle Spezialisierung
- Template-Metaprogrammierung
- Beispiel: Warteschlange

Defaultparameter

- Genau wie bei Funktionsargumenten können wir auch in Parameterlisten **Defaultwerte** angeben
- Der erste Parameter einer Template-Klasse darf aber keinen Defaultwert haben
- **Defaultparameter können bei Instanziierung entfallen**

Beispiel:

```
template <typename T1, typename T2 = T1>
class MyPair
{
    public:
    T1 first;
    T2 second;
};
int main()
{
    MyPair<int> iip;           // MyPair<int,int>
    MyPair<int, short> isp;  // MyPair<int,short>
}
```

Explizite Parameter bei Funktionen

- Templatefunktionen **deduzieren ihre Parameter** aus den Argumenten
- Unterscheiden sich die Templates im Rückgabewert, ist das Deduzieren nicht möglich
- Parameter können auch **explizit spezifiziert** werden, wenn eine automatische Deduktion nicht möglich oder gewünscht ist
- Dazu wird dem Funktionsnamen in spitzen Klammern eine Parameterliste nachgestellt

Beispiel:

```
template <T1, T2, T3>  
T1 sum(T2 a, T3 b);
```

```
double x = sum(1.2, 2.3); // 3.5
```

```
double y = sum<double, int, int>(1.2, 3.4); // 3
```

```
int i = sum<int>(1, 2.4); // sum<int, int, double>(..) = 3
```

Nichttyp-Parameter

- Template-Parameter können nicht nur Typdeklarationen sein
- Parameter können auch analog zu lokalen Variablendeklarationen gebraucht werden
- Nichttyp-Parameter müssen integrale Typen sein (ganzzahlige Typen)
- Fließkommatypen sind nicht erlaubt (evtl. in C++ 0x)
- Da die angegebenen Werte zur Compilezeit in der Funktion bzw. Klasse ersetzt werden müssen, sind sie im Templatecode selbst konstant

Nichttyp-Parameter

- Zum Beispiel ließe sich damit eine Vektor-Klasse für beliebige Dimensionen definieren:

```
template <typename T, int d = 3>
class VectorD
{
    public:
    VectorD() : x_(d) {}
    T& operator [] (int i) { return x_[i]; }
    protected:
    vector<T> x_;
};

...
VectorD<double, 4> v;
v[3] = 7.0;
```

Templatemethoden

- Wie man Funktionen als Templates definieren kann, so können natürlich auch Klassenmethoden selbst Templates sein
- Beispiel: **Range-Konstruktoren** in `std::vector`
 - Um ein Array aus einem Iteratorbereich zu konstruieren, benötigt man einen Konstruktor je Iteratortyp!
 - Jede Containerklasse (und jede Instanziierung davon!) definiert eigene Iteratortypen!
 - Damit müsste man einen eigenen Konstruktor z.B. für jede Variante schreiben (also für `list<int>`, `list<short>`, `vector<int>` etc.)
- Die Definition von `vector<T>::vector(iterator start, iterator end)` verbirgt sich eine **Templatemethode**

Templatemethoden

- Implementierung von `std::vector` sieht etwas komplexer aus, funktioniert aber ungefähr wie folgt:

```
template <typename T>
class vector
{
    ...
    template <typename Iterator>
    vector(Iterator start, Iterator end)
    {
        resize(end - start);
        for (; start != end; ++start)
        {
            (*this)[i] = *start;
        }
    }
};
...
deque<int> d = ...;
vector<int> v(d.begin(), d.end());
```

Template-Spezialisierung

- Templates lassen sich **spezialisieren**
- Damit kann man die Klassen bzw. Funktionen an die Erfordernisse bestimmter Typen anpassen

Beispiel: (16_greater.C)

```
template <typename T>
bool greaterThan(T a, T b)
{
    return (a > b);
}

...

cout << greaterThan(7, 6) << endl; // 1
const char* abc = "abc";
const char* ABC = "ABC";
cout << greaterThan(abc, ABC) << endl;
// 0! Fehler: Vergleich der Zeiger, nicht der Strings!
```

Template-Spezialisierung

- **Lösung:**
Definition eines **spezialisierten Templates**, das für einen bestimmten Typ definiert wird
- Compiler bevorzugt immer die spezialisierte Variante über der allgemeineren

Beispiel:

```
template <typename T>
bool greaterThan(T a, T b) {...}

template <>
bool greaterThan<const char*>(const char* a,
                               const char* b)
{
    return (std::strcmp(a, b) > 0);
}
...
cout << greaterThan(abc, ABC) << endl;
// 1! Verwendet greaterThan<const char*>()
```

Partielle Spezialisierung

- Spezialisierung muss nicht für alle Parameter erfolgen
- Wird nur ein Teil der Parameter spezialisiert, spricht man von partieller Spezialisierung
- Damit lassen sich für bestimmte Typkombinationen spezielle Behandlungen definieren
- Partielle Spezialisierung ist auch für Nichttypparameter möglich
- Damit lassen sich z.B. rekursive Funktionen effizient implementieren

Linken von partiell spezialisierten Templates

- Implementierungen **vollständig spezialisierter Templates** müssen beim g++ in den **.C-Dateien** stehen!
- Vollständig spezialisierte Templates haben keinen Template-Charakter mehr, sie müssen also nicht erst zur Compilezeit instanziiert werden
- Daher kann man sie in .C-Dateien unterbringen
- Sie werden in der Linkphase dann dazu gebunden
- **Partiell spezialisierte Templates** können in der **Header-Datei** stehen

Template-Metaprogrammierung

```
// 15_fak.C
template <int N>
class Fak
{
    public:
    static int value() { return N * Fak<N-1>::value(); }
};
```

```
// Abbruch der Rekursion durch
// partielle Spezialisierung für 0!
```

```
template <>
class Fak<0>
{
    public:
    static int value() { return 1; }
};
```

```
...
```

```
Fak<7>::value(); // == 5040 = 7!
```

Templates als Metasprache

- Templates definieren eine mächtige Metasprache, die sich zu vielen Dingen missbrauchen lässt
- Das gerade gezeigte Konzept nennt sich **Template-Metaprogrammierung**
- Template-Sprache wird dazu verwendet zur Compilezeit Rechnungen (hier: 7!) auszuführen und das Ergebnis zur Codeerzeugung heranzuziehen
- Man kann den Compiler sogar zum Berechnen von Primzahlen heranziehen ohne überhaupt irgendwelchen Code zu erzeugen, wie Erwin Unruh gezeigt hat [4]
- Hier ist das Ergebnis der Rechnung in den Fehlermeldungen des Compilers versteckt!

Template-Metaprogrammierung

```
// Prime number computation by
// Erwin Unruh
template <int i>
struct D
{
    D(void*);
    operator int();
};
template <int p, int i>
struct is_prime
{
    enum { prim = (p==2) || (p%i) &&
        is_prime<(i>2?p:0), i-1> :: prim };
};
template <int i>
struct Prime_print
{
    Prime_print<i-1> a;
    enum{prim = is_prime<i, i-1>::prim};
    void f()
    {D<i> d = prim ? 1 : 0; a.f();}
};
```

```
template<>
struct is_prime<0,0>
{ enum {prim=1}; };
template<> struct is_prime<0,1>
{ enum {prim=1}; };

template<>
struct Prime_print<1>
{
    enum {prim=0};
    void f() { D<1> d = prim ? 1 : 0; };
};

main()
{
    Prime_print<18> a;
    a.f();
}
```

Template-Metaprogrammierung

- Diese Technik ist als **Template-Metaprogrammierung** bekannt
- Compiler löst Templates schrittweise auf, berechnet dadurch in diesem Fall Primzahlen
- Einbauen gezielter Fehler erzwingt die Ausgabe der Ergebnisse als Fehlermeldung
- Ausgabe beim Compilieren des gezeigten Programms:

```
>g++ prime.C 2>&1 |grep "Prime_print" |grep In\ member
```

```
prime.C: In member function `void Prime_print<i>::f() [with int i = 17]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 13]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 11]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 7]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 5]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 3]':  
prime.C: In member function `void Prime_print<i>::f() [with int i = 2]':
```

Turing-Vollständigkeit

- Todd-Veldhuizen hat 2003 skizziert [3], wie man eine Turing-Maschine über Templates realisiert
- Lässt man den Compiler den Code übersetzen, so geben die Fehlermeldungen den Zustand der Turing-Maschine aus:
- Das Interessante daran: die Turing-Maschine läuft, ohne das Programme ausgeführt werden - der Compiler führt den Template-Code aus!
- Ließe man die Verschachtelungstiefe außer acht, könnte man nicht entscheiden, ob zu einem Programm der Compiler jemals terminiert (Halteproblem)!
- Template-Metaprogrammierung hat aber ernsthaftere Anwendungen
- Insbesondere zur Hocheffizienten Implementierung numerischer Algorithmen hat sie sich durchgesetzt (Bibliotheken Blitz++, POOMA)

Beispiel: eine Warteschlange

- Warteschlangen (queues) sind nützliche Datenstrukturen
- Eine Queue ist eine FIFO-Datenstruktur (first in, first out): was zuerst reingeht, kommt zuerst wieder heraus (analog zu Stacks, die LIFO-(last in, first out)-Datenstrukturen sind)
- Das lässt sich z.B. mit Listen einfach implementieren
- Um die besonderen Bedingungen (FIFO) durchzusetzen, unterstützt die Queue einige Operationen nicht, die Container anbieten (Einfügen/Entfernen an beliebiger Stelle z.B.)
- Queues sind also keine Modelle von Container

- Unterstützte Operationen
 - **push** neuen Wert einfügen
 - **pop** nächsten Wert entfernen
 - **front** nächster Wert
 - **back** letzter Wert

Beispiel: eine Warteschlange

Anforderungen

- **Generizität**
- Für unterschiedliche Anwendungen können unterschiedliche Datenstrukturen effizienter sein (z.B. Liste vs. Deque)
⇒ Queue-Implementierung sollte intern **unterschiedliche Container** verwenden können (Default sollte aber `std::deque` sein)
- Soweit möglich sollte die Implementierung **kompatibel zum STL-Sequence-Konzept** sein
- STL-Konzepte: **Assignable, DefaultConstructible, EqualityComparable, LessThanComparable**

Queue - Implementierung

Anforderungen

- Generizität
- austauschbare Container

1. Versuch:

```
template <typename T,  
         typename Container = Deque<T> >  
class Queue  
{...  
    protected:  
    Container data_  
};
```

Problem:

Explizite Änderung des Containers erfordert die Angabe des selben Typs zweimal, Konstruktion unpassender Kombinationen möglich:

```
Queue<int> q1; // OK  
Queue<int, list<float> > q2; // Unsinnig, aber möglich!
```

Queue - Implementierung

2. Versuch:

```
template <typename T,  
        template <typename> class Container  
        = std::deque>  
class Queue  
{...  
    protected:  
    Container<T> data_;  
    // Erst hier kommt jetzt Template-Argument für den Container!  
};
```

Funktioniert wie gehofft:

```
Queue<int> q1; // OK  
// Queue<int, list<float> > q2; // Compiliert nicht mehr!  
Queue<int, list> q2; // Geht.
```

Queue - Implementierung

Anforderungen:

- DefaultConstructible
- Assignable
- LessThanComparable
- EqualityComparable

```
template <...> class Queue
{
    public:
    Queue() : data_() {}
    Queue(const Queue& q) : data_(q.data_) {}
    Queue& operator = (const Queue& q);
    bool operator == (const Queue& q) const;
    bool operator < (const Queue& q) const;
    ...
}
```

Queue - Implementierung

Anforderung

- Möglichst kompatibel zum Konzept Sequence (Verfeinerung von ForwardContainer)

Notwendig

- `empty()`, `size()`
- `front()`, `back()`
- Typedefs (`value_type`, `reference`, ...)

Nicht unterstützte Teile von Sequence

- `insert(...)`, `erase(...)`
- `resize()`
- `begin()` / `end()`: würde über Iterator Zugriff auf beliebige Elemente erlauben!
- Keine Typedefs für Iteratoren

Queue - Implementierung

```
template <...> class Queue
{
    ...
public:
    typedef          Container<T> container_type;
    typedef typename container_type::value_type value_type;
    typedef typename container_type::size_type size_type;
    typedef typename container_type::reference reference;
    typedef typename container_type::const_reference
        const_reference;
```

- Typdefinitionen können vollständig vom eingebetteten Container übernommen werden
- Definitionen erfordern zusätzliches Schlüsselwort **typename** um klarzustellen, dass es sich um Typen handelt

Queue - Implementierung

```
...
size_type empty() { return data_.empty(); }
bool empty() const { return data_.empty(); }
size_type size() const { return data_.size(); }
reference front() { return data_.front(); }
const_reference front() const { return data_.front(); }
reference back() { return data_.back(); }
const_reference back() const { return data_.back(); }
void push(const_reference v) { data_.push_back(v); }
void pop() { data_.pop_front(); }
```

- Auch die passenden Methoden des Containers werden einfach weitergereicht
- Methoden `push_back`, `pop_front` werden zu den Methoden `pop/back` der Warteschlange
- Dadurch geht leider `push_back` verloren - wir können also keinen `back_inserter` verwenden! \Rightarrow zusätzliche Methode `push_back`

Queue - Implementierung

```
template <...> class Queue
{
    ...
    void push_back(const_reference v) { data_.push_back(v); }
    ...
};

...
std::vector<int> v(50, 5);
Queue<int> q;
std::copy(v.begin(), v.end(), back_inserter(q));
```

- Hinzufügen der push_back-Methode erlaubt so den Einsatz von back_inserter

STL-Klasse `std::queue`

- STL definiert eine Template-Klasse `std::queue`, die unsere Anforderungen erfüllt
- Definiert im Header `queue`
- Funktionalität und Interface praktisch identisch zur gerade beschriebenen Klasse `Queue`
- Diese Klasse zählt zu den so genannten **Adapter-Klassen**: sie stellt Funktionalität anderer Container unter einem neuen Interface zur Verfügung
- Unterschiede
 - Operatoren `==` und `<` sind als globale Funktionen definiert
 - Vorteile, falls die `queue` automatisch konvertierbar ist
 - Passung des zweiten Template-Arguments wird auf andere Art sichergestellt
 - Keine `push_back`-Methode

STL-Klasse `std::queue`

```
// nach http://www.sgi.com/tech/stl/stl\_queue.h
template <class _Tp, class _Sequence>
class queue
{
    typedef typename _Sequence::value_type _Sequence_value_type;

    template <class _Tp1, class _Seq1>
    friend bool operator==(const queue<_Tp1, _Seq1>&,
                          const queue<_Tp1, _Seq1>&);

    template <class _Tp1, class _Seq1>
    friend bool operator< (const queue<_Tp1, _Seq1>&,
                          const queue<_Tp1, _Seq1>&);

public:
    typedef typename _Sequence::value_type      value_type;
    typedef typename _Sequence::size_type       size_type;
    typedef          _Sequence                  container_type;
};
```

STL-Klasse `std::queue`

```
typedef typename _Sequence::reference reference;
typedef typename _Sequence::const_reference
    const_reference;

protected:
    Sequence c;

public:
    queue() : c() {}
    explicit queue(const _Sequence& __c) : c(__c) {}

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    const_reference front() const { return c.front(); }
    reference back() { return c.back(); }
    const_reference back() const { return c.back(); }
    void push(const value_type& __x) { c.push_back(__x); }
    void pop() { c.pop_front(); }
};
```

Literatur zu diesem Teil der Vorlesung:

- [1] Lippman, Lajoie, Moo, C++ Primer, 4. Auflage, Addison Wesley, 2005
(besonders Kapitel 16)
- [2] Stroustrup, Die C++ Programmiersprache, 4. Auflage, Addison Wesley, 2000 (besonders Kapitel 13)
- [3] Todd Veldhuizen, C++ Templates are Turing Complete, 2003
<http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf>
- [4] Erwin Unruh, Prime number computation, 1994. ANSI
X3J16-94-0075/ISO WG21-462.
<http://www.erwin-unruh.de/Prim.html>

Quotes

C++ is history repeated as tragedy. Java is history repeated as farce.
(Scott McKay)

Using Java for serious jobs is like trying to take the skin off a rice pudding wearing boxing gloves. (Tel Hudson)

Claiming Java is easier than C++ is like saying that K2 is shorter than Everest. (Larry O'Brien)