

Programmieren in C++ (3): Templates — Wir fertigen Schablonen und gehen in Serie

Silke Seehusen, FH Lübeck

Es wird eine Einführung ins Programmieren in C++ gegeben, die an Programmierkonzepten wie Datenabstraktion, objektorientierte Programmierung und Polymorphismus ausgerichtet ist. In diesem dritten Teil wird das Konzept der Templates eingeführt und gezeigt, wie mit Templates parametrisierte Datentypen bzw. generische Klassen implementiert werden können.

Templates

Nachdem in den vorhergehenden Artikeln dargestellt wurde, wie funktionale Abstraktion durch Funktionen, Datenabstraktion durch Klassen und Abstraktion in verschiedenen Anwendungsschichten durch Vererbung in C++ realisiert werden kann, soll in diesem Artikel das sehr mächtige Konzept der Templates in C++ vorgestellt werden. Mit dem Konzept können nicht nur Funktionen und Klassen, sondern Schemata und Schablonen von Funktionen und Klassen wiederverwendet werden.

Der Begriff "template" aus dem Englischen wird am besten mit Schablone übersetzt. Wir benutzen im weiteren "Template" als Fremdwort, da sich dafür kein allgemein akzeptierter Begriff durchgesetzt hat.

Eine einfachste, maximale Schablone

Eine sehr häufig verwendete Schablone ist die der Maximumbestimmung von zwei Objekten beliebigen Typs. In C finden wir dafür meistens ein Makro der Form:

```
#define MAX(a,b) \
    (((a)<(b))?(b):(a))
```

Dieses Makro kann für alle Datentypen verwendet werden, für die der Operator < definiert ist. Da ein Makro durch Textersatz aufgelöst wird, bekommen wir spätestens dann Probleme, wenn wir nicht die Ausführung einer Maximumbestimmung benötigen, sondern die Funktion selbst, wenn diese zum Beispiel als Parameter übergeben werden soll. Außerdem kann Textersatz in größeren Programmen zu Überraschungen führen.

Eine Schablone für eine Funktion wird in C++ als Template definiert:

```
template < class T >
const T& max ( const T& a,
              const T& b)
{ return a < b ? b : a; }
```

Das Template ist mit dem Typ *T* der Argumente der Funktion *max()* parametrisiert. Die Template-Parameter werden hinter dem Schlüsselwort **template** in spitzen Klammern angegeben. Danach folgt eine normale Funktionsdefinition, in der *T* als

normaler Klassenname benutzt wird.

Eine Template-Funktion selbst ist noch keine ausführbare Funktion, es ist eben nur eine Schablone, aus der Funktionen generiert werden können. Soll aus dem oben angegebenen Template eine Funktion generiert werden, müssen alle Parameter des Templates bekannt sein. Dies ist z.B. beim folgenden Aufruf der Fall:

```
int a, b;
while ( cin >> a >> b )
    cout << max(a, b) << endl;
```

Hier wird die Funktion

```
const int& max (const int& a,
               const int& b)
{ return a < b ? b : a; }
```

benutzt, die aus dem obigen Template automatisch bei der Benutzung generiert wird.

Für jede verwendete Template-Parameterkombination wird eine Funktion generiert. Im folgenden werden zusätzlich *const double& max(const double& a, const double& b)* und *const string& max(const string& a, const string& b)* generiert und benutzt:

```
double g = 3.14, h = 2.356;
cout << max(g, h) << endl;

string s = "klein?";
string t = "gross?";
cout << max(s, t) << endl;
```

Ein Template setzt meistens Eigenschaften seiner Parameter voraus. Diese Eigenschaften sind in der Regel der Implementierung des Templates zu entnehmen. Das Template *max()* setzt von dem Typ *T* voraus, daß zwischen zwei Objekten vom Typ *T* der Operator *<* definiert ist und einen booleschen Wert beziehungsweise einen in einen solchen konvertierbaren Wert zurückliefert. Eine Generierung einer Funktion mit einem Typ, für den das nicht gilt, wird zu einem einschlägigen Übersetzungsfehler bei der Generierung führen.

Ein C++-Übersetzer sorgt dafür, daß innerhalb einer Übersetzungseinheit nur jeweils einmal der Code für eine Template-Parameterkombination eines definierten Templates generiert wird. Für nur deklarierte Templates wird natürlich kein Code generiert. Eine Deklaration eines Funktions-Templates wird analog zu einer Funktionsdeklaration angegeben. Eine Deklaration des Templates von *max()* sieht folgendermaßen aus:

```
template <class T>
const T& max ( const T& a,
              const T& b );
```

Wie jedoch das Übersetzungssystem verhindert oder ob es verhindert, daß in verschiedenen Übersetzungseinheiten die gleichen Funktionen generiert werden, läßt die Sprachbeschreibung von C++ offen. Das führt zur Zeit zu unterschiedlichen Lösungen. Eine der schlechten Lösungen bietet *g++*: Es liegt in der Verantwortung des Programmierers oder der Programmiererin. In einem konkreten Projekt muß für dieses Problem einmal eine Abstimmung erzielt werden, und dann ist es wirklich kein Problem mehr, denn der Binder wird auf Unstimmigkeiten schon hinweisen.

Bessere Übersetzungssysteme legen bei der Übersetzung Template-Sammlungen an und sorgen für die einmalige Generierung von genau den benötigten Funktionen und Klassen.

Parametrisierter Vergleich

Dem Template *max()* soll nun der vom Typ *T* vorausgesetzte Vergleich als Parameter übergeben werden können. Solche Funktionen werden in C++ in der Regel in einer Klasse gekapselt, die als Parameter übergeben wird. Dazu führen wir ein weiteres Template ein:

```
template < class T,
          class Compare >
const T& max ( const T& a,
              const T& b,
              Compare comp)
{ return comp(a,b) ? b : a; }
```

Der zweite Template-Parameter ist ein Vergleicher (*comparator*), eine Klasse, deren Elementoperator () eine Vergleichsfunktion ist. Eine solche Klasse wird zum Vergleich von ganzen Zahlen folgendermaßen definiert:

```
class IntLess
{ public:
  bool operator()
    (const string& x,
     const string& y)
    const
  { return x < y; }
};
```

Ein Vergleicherojekt wird mit

```
IntLess intless;
```

erzeugt und durch

```
cout << max(a, b, intless);
```

benutzt. Die beiden Funktions-Templates für die Maximumbestimmung, die sich in den Template-Parametern unterscheiden, können nebeneinander definiert und benutzt werden. Die aktuellen Parameter beim Aufruf legen fest, aus welchem Template die entsprechende Funktion generiert wird.

Template für einen Vergleicher

Der Vergleicher wird in der Regel auch aus einem Template, einem Klassen-Template, generiert, z.B. aus:

```
template <class T>
class less
{ public:
  bool operator()(const T& x,
                 const T& y)
    const{ return x < y; }
};
```

Ein Klassen-Template wird wie eine Klasse definiert, wobei wie bei einem Funktions-Template das Schlüsselwort *template* mit den Template-Parametern in spitzen Klammern vorangestellt wird.

Aus einem Klassen-Template können direkt keine Objekte, sondern "nur" Klassen generiert werden. Bei der Generierung einer Klasse werden die aktuellen Template-Parameter hinter dem Template-Namen in spitzen Klammern angegeben. Z.B. wird durch

```
less<double> lessdouble;
```

die Klasse *less<double>* generiert und das Objekt *lessdouble* der generierten Klasse erzeugt. Benutzt wird dieses Objekt in:

```
cout<< max(g, h, lessdouble);
```

Das Minimum erhalten wir unter Verwendung des Klassen-Templates:

```
template <class T>
class greater
{ public:
    bool operator()(const T& x,
                   const T& y)
        const{ return y < x; }
};
```

Die kleinere von zwei Zeichenketten erhalten wir jetzt durch:

```
greater<string> greatstring;
cout<< max(s, t, greatstring);
```

Objekte wie *lessdouble* und *greatstring* werden auch **Funktionsobjekte** im vorgeschlagenen Standard der C++-Bibliothek [1] genannt. Die oben angegebenen Templates dafür finden sich in ähnlicher Form in der vorgeschlagenen Bibliothek unter *<functional>*.

Doch können jederzeit Vergleichobjekte maßgeschneidert werden, z.B. als reiner Längenvergleich von Zeichenketten:

```
class CmpLength
{ public:
    bool operator()
        (const string& x,
         const string& y)
        const
    { return  x.length()
              < y.length(); }
};
CmpLength cmlength;
cout << max(s, t, cmlength);
```

Die entwickelte Maximumfunktion hat schon alle Eigenschaften, die im Hinblick auf Parametrisierbarkeit von einem einfachen Sortierverfahren verlangt werden.

Sortieren zum letzten

Der Algorithmus eines Sortierverfahrens ist eine Schablone, nach der ein Feld von Objekten beliebigen Typs sortiert werden kann. Als Beispiel diene der Quicksort *qsort()*, der ein Feld *v[]* von Elementen im Bereich *v[low]..v[high]* sortiert. Der Typ *ITEM* der zu sortierenden Elemente ist der erste Template-Parameter. Eine auf Elementen des Typs *ITEM* definierte Ordnung wird durch den zweiten Template-Parameter, den Vergleich *Compare*, festgelegt.

```
template < class ITEM,
           class Compare >
void qsort ( ITEM v[],
            int low, int high,
            Compare comp )
{
    if (low < high)
    { int from = low;
      int to   = high;
      ITEM x = v[low/2+high/2];
      do
      { while(comp(v[low],x))
          ++low;
```

```

        while(comp(x,v[high]))
            --high;
        if (low <= high)
        {
            ITEM temp= v[low];
            v[low] = v[high];
            v[high] = temp;
            ++ low, -- high;
        }
    } while ( low <= high );
    qsort(v,from,high,comp),
    qsort(v,low,to,comp);
}
}

```

Wegen der besseren Übersichtlichkeit ist die rekursive Variante des Quicksort gewählt. Damit der Quicksort auch einen vorgelegten Vergleichsoperator hat, wird zusätzlich folgendes Template eingeführt:

```

#include <functional>

template < class ITEM >
void qsort ( ITEM v[],
            int low, int high )
{
    less<ITEM> comp;
    qsort(v, low, high, comp);
}

```

Es können jetzt alle Felder von Elementen sortiert werden, wobei auf den Elementtypen ein entsprechender Vergleichsoperator wie oben definiert ist. Wichtig ist wie immer, daß die Zuweisung und der Kopierkonstruktor vernünftig definiert sind. Aber das ist seit dem ersten Artikel dieser Reihe selbstverständlich. Beispiele für die Benutzung des Quicksort:

```

const int MAXN = 1000;
int      intZahl [MAXN];
int      n; // Anzahl Zahlen
// ...
qsort(intZahl, 0, n-1);

double   dZahl[MAXN];
// ...
greater<double>
    greaterdouble;
qsort ( dZahl, 0, n-1,
        greaterdouble );

string   Wort [MAXN];
// ...
qsort(Wort, 0, n-1);

```

Das Quicksort-Template bietet eigentlich nicht mehr Funktionalität als die Funktion *qsort()* aus der C-Standard-Bibliothek. Doch wer mehrfach versucht hat, Studierende zu überzeugen, den Quicksort aus der C-Bibliothek zu nehmen statt eine eigene Sortierfunktion zu schreiben, kann verstehen, warum ein Template wesentlich besser ist, es ist wesentlich einfacher zu benutzen.

Entwicklung eines Templates

Beim Quicksort ist gut nachzuvollziehen, wie das Template entwickelt wird. Als erstes wird die Funktion ohne Template implementiert und getestet. Das bedeutet im obigen Beispiel, daß die Zeile, die mit *template* beginnt, durch folgende Zeile ersetzt wird:

```
typedef int ITEM;
```

Damit kann das Sortieren von Feldern ganzer Zahlen getestet werden. Ein weiterer Test mit einer anderen Typdefinition, z.B. Zeichenketten, bietet sich dann an.

Im nächsten Schritt wird das Template definiert, siehe Beispiel. Dieses Template sollte dann für "gängige" Typen getestet werden. Für alle erdenklichen Typen kann es nicht getestet werden. Ein Template ist eine Schablone, und alle durch eine Schablone erzeugbaren Dinge können nicht getestet werden. Bei einem Sortierverfahren käme allerdings ein formaler Korrektheitsbeweis des Algorithmus in Frage, da er überschaubar ist. Offen bleibt in der Praxis dennoch, wie die Voraussetzungen an die aktuellen Template-Parameter überprüft werden.

Die Vorgehensweise, daß zunächst eine Funktion für eine ganz bestimmte Anwendung mit festgelegten Typen entwickelt und dann die Funktion verallgemeinert wird, ist sehr typisch für die Entwicklung von Templates und für die Entwicklung von wiederverwendbarer Software im allgemeinen. Diese Vorgehensweise ist auch für die Entwicklung von Klassen-Templates geeignet.

Klassen-Templates

Die Funktions-Templates sind eine sehr wichtige Art von Templates. Noch stärker wird das Konzept in der Anwendung auf Klassen. Ein einfaches Klassen-Template wurde schon als Vergleichler vorgestellt. Klassen-Templates dienen der Realisierung von parametrisierten Datentypen bzw. generischen Klassen in C++.

Im folgenden sollen als Beispiel einfache Container-Klassen (Behälterklassen) eingeführt werden. Ein **Container** ist ein Objekt, das der Speicherung anderer Objekte dient, z.B. eine Liste, ein Feld oder eine Warteschlange. Mit einer Container-Klasse wird der Typ eines solchen Objekts definiert. Da es Listen von Personen, Listen von Parteien und Fahndungslisten gibt, ist eine Container-Klasse für eine Liste ein typischer Kandidat für ein Template.

An dieser Stelle sei eine der einfachsten, übersichtlichsten Container-Klassen eingeführt, der Stapel, auch Keller und Englisch "stack" genannt. Die Template-Klasse *STACK*<class *ITEM*> ist in Abbildung 1 definiert.

```
template<class ITEM> class STACK
{
private:
    unsigned int _atop;           // Index ueber top
    unsigned int _maxsize;       // max. Stapelgroesse
    ITEM *       _stack;         // Feld von Elementen
public:
    STACK( unsigned int mySize = 100 )
        { _maxsize = mySize;
          _stack = new ITEM[_maxsize];
          if ( !_stack ) _maxsize = 0;
          _atop = 0;
        }
    virtual ~STACK()             { delete [] _stack;
    void     push ( const ITEM& );
    void     pop  ()             { assert ( _atop > 0 );
                                --_atop;
    const ITEM& top () const     { assert ( _atop > 0 );
                                return _stack[_atop-1];
    ITEM&     top  ()             { assert ( _atop > 0 );
                                return _stack[_atop-1];
    bool      empty() const     { return ( _atop == 0 );
    unsigned int size () const   { return _atop;
};

template<class ITEM>
void STACK<ITEM>::push( const ITEM& item )
```

```

{  assert ( _atop < _maxsize );
   _stack[_atop++] = item;
}

```

Abbildung 1: Template-Klasse STACK

Der Typ der Objekte, die ein solcher Stapel verwaltet, wird erst bei der Generierung eines Stapels festgelegt. Er wird als aktueller Template-Parameter in spitzen Klammern hinter der Template-Klasse angegeben. Im folgenden Beispiel werden Blumennamen auf einen Stapel gelegt:

```

STACK<string> Blume;
    // Stapel in
    // Standardgroesse
Blume.push("Rose");
Blume.push("Nelke");
while ( ! Blume.empty() )
{  cout << Blume.top();
   Blume.pop();
}

```

In Abbildung 1 ist somit ein Stapel definiert, der für die Verwaltung von Objekten beliebigen Typs geeignet ist. An diesen Typ werden Anforderungen gestellt, die aus der Abbildung 1 abgeleitet werden können. Es muß ein parameterloser Konstruktor existieren, der Kopierkonstruktor und die Zuweisung müssen sinnvoll definiert und der Operator [] sollte, wenn überhaupt, bedeutungserhaltend überdefiniert sein. Dies sind Anforderungen, die wir an jede Klasse stellen, die einen Typ repräsentiert, von dem Objekte gebildet werden können.

Der Stapel kann beliebig komplexe Objekte verwalten. Wir verwenden die Klasse *PERSON* aus dem letzten Artikel [2] wieder:

```

STACK<PERSON> Kunde(200);
    // max. 200 Kunden
PERSON Ida("Ida", "Luebeck");
Kunde.push(Ida);
PERSON p;
while ( cin >> p )
    Kunde.push(p);
cout << Kunde.top();

```

Aus einem Klassen-Template werden beliebig viele Klassen generiert. In den beiden obigen Beispielen sind *STACK<int>* und *STACK<PERSON>* generierte Klassen, wie Abbildung 2 veranschaulicht.

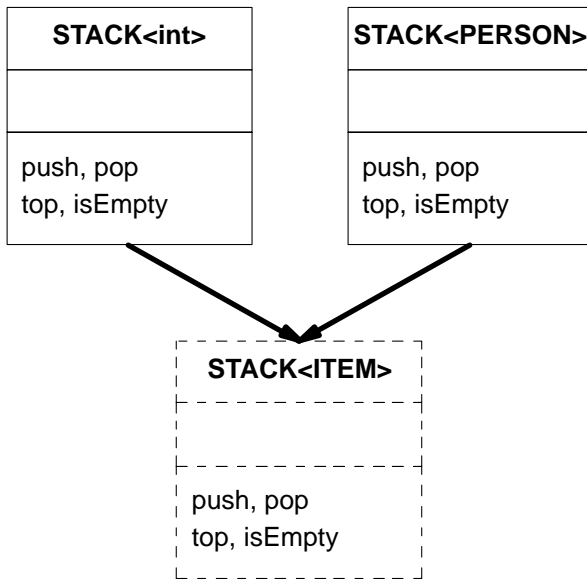


Abbildung 2: Aus dem Template *STACK* generierte Klassen

Die Syntax der Implementierung einer Element-Funktion, die außerhalb der Definition des Klassen-Templates steht, ist etwas gewöhnungsbedürftig, wie das Beispiel `STACK<ITEM>::push()` in Abbildung 1 zeigt. Jede Elementfunktion ist letztendlich ein Funktions-Template und muß entsprechend definiert werden. Desweiteren ist zu beachten, daß die Funktion `push()` zur Klasse `STACK<ITEM>` gehört.

Mit Klassen-Templates kann der alte Traum der adäquaten Implementierung von parametrisierten abstrakten Datentypen verwirklicht werden.

Ein allgemeineres und wesentlich besser implementiertes Klassen-Template für `stack` als hier angegeben findet sich in der vorgeschlagenen C++-Standard-Bibliothek [1]. Steht diese Bibliothek als Standard zur Verfügung, müssen wir für die gängigen Container-Klassen wie Mengen, Listen, Warteschlangen, Stapel und Abbildungen keine eigenen Templates mehr entwickeln. Das tun wir nur noch zur Übung!

Abbildungen (maps)

Abbildungen (maps) sind sogenannte **assoziative Container**, die einen Zugriff auf Daten über Schlüssel zur Verfügung stellen. Eine Abbildung bildet den Schlüssel auf abgespeicherte Daten ab.

Eine typische Anwendung ist die Verwaltung von Beschäftigten einer Firma, die jeweils eine eindeutige Personalnummer besitzen und über die noch weitere Daten abgespeichert werden sollen. Die Personalnummer wird hier als Schlüssel verwendet.

Der assoziative Container wird durch das Template

```
template< class Key, class T,
          class Compare=less<Key> >
class Map
```

zur Verfügung gestellt, siehe Abbildung 3.

```
template< class Key, class T, class Compare=less<Key> >
class Map
{
public:
    Map();
    virtual ~Map();
```

```

bool    insert (const Key &, const T & );
bool    erase  (const Key&);
typedef Suchbaum<Key,T,Compare>::Iterator Iterator;
Iterator find  ( const Key & );
Iterator begin ();
Iterator end   ();
T &        operator [] ( const Iterator & );
private:
    Suchbaum<Key,T,Compare> _baum;
};

```

Abbildung 3: Template Map

Ein Klassendiagramm der Personalverwaltung ist in Abbildung 4 angegeben.

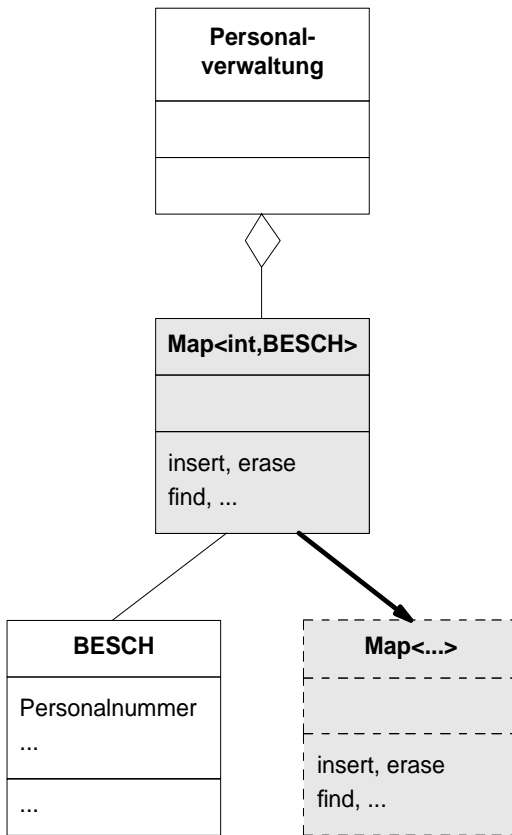


Abbildung 4: Personalverwaltung

Der erste Template-Parameter von *Map* gibt den Typ des Schlüssels an, der zweite Parameter ist der Typ der verwalteten Daten und der dritte Parameter ist ein Vergleich für die Schlüssel zur Definition einer Ordnung, wie er schon beim Sortierverfahren verwendet worden ist. Für den Vergleich ist die Kleiner-Relation voreingestellt. (Auch Templates kennen voreingestellte Parameter.)

Unter Verwendung des Templates *Map* und der Klasse *BESCH* für Beschäftigte aus [2] bauen wir eine Personalbasis auf:

```

Map<int,BESCH> personal;
BESCH pers;
int key;
while ( cin >> pers )
    personal.insert
        ( pers.PersNr(), pers );

```

Einige Beschäftigte werden gelöscht durch:

```

while ( ( cin >> key ) &&
        ( key > 0 ) )

```

```

{
    if ( personal.erase(key))
        cout << "geloescht: "
            << key << endl;
    else
        cout <<"nicht gefunden:"
            << key << endl;
}

```

Iteratoren

Eine häufige Anforderung an einen Container besteht darin, alle im Container abgespeicherten Elemente einmal zu "besuchen", z.B. um alle Elemente auszugeben. Die Elemente eines Feldes $e[]$ werden normalerweise unter Verwendung eines Index besucht:

```

for ( int i=0; i < 10; i++ )
    cout << e[i];

```

Ein dem Index entsprechendes Konzept ist das der Iteratoren auf Containern. Ein Container C stellt den Typ $C::iterator$ zur Verfügung. Ein Objekt i dieses Typs kann mit der Funktion $C::begin()$ initialisiert werden und verweist dann auf das erste Element des Containers. Der Ausdruck $*i$ liefert das Element zurück, auf das i verweist. Es wird $C::end()$ zurückgeliefert, wenn i auf das Element verweist, das dem letzten Element des Containers folgt. Das ist ein gedachtes, fiktives Element.

Auf Iteratoren sind u.a. die Gleichheit ($=$), Ungleichheit ($!=$) und das Inkrement ($++$) definiert. Nach Auswertung des Ausdrucks $i++$ verweist i auf das nächste Element im Container.

Soll eine Namensliste des Personals erstellt werden, so wird ein Iterator über die Abbildung Map benutzt:

```

Map<int,BESCH>::Iterator i;
for ( i = personal.begin();
      i != personal.end();
      i++ )
    cout << (*i).Name()
        << endl;

```

Auch die Suche in einem Container wie der Map liefert einen Iterator zurück:

```

if ( (i = personal.find(key))
      != personal.end() )
    cout << "Die Adresse von "
        << (*i).Name()
        << " lautet:" << endl
        << (*i).Adresse()
        << endl;
else
    cout << "nicht gefunden: "
        << key << endl;

```

Mit einem Container wie Map in der Template-Bibliothek lassen sich recht einfach und überschaubar Anwendungen schreiben, die dynamische Datenstrukturen benötigen. In Abbildung 4 sind die Klassen grau unterlegt, die in der Bibliothek stehen oder aus einem Template der Bibliothek generiert werden. Die Programmentwicklung kann sich auf die Anwendung konzentrieren.

Als "Übung" zeigt Abbildung 3 eine Realisierung einer Abbildung $Key \rightarrow T$, wobei die Schlüssel durch $Compare$ geordnet sind. Die Schlüssel sind eindeutig. Das bedeutet, daß nur eingefügt wird, wenn der einzufügende Schlüssel noch nicht vorhanden ist. Desweiteren ist zu beachten, daß ein Container in der Regel eine

Kopie des übergebenen Elements einfügt.

Die Realisierung zeigt, daß *Map* auf einen binären Suchbaum abgebildet wird. Das Template *Map* benutzt einfach ein weiteres Template, wobei in diesem Fall die Template-Parameter gleich sind. Die Implementierung des binären Suchbaums ist klassisch, außer daß es ein Template ist.

In der vorgeschlagenen C++-Standard-Bibliothek [1] gibt es diverse Container-Templates, u.a. *map*, die sehr ähnlich zu der hier verwendeten *Map* ist. Desweiteren gibt es eine Abbildung, deren Schlüssel nicht eindeutig sein müssen, die sogenannte *multimap*. Entsprechend gibt es für eine Menge *set* und für eine Ansammlung von Elementen, die nicht unterschiedlich sein müssen, das *multiset*. Also sind Implementierungen von Containern wirklich nur noch reine Übungen!

Anwendungsspezifische Templates

Templates werden sehr viel in Bibliotheken benutzt. Aber auch bei der Entwicklung einer konkreten Anwendung können sie sinnvoll sein, zum Beispiel dann, wenn eine mit einem Typ parametrisierte Klasse benötigt wird. Mein letztes Beispiel hierzu war die Implementierung eines Hopfield-Netzes, eine spezielle Art eines neuronalen Netzes, das sich u.a. gut für die Rekonstruktion von teilweise unvollständigen Bildern eignet.

Ein Problem bei der Implementierung eines Hopfield-Netzes besteht darin, daß eine sehr große Matrix von numerischen Werten, die Gewichtematrix, abgespeichert werden muß, auf der sehr viele Operationen ausgeführt werden. (Das Problem besteht eigentlich "nur" darin, daß eine Matrix, die einige Male größer als der real verfügbare Hauptspeicher ist, in der Lernphase zu Laufzeiten von mehreren Wochen führen kann.) Die Größe der Gewichtematrix hängt von der maximalen Größe der Bilder ab, die verarbeitet werden sollen.

Deshalb wird ein solches Hopfield-Netz *HopfieldNetz<class GEWICHT>* in Abhängigkeit vom numerischen Typ der Gewichtematrix definiert, und je nach Bildgröße kann dann mit *float* in *HopfieldNetz<float>*, mit *double* wohl weniger, wohl eher mit *short int* oder einem noch kleiner darstellbaren Typ, der eigens dafür entwickelt wird, gearbeitet werden. Die Algorithmen und die Struktur des Hopfield-Netzes sind von dem numerischen Typ unabhängig, also ist es ein idealer Kandidat für ein Template.

Zusammenfassung

Template-Bibliotheken sind ein Schritt in Richtung Softwarefabrik. Aus vorgefertigten Teilen kann in der Regel einfacher und schneller Software entwickelt werden. Da die vorgefertigten Teile aus einer Bibliothek sorgfältiger entwickelt und getestet sind, kann sogar Software mit weniger Fehlern erzeugt werden. Im einzelnen:

- Klassen-Templates dienen der Implementierung von parametrisierten abstrakten Datentypen.
- Dadurch wird eine weitere Art des Polymorphismus in der Implementierung möglich.
- Aus vorgefertigten Teilen kann einfacher anwendungsspezifische Software entwickelt werden.
- Templates lassen die Schemata von Algorithmen und Klassen besser darstellen und implementieren.
- Templates sind sowohl für die Entwicklung von allgemeinen Bibliotheken als auch für die Entwicklung von anwendungsspezifischen Schablonen geeignet.

Jedoch:

- Die Eigenschaften des Template-Parameters, die im Template verwendet werden, werden in der Definition des Template nicht explizit angegeben. Daß eine Eigenschaft fehlt, wird erst bei der Generierung bemerkt.
- Die Syntax zur Implementierung von Templates ist etwas gewöhnungsbedürftig, aber systematisch.
- Die Generierung von Code für jede verwendete unterschiedliche Template-Parameterkombination kann zu einer großen Code-Menge führen.

Inwieweit Templates der Unterstützung der Anwendung von Entwurfsmuster (*design patterns*) dienen können, bleibt noch zu zeigen.

Templates sind nicht die eleganteste und hoffentlich auch nicht die letzte Art, Schablonen zu entwickeln. Aber zur Zeit kann ich mir Softwareentwicklung ohne Templates kaum noch vorstellen.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server <ftp.informatik.uni-osnabrueck.de> unter `pub/hanser/um`.

Literatur

- [1] A. Koenig (Ed.) *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*, Dokumentnr. X3J16/95-0087, WG21/N0687, 1995.
- [2] S. Seehusen *Programmieren in C++; Teil II: Einfache Vererbung*, **unix/mail** 3/95, 1995.