

# Templates

## Eine kurze Einführung

### Inhalt

1. Grundlagen .....	2
1. 1. Einfache Templates.....	2
1. 1. 1. Funktions-Templates.....	2
1. 1. 2. Klassen-Templates.....	2
1. 2. Two-phase lookup und dependent types.....	3
1. 3. Template-Argumente .....	3
1. 4. Spezialisierung .....	4
1. 4. 1. Vollständige Spezialisierung .....	4
1. 4. 2. Partielle Spezialisierung.....	4
2. Template-Metaprogrammierung .....	5
2. 1. Einführung .....	5
2. 1. 1. Was ist Template-Metaprogrammierung?.....	5
2. 1. 2. Anfänge der Template-Metaprogrammierung.....	5
2. 1. 3. Warum Template-Metaprogrammierung?.....	5
2. 1. 4. Einfache Beispiele .....	6
2. 1. 5. Funktionale Programmiersprache.....	7
2. 2. Ein paar grundlegende Techniken .....	7
2. 2. 1. Traits.....	7
2. 2. 2. Policies .....	8
2. 2. 3. <i>sizeof</i> .....	8
2. 2. 4. Verzweigung .....	9
2. 2. 5. <i>Int2Type</i> .....	9
2. 2. 6. <i>Listen</i> .....	9
2. 3. Expression-Templates.....	10
2. 4. Vergleich mit Präprozessor.....	12

## Grundlagen

### Einfache Templates

#### Funktions-Templates

Ein Funktions-Template ist eine Schablone, aus welcher der Compiler zur Compile-Zeit eine Funktion *instanziert*. Ein Beispiel:

```
template< typename T >
inline T max(T lhs, T rhs)
{
    return lhs>rhs? lhs : rhs;
}
```

Verschiedene Instanzen einer solchen Schablone sind syntaktisch unterschiedliche Funktionen.

Die Template-Argumente zur Bestimmung der Instanz können beim Aufruf weggelassen werden, wenn sie aus den Funktionsargumenten hergeleitet werden können:

```
int f(int i1, int i2)
{
    return max(i1,i2);
}
```

Im Zweifelsfall müssen sie jedoch mit angegeben werden<sup>i</sup>:

```
int f(int i, double d)
{
    return max<int>(i,d);
}
```

#### Klassen-Templates

Ein Klassen-Template ist eine Schablone für eine Klasse.

```
template< class T >
class my_vector {
public:
    typedef T    value_type;
    // ...
};
```

Verschiedene Instanzen eines Klassen-Templates sind unterschiedliche Typen.

Auf die Template-Argumente kann nur innerhalb des Templates zugegriffen werden.

Für Klassen-Templates sind auch Default-Argumente möglich:

```
template< typename T, class TCont = my_vector<T> >
class my_stack {
public:
    // ...
};
```

---

<sup>i</sup> Dies ließe sich vermeiden, wenn man für beide Funktionsparameter verschiedene Template-Parameter angeben würde.:

```
template< typename T1, typename T2 >
inline ? max(T1 o1, T1 o2);
```

Allerdings stellt sich dann die Frage, welcher Type zurückgegeben werden soll. Template-Metaprogrammierung kann dieses Problem lösen, die Lösung ist allerdings nicht trivial.

## Two-phase lookup und dependent types

Templates werden vom Compiler zweimal analysiert. Der erste Parse-Vorgang erfolgt, wenn der Compiler das Template findet, der zweite, wenn das Template mit konkreten Template-Parametern instanziiert wird.

Beim ersten Parse-Vorgang analysiert der Compiler die syntaktische Korrektheit, soweit das möglich ist. Es ist *nicht* möglich für alle Template-Argumente und für alle Typen, die von diesen abhängig sind. Beim Instanzieren eines Templates parst der Compiler den instanziierten Typ und überprüft die vollständige syntaktische Richtigkeit des Funktions-Templates bzw. aller benutzten Elemente eines Klassen-Templates.

Der genaue Typ eines von Template-Argumenten abhängigen Namens (*dependent type*) ergibt sich erst dann, wenn die Template-Parameter, von denen er abhängig ist, eingesetzt wurden. Beim ersten Parsen stellt das ein Problem dar. Wollen wir z.B. in `my_stack` aus dem Beispiel in 0 den Typen `value_type` so definieren

```
typedef TCont::value_type value_type; // Compile-Fehler
```

dann ist `value_type` ein von einem Template-Parameter abhängiger Typ. Abhängig davon, was als `TCont` übergeben wird, kann `TCont::value_type` der Name eines Typ, eines Datenelements, einer Elementfunktion oder eines **enum**-Wertes sein. Beim ersten Parsen des Templates (nicht bei der Instanzierung) könnte der Compiler daher nicht entscheiden, ob eine so einfache Definition wie die von `TCont::value_type` syntaktisch korrekt ist. Laut Standard muß der Compiler in diesem Falle annehmen, daß es sich um den Namen eines Elements (Datenelements oder Elementfunktion) handelt. Falls es sich um einen Datentyp handelt, dann muß der Compiler durch Verwendung des Schlüsselwortes **typename** explizit darauf hingewiesen werden:

```
typedef typename TCont::value_type value_type;
```

Aufgrund dessen kann der Compiler – unter der Annahme, daß innerhalb von `TCont` ein Typ namens `value_type` definiert ist – die grundsätzliche syntaktische Richtigkeit der obigen Typdefinition prüfen.

## Template-Argumente

Bisher wurden in allen Beispielen Typen als Template-Argumente verwendet. Aber auch andere Arten sind möglich. Template-Argumente können sein:

- Typen (*type template parameter*)
 

```
template< class T >
class my_vector;
```
- integrale Konstanten (*non-type template parameter*)
 

```
template< class T, std::size_t N >
class my_array;
```
- Templates (*template template parameter*)
 

```
template< class T, template <typename> class C >
class my_stack {
    typedef C<T> container_type;
    // ...
};
```

Typ-Template-Argumente können entweder (traditionell) durch das Schlüsselwort **class** oder durch das Schlüsselwort **typename** eingeleitet werden. Das hat historische Gründe. Ursprünglich war nur **class** möglich und **typename** noch gar kein Schlüsselwort. Als später aus einem anderen Grund **typename** eingeführt wurde, wurde die Möglichkeit gesehen, das eigentlich nicht ganz passende (es sind ja nicht nur Klassen, sondern auch eingebaute Datentypen als Template-Parameter erlaubt) **class** zu ersetzen. Allerdings

behält man **class** als zusätzliche Möglichkeit weiterhin bei. Beide sind an dieser Stelle syntaktisch völlig äquivalent und beliebig austauschbar.

Wie immer in solchen Fällen, bilden sich verschiedene Vorlieben dafür heraus, wann welches Schlüsselwort eingesetzt wird. Ich bevorzuge die Idee, daß **class** dann eingesetzt wird, wenn das Template-Argument wirklich eine Klasse sein muß (weil z.B. auf Elemente zugegriffen wird) und **typename** dann, wenn es ein beliebiger Typ (also z.B. auch ein **int**) sein kann.

## Spezialisierung

### Vollständige Spezialisierung

Templates lassen sich für bestimmte Typen *spezialisieren*. Dabei wird für eine bestimmte Instanz des Templates eine eigenständige Implementierung angegeben. Diese kann u.U. völlig anders sein, als die Implementierung des primären Templates:

```
template< typename T >
bool less(const T& lhs, const T& rhs) {
    return lhs < rhs;
}
template<>
bool less<char*>(const char* sz1, const char* sz2)
{
    return -1==std::strcmp(sz1,sz2);
}
```

### Partielle Spezialisierung

Bei der teilweisen Spezialisierung wird eine eigenständige Implementierung für eine ganze Gruppe von Instanzen angegeben. Dabei wird ein Muster angegeben, daß die Gruppe beschreibt:

```
// primary template
template< class T >
class my_vector {
    // ...
};
// full specialization
template<>
class my_vector<void*> {
    // ...
};
// partial specialization
template<class T>
class my_vector<T*> : private my_vector<void*> {
    // ...
};
```

Partielle Spezialisierung für Funktions-Templates gibt es nicht. Statt dessen gibt es Überladung. Funktions-Templates können mit anderen Funktions-Templates und mit normalen Funktionen überladen werden.

```
template< typename T > void f(T);
template< typename T> void f(T*);
void f(void*);
```

Nachteil: Es gibt subtile Unterschiede zwischen dem Matchen von Funktionsargumenten bei überladenen Funktionen und der Auswahl von Template-Spezialisierungen. Außerdem lassen sich Funktionen nicht anhand ihres Rückgabetypen überladen, es ist aber u.U. wünschenswert, ein Funktions-Templates anhand des Rückgabetypen zu spezialisieren – was

aber wegen der fehlenden teilweisen Spezialisierung wiederum nicht geht. Es wird deshalb i.A. angenommen, daß die nächste Version des C++-Standards die teilweise Spezialisierung von Funktions-Templates (*function template partial specialization*) erlaubt. Bis dahin kann man Funktions-Templates, die eigentlich eine teilweise Spezialisierung benötigen, mit Hilfe von statische Elementfunktionen von teilweise spezialisierten Klassen implementieren:

```
// primary template
template< typename T >
class F_impl {
    static void f(T) { /*...*/ }
};
// partial specialization
template< typename T >
class F_impl<T*> {
    static void f(T*) { /*...*/ }
};
template< typename T>
inline void f(T o) {F_impl<T>::f(o)}
```

## Template-Metaprogrammierung

### Einführung

#### Was ist Template-Metaprogrammierung?

Template-Metaprogrammierung ist das Ausführen von Algorithmen durch den Compiler *zur Compile-Zeit*.

Es kann gezeigt werden, daß sich sowohl Verzweigung als auch Wiederholung mittels Template-Metaprogrammierung realisieren lassen. Folglich läßt sich auch jeder Algorithmus prinzipiell mittels Template-Metaprogrammierung implementieren.

#### Anfänge der Template-Metaprogrammierung

*Template-Metaprogrammierung in C++ ist ein Zufall.*

- BARTON/NACKMAN, 1994: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, erste Schritte in Richtung Metaprogrammierung
- ERWIN UNRUH (Siemens), 1994: Verdacht, daß C++-Templates *Turing-complete*<sup>i</sup> sind; Beispielprogramm, daß Primzahlen als Fehlermeldungen ausgibt
- TODD VELDHUIZEN, 1995: „*Expression-Templates*“, *Blitz++*, Bibliothek für numerische Mathematik (Felder, Matrizen), Kombination des Abstraktions-Niveau von C++ mit der Performance von handgeschriebenem FORTRAN-Code
- KRZYSTOF CZARNECKI/ULRICH EISENECKER ab 1997: formale Beschreibungen verschiedener Anweisungen (IF<>, WHILE<> etc.)
- ANDREI ALEXANDRESCU, 2001: *Modern C++ Design*, sehr kreativer Umgang mit Templates, der die verschiedensten Techniken mixt

#### Warum Template-Metaprogrammierung?

Template-Metaprogrammierung erlaubt es, Algorithmen zur Compile-Zeit auszuführen. Das hat mehrere Vorteile:

---

<sup>i</sup> Eine Programmiersprache ist *turing-complete*, wenn mit ihr zumindest prinzipiell alle auf einer Turing-Maschine ausführbaren Algorithmen implementierbar sind. Die Voraussetzung dafür ist, daß sie – neben einfachen Anweisungen – auch Verzweigungen und Schleifen unterstützt.

- Wenn ein Teil der Arbeit zur Compile-Zeit erledigt wird, laufen die resultierenden Programme schneller.
- Durch den Einsatz von Templates (statt z.B. virtueller Funktionen) wird das statische Typ-System von C++ genutzt. Der Compiler kann so mehr Typfehler zur Compile-Zeit abfangen.
- Sind diese Algorithmen fehlerhaft, so führt das zu einer Fehlermeldung des Compilers statt eines Laufzeitfehlers.

Template-Metaprogrammierung hat aber auch einen Nachteil: Da der Compiler gegenüber der „normalen“ dynamischen Programmierung einen größeren Anteil an der Arbeit erledigt, benötigt er auch länger dafür.

## Einfache Beispiele

Das Klassen-Template `EqualTypes<>` findet heraus, ob seine zwei Template-Argumente vom selben Typ sind<sup>i</sup>:

```
template< typename T1, typename T2 >
struct EqualTypes {
    enum { result = false };
};
template< typename T >
struct EqualTypes<T,T> {
    enum { result = true };
};
```

Dabei ist zu sehen, wie durch (hier partielle) Spezialisierung eine Verzweigung realisiert werden kann. Wichtig: Das Ergebnis (`EqualTypes<T,T>::result`) ist eine Compile-Zeit-Konstante.

`Factorial<>` (nach CZARNECKI/EISENECKER) berechnet die Fakultät einer Ganzzahl:

```
template< unsigned int N >
struct Factorial {
    enum { result = N * CFactorial<n-1>::result };
};
template<>
struct Factorial<0> {
    enum { result = 1 };
};
```

Es benutzt sowohl die Wiederholung (in Template-Metaprogrammierung immer implementiert durch Rekursion) als auch die (immer mittels Spezialisierung implementierte) Verzweigung.

Der Algorithmus unterscheidet sich nur in der Syntax von der typischen rekursiven Laufzeit-Implementierung der Fakultät:

```
unsigned int factorial(unsigned int n)
{
    if( n==0 ) return 1;
    else return n * factorial(n-1);
}
```

`Max<>` (nach CZARNECKI/EISENECKER) liefert den größeren von zwei Ganzzahl-Werten:

---

<sup>i</sup> Statt des `enums` kann auch ein statische integrale Konstante verwendet werden. Da die Möglichkeit, solche Elemente direkt in der Klassendefinition zu initialisieren, aber erst relativ spät in die Sprache aufgenommen wurde, wird stattdessen weiterhin oft der ältere `enum`-Trick verwendet.

```
template< std::size_t N1, std::size_t N2 >
struct Max {
    enum { result = (N1>N2) ? N1 : N2 };
};
```

Man kann es z.B. dafür einsetzen, ein **char**-Array zu deklarieren, das groß genug ist, um die Zeichen eines von zwei anderen beliebig großen Arrays aufzunehmen. (Die Größe eines Arrays muß ja zur Compile-Zeit feststehen.)

```
const char array1[] = "test array";
const char array2[] = "another Test array ";
// this has room for the content of either one of the above
char arrayX[ Max< sizeof(array1), sizeof(array2) >::result ];
```

Abgesehen von der vielleicht etwas ungewohnten Syntax unterscheidet sich der „Aufruf“ von `Max<>` eigentlich nicht von einem Aufruf von `std::max()` – allerdings wird sein Ergebnis eben nicht erst zur Laufzeit ermittelt, sondern steht schon zur Compile-Zeit fest.

## Funktionale Programmiersprache

In der Template-Metaprogrammierung sind Klassen-Templates das Compile-Zeit-Pendant zu Funktionen: Man übergibt ihnen (Template-) Parameter und erhält dafür ein oder mehrere Ergebnisse. Parameter und Ergebnisse können Ganzzahlen, Typen oder Templates sein. Dies läßt sich beliebig kombinieren.

Hingegen sind **typedef**-Namen und Ganzzahl-Konstanten „Variablen“. Allerdings können deren Werte nach der Initialisierung nicht mehr verändert werden, weil es keine Zuweisung gibt. (Im C++-Sinne sind sie also eigentlich Konstanten.) Dadurch kann z.B. nicht über Listen iteriert werden. (Das erfordert ja eine Schleifenvariable, die bei jedem Durchlauf verändert wird.) Statt dessen kann man Listen aber rekursiv abarbeiten.

Wer schon mal in einer funktionalen Programmiersprache – wie LISP – programmiert hat, dem wird das Prinzip Rekursion statt Iteration und die Konstanz von Variablen sehr bekannt vorkommen.<sup>i</sup>

## Ein paar grundlegende Techniken

### Traits

Traits sind Klassen-Templates, die dazu dienen, (Compile-Zeit-)Informationen über Typen zu speichern. Im Allgemeinen werden sie nur als Template-Parameter verwendet und es werden keine Instanzen von diesen Typen erstellt<sup>ii</sup>. Das Entscheidende an Traits ist, daß die Typen, für die diese Informationen gespeichert werden, dafür nicht verändert werden müssen.

Ein Beispiel ist, anhand eines Iterators den Typ der durch ihn referenzierten Objekte herauszufinden. Dafür gibt es das Klassen-Template `std::iterator_traits<>`:

```
template< typename iter >
typename std::iterator_traits<iter>::value_type find_object(iter, iter);
```

Der Sinn ist hier<sup>iii</sup>, daß der Iterator-Typ u.U. ja auch ein einfacher Zeiger sein kann. Da dies kein Klassen-Typ ist, kann man die notwendigen Informationen nicht als gekapselten Typen (*nested type*) innerhalb des Iterators hinterlegt werden kann.

<sup>i</sup> Das gleiche gilt übrigens auch für die für LISP so typischen „Klammergebirge“ – nur, daß es sich in der Template-Metaprogrammierung um spitze Klammern handelt, während es in LISP runde Klammern sind.

<sup>ii</sup> Das setzt voraus, daß alle Elemente **static** sind.

<sup>iii</sup> Hinweis: Der Algorithmus selbst wäre natürlich nicht unbedingt semantisch sinnvoll. (Was liefert er, wenn sich das gesuchte Objekt nicht in der übergebenen Sequenz befindet?) Aber als syntaktisches Beispiel ist er gut genug.

## Policies

Policies sind Klassen-Templates, die dazu dienen, Verhalten auszulagern. Dazu wird das auszulagernde Verhalten in Typen und (statischen) Elementfunktionen von Policies implementiert. Diese Policies werden als Template-Parameter an Funktions- oder Klassentemplates übergeben, welche dann die jeweilige Funktion aufrufen. Durch Übergabe anderer Policies kann ein anderes Verhalten erreicht werden.

Ein fast schon klassisches Anwendungsgebiet ist eine *Threading Policy*. In einer Anwendung mit mehreren Threads müssen Algorithmen u.U. Vorkehrungen treffen, damit Threads, die auf dieselbe Ressource zugreifen wollen, nicht kollidieren. In Single-Thread-Anwendungen ist das nicht nötig. Zur Unterscheidung könnten die folgenden beiden Policies benutzt werden:

```

struct MultiThreadingPolicy {
    typedef /*...*/    Mutex;
    struct Lock {
        Lock(Mutex& mtx) : mtx_(mtx) {lock(mtx_);}
        ~Mutex()         {unlock(mtx_);}
        Mutex&          mtx_;
    };
};

struct SingleThreadingPolicy {
    class Mutex {};
    struct Lock {
        Lock(Mutex&)    {}
        ~Mutex()        {}
    };
};

```

Ein Algorithmus könnte jetzt so aussehen:

```

template< class ThreadingPolicy >
void f(typename ThreadingPolicy::Mutex& mtx)
{
    // ...
    if( needToTouchThreadSensibleData() ) {
        typename ThreadingPolicy::Lock lock(mtx); // lock mutex
        // thread-safe section
    }
    // ...
}

```

Wird dieser Algorithmus mit `SingleThreading` als Policy instanziiert, dann expandiert die mit „lock mutex“ kommentierte Zeile zu leerem **inline**-Code, den der Optimierer des Compilers entfernen wird. Das gleiche gilt für den impliziten Aufruf des Destruktors von `lock`. Wird er mit `MultiThreading` instanziiert, so verriegelt er Thread-sensible Daten vor dem Zugriff.

**sizeof**

Der Operator **sizeof** ermittelt zur Compile-Zeit die Größe eines Ausdruckes oder Typs. Dazu muß er natürlich in der Lage sein, zur Compile-Zeit den Typ eines beliebigen Ausdruckes zu bestimmen. Leider gibt es keinerlei Möglichkeit, direkt an diese Information heranzukommen<sup>i</sup>. Ein Ausweg ist, aus der Größe des Ausdruckes Rückschlüsse auf dessen Typ zu ziehen. Dabei ist es sehr hilfreich, daß **sizeof** einen ihm übergebenen Ausdruck *nicht* auswertet<sup>ii</sup> und daß sein Ergebnis zur Compile-Zeit feststeht. Eine typische Anwendung

<sup>i</sup> Dementsprechend gibt einen Antrag, einen Operator **typeof** in den Standard aufzunehmen, der genau dies ermöglicht.

<sup>ii</sup> Der Ausdruck **sizeof**( a+=b ) ruft niemals **operator+=**( a , b ) auf und verändert daher a nicht!

für **sizeof** ist es, zur Compile-Zeit herauszufinden, ob ein Typ in einen anderen umgewandelt werden kann (nach ALEXANDRESCU):

```
template< class T, class U >
class Conversion {
    typedef char small;           // sizeof(small) == 1
    class big { char dummy[2]; }; // sizeof(big) != 1
    static small test(U);        // could be called with 'T' or 'U'
    static big test(...);       // could be called with anything
    static T makeT();           // would deliver a 'T'
public:
    enum { exists = sizeof(test(makeT()) == sizeof(small) ) };
};
```

Wichtig ist hier, daß die Funktionen `test()` und `makeT()` gar nicht implementiert sind. Sie werden auch niemals aufgerufen.

## Verzweigung

Die Verzweigung aus `Factorial<>` läßt sich z.B. so generalisieren:

```
template< bool Cond, typename Then, typename Else >
struct If;
template< typename Then, typename Else >
struct If<true, Then, Else > {
    typedef Then Result;
};
template< typename Then, typename Else >
struct If<false, Then, Else > {
    typedef Else Result;
};
```

Damit können wir z.B. schreiben:

```
typedef If< sizeof(int)<sizeof(long), long, int >::Result IntType;
```

Jetzt ist `IntType` **long**, wenn **long** größer als **int** ist, ansonsten ist es **int**.

## Int2Type

Es kommt in der Template-Metaprogrammierung immer wieder vor, daß verschiedene Ganzzahlen als verschiedene Typen repräsentiert werden müssen. Dazu müssen Zahlen in Typen (und Typen wieder zurück in Zahlen) gewandelt werden können:

```
template< int i >
struct Int2Type { enum { value = i }; };
```

Damit stellt `Int2Type<4>` einen anderen Typen dar, als `Int2Type<42>`.

## Listen

Da, wie schon oben erwähnt, Iteration in der Template-Metaprogrammierung nicht möglich ist, sollten Listen-Typen dem Rechnung tragen. Deshalb ist der Listen-Typ der Template-Metaprogrammierung (ähnlich dem in LISP) eine rekursive Liste. Das Ende einer solchen Liste wird durch einen „leeren“ Typ bestimmt:

```
struct Nil {};
```

Hier eine Liste, die nur Typ-Informationen, aber keinerlei Werte speichert:

```
template< typename H, typename T >
struct TypeList {
    typedef H Head;
    typedef T Tail;
};
```

Eine Liste mit drei solchen Elementen sähe dann also so aus<sup>i</sup>:

```
typedef
    TypeList< int, TypeList< float, TypeList< double, Nil > > >
    MyList;
```

Will man zur Compile-Zeit die Länge einer solchen Liste erfahren, hilft folgendes Template:

```
template< class List >
struct Size {
    enum { result = 1 + Size< typename List::Tail >::result };
};
template<>
struct Size<Nil> {
    enum { result = 0; };
};
```

Damit könnte z.B. die Größe der obigen Liste ausgegeben werden:

```
std::cout << Size<MyList>::result << std::endl;
```

## Expression-Templates

Wenn man in der Algebra mit Matrizen rechnet, könnte solcher Code vorkommen (nach VANDEVOORDE/JOSUTTIS):

```
Array<double> a1(1000);
Array<double> a2(1000);
// ...
Array<double> a3 = 1.2*a1 + a1*a2;
```

Nehmen wir an, der entsprechende Multiplikationsoperator ist so deklariert:

```
template< typename T >
Array<T> operator*(const Array<T>&, const Array<T>&);
```

Dann wird der Compiler für den Ausdruck  $1.2*a1 + a1*a2$  Code erzeugen, der etwa so aussieht:

```
Array<T> __tmp1 = 1.2 * a1;
Array<T> __tmp2 = a1 * a2;
Array<T> __tmp3 = __tmp1 + __tmp2;
Array<T> a3 = __tmp3;
```

Dies ist aus zwei Gründen sehr ineffizient:

1. Für die Anwendung eines jeden Operators wird ein temporäres Array erzeugt. Selbst wenn wir annehmen, daß der Compiler kann andere temporäre Arrays als die oben gezeigten<sup>ii</sup> wegoptimieren kann, dann müssen trotzdem noch drei Arrays mit jeweils 1000 Werten erzeugt und gelöscht werden.
2. Für die Anwendung eines jeden Operators werden alle Werte aus den beteiligten Arrays gelesen und wieder geschrieben. Dabei werden also mindestens 6000 Werte gelesen und 4000 Werte geschrieben. Es ist relativ unwahrscheinlich, daß so viele Werte, verteilt auf so viele Objekte, alle in den Cache des Prozessors passen. Wenn nicht, würde das bei aktuellen Prozessorarchitekturen aber z.T. erhebliche Laufzeiteinbußen mit sich bringen.

Das alles kostet Performance und führt dazu, daß dies für ernsthafte numerische Aufgaben nicht eingesetzt werden kann. Stattdessen müßte so etwas von Hand geschrieben werden:

```
for( Array<double>::size_type idx=0; idx<a1.size(); ++idx )
    a3[idx] = 1.2*a1[idx] + a1[idx]*a2[idx];
```

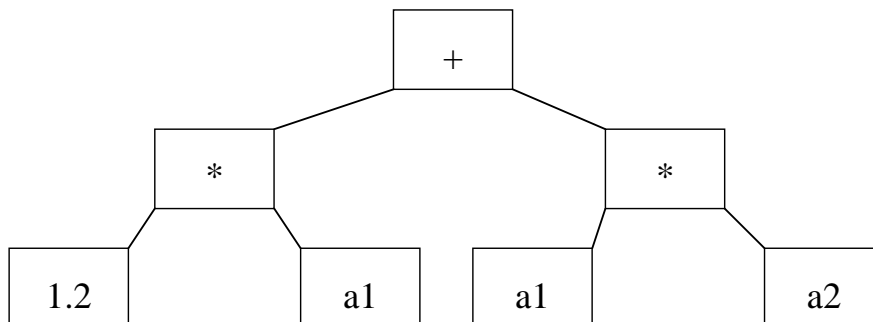
Dieser Code ist sehr effizient. Allerdings ist er schwerer zu schreiben, schwerer fehlerfrei zu schreiben, schwerer zu lesen, schwerer zu verstehen und schwerer zu warten als der obige

<sup>i</sup> In LISP würde übrigens folgender Code verwendet werden: (cons int (cons float (cons double nil))). Natürlich gibt es in LISP weder **int** noch **float** noch **double** (noch überhaupt irgendeinen Datentypen außer Listen). Aber das Prinzip ist dasselbe.

<sup>ii</sup> Also z.B. die vom Operator zurückgegebenen Kopien.

Ausdruck. Expression-Templates bieten nun eine Möglichkeit, weiterhin das ausdrucksstarke  $1.2 * a1 + a1 * a2$  zu schreiben und vom Compiler trotzdem die effektivere **for**-Schleife erzeugen zu lassen.

Dabei gehen wir davon aus, daß der Ausdruck wie folgt aufgebaut ist:



Wir erzeugen dabei für Multiplikation und Addition kein fertiges Ergebnis, sondern Objekte, welche die originalen Werte (1.2, a1, a2) lediglich referenzieren und die oben dargestellte Struktur des Ausdrucks  $1.2 * a1 + a1 * a2$  wiedergeben.

Erst, wenn das Ergebnis wirklich gebraucht wird, wird es berechnet. Zu diesem Zeitpunkt ist aber der gesamte Ausdruck bekannt, weshalb (durch clevere Template-Metaprogrammierung) Code erzeugt werden kann, der dem Code der obigen **for**-Schleife gleicht.

Das für die Addition verwendete Objekt des obigen Baumes könnte z.B. so aussehen<sup>1</sup>:

```

template< typename T, typename Op1, typename Op2 >
class ArrayAdd {
public:
    ArrayAdd(const Op1& o1, const Op2& o2) : op1_(&o1), op2_(&o2) {}
    T operator[](size_t idx) const {return op1_[idx] + op2_[idx];}
    size_t size() const {return (op1_.size() != 0) ? op1_.size() : op2_.size();}
private:
    Op1* op1_;
    Op2* op2_;
};
  
```

Das Multiplikationsobjekt sieht – natürlich bis auf den anderen Operator in der Elementfunktion **operator[]()** – genauso aus.

Außerdem brauchen wir noch ein Objekt für den skalaren Wert:

```

template< typename T >
class ArrayScalar {
public:
    ArrayScalar(const T& v) : v_(v) {}
    T operator[](size_t) const {return v_;}
    size_t size() const {return 0;}
private:
    const T& v_;
};
  
```

Nun definiert man die oben verwendeten Operatoren so, daß sie Instanzen dieser Templates zurückliefern:

<sup>1</sup> Ich habe das Ganze hier *sehr stark* vereinfacht. Es wird daher nicht compilieren. Aber es reicht hoffentlich, um die zugrunde liegenden Prinzipien zu verstehen.

```

template< typename T, typename R1, typename R2 >
inline ArrayAdd operator+( const R1& lhs, const R2& rhs)
{
    return ArrayAdd<T,R1,R2>(lhs,rhs);
}
template< typename T, typename R1, typename R2 >
inline ArrayMul operator*( const R1& lhs, const R2& rhs)
{
    return CArrayMul<T,R1,R2>(lhs,rhs);
}

```

Dann definieren wir noch einen Zuweisungsoperator, der beliebige Ausdrücke nimmt und die darin gespeicherten Operationen ausführt:

```

template< typename T >
Array {
public:
    // ...
    template< typename Expr >
    Array& operator=(const Expr& expr)
    {
        for( /* ... */ )
            this->data_[idx] = expr[idx];
    }
    // ...
};

```

Hierbei wird der `operator [ ] ( )` des Ausdruckes aufgerufen, der den Wert an dieser Stelle des Feldes berechnet.

### Vergleich mit Präprozessor

Code wie dieser ist nicht so unüblich:

```

#if X<Y
    f1();
#else
    f2();
#endif

```

In Abhängigkeit einer bestimmten, statischen (also zur Compile-Zeit feststehenden) Bedingung soll entweder der eine oder der andere Teil des Codes ausgeführt werden. Üblicherweise wird das mit Hilfe des Präprozessors erledigt. Das ist in vielen Fällen ausreichend, in anderen aber nicht.

Der Präprozessor ist ja ein „dummer“ Text-Ersetzer, der von C++ keine Ahnung hat. Deshalb können in der Bedingung auch keinerlei statischen C++-Daten (wie `consts`, `enums` oder `sizeof`) verwendet werden. Es ist auch schwer, mittels des Präprozessors komplexeren Code (als den obigen Vergleich `X<Y`) auszudrücken<sup>i</sup>.  
Template-Metaprogrammierung kann all das bieten.

<sup>i</sup> Allerdings hat boost ([www.boost.org](http://www.boost.org)) eine Präprozessor-Library, die offensichtlich Erstaunliches kann. Da aber der Präprozessor noch weniger für solche Dinge gedacht und geeignet ist, als der Template-Mechanismus des Compilers, ist die Unterstützung bei syntaktischen Fehlern noch geringer als bei Template-Metaprogrammierung.