variadic templates

Templates mit variabler Argumentanzahl?

bislang in C++98 nicht direkt möglich, stattdessen (geniale Idee von A. Alexandrescu): Typlisten

```
template <class T, class U>
struct Typelist {
  typedef T Head;
  typedef U Tail; // U kann selbst wieder Liste sein
};
class NullType {}; // a „non“ type
// z.B.
typedef Typelist<
        char,
        Typelist<
                signed char,
                Typelist<
                        unsigned char,NullType>>> AllCharTypes;
```

variadic templates

+ Makros zur handlichen Erzeugung und Template-Magie für Listenoperationen:

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2)>
...

// Listenlänge:
template <class TList> struct Length;
template <>
struct Length<NullType> {
        enum { value = 0 };
};
template <class T, class U>
struct Length<Typelist<T,U>> {
        enum { value = 1 + Length<U>::value };
};
```

variadic templates

```
// indizierter Zugriff:
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
        typedef Head Result;
};

template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
        typedef typename TypeAt<Tail, i-1>::Result Result;
};
// Wow !
```

unhandlich, unübersichtlich ;-(

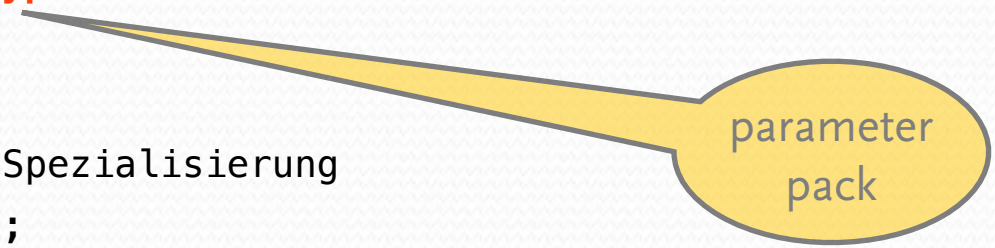
besser: direkte Unterstützung innerhalb der Sprache mit Compilerunterstützung für (Typ-)Listenverarbeitung (ggf. Abb. auf ‚TypeList')

variadic templates

```
template<typename ... Types>
class simple_tuple;

template<> // triviale Spezialisierung
class simple_tuple<> {};

template<typename First, typename ... Rest> // rekursive Spezialisierung
class simple_tuple<First,Rest...>: // Vererbung !!!
    private simple_tuple<Rest...>  {
    First member;
public:
    simple_tuple(First const& f,Rest const& ... rest):
        simple_tuple<Rest...> {rest...}, member{f}   {}
    First const& head() const { return member; }
    simple_tuple<Rest...> const& rest() const { return *this; }
};
```



variadic templates

```
// indizierter Zugriff:
template<unsigned index, typename ... Types>
struct simple_tuple_entry; // allgemein

template<typename First, typename ... Types>
struct simple_tuple_entry<0, First, Types...> // Index 0
{
    typedef First const& type;
    static type value(simple_tuple<First,Types...> const& tuple) {
        return tuple.head();
    }
};
```

variadic templates

```
// indizierter Zugriff (cont.):
template<unsigned index,typename First,typename ... Types>
struct simple_tuple_entry<index,First,Types...> {
   typedef typename simple_tuple_entry<index-1,Types...>::type type;

   static type value(simple_tuple<First,Types...> const& tuple) {
       return simple_tuple_entry<index-1,Types...>::value(tuple.rest());
    }
};

template<unsigned index,typename ... Types>
typename simple_tuple_entry<index,Types...>::type
get_tuple_entry(simple_tuple<Types...> const& tuple) {
    return simple_tuple_entry<index,Types...>::value(tuple);
}
// zumeist in Bibliotheken
```

variadic templates

std::tuple

```
// Anwendung leicht verständlich:

int main() {
    simple_tuple<int,char,double> st {42,'a',3.141};
    std::cout<<get_tuple_entry<0>(st)<<","
             <<get_tuple_entry<1>(st)<<","
             <<get_tuple_entry<2>(st)<<std::endl;
     std::cout<<"sizeof(st)="<<sizeof(st)<<std::endl;

}
```

std::tuple<...> tup;
std::get<i>(tup)

```
$ g++ -std=c++0x -v variadic.cpp
… GNU C++ … version 4.4.3 …
$ a.out
42,a,3.141
sizeof(st)=16
```

variadic templates

std::tuple hat außerdem noch:

```
auto c0 = std::make_tuple(4, 5, 6, 7);
```

und

```
int    v4 = 1;
double v5 = 2;
int    v6 = 3;
double v7 = 4;
std::tie(v4, v5, v6, v7) = c0;

// vi == i
```

[std::tuple schon in TR1 (2003) aber noch ohne variadic templates implementiert]

```
template <typename ... Args>
auto Nto1 (Args... args) -> decltype(make_tuple(args...))
{ return make_tuple(args ...); }
```

variadic templates

Überladung anhand von *parameter packs* ist möglich:

Douglas Gregor: **A Brief Introduction to Variadic Templates** (n2087.pdf) .... ein typsicheres printf in C++

```
template<typename T, typename... Args>
void printf(const char* s, const T& value, const Args&... args) {
    while (s) {
        if (s == '%' && ++s != '%') { // ignore the character that follows
                                      // the '%': we already know the type!
            std::cout << value;
            return printf(++s, args...);
        }
        std::cout << s++;
    }
    throw std::runtime error("extra arguments");
}
void printf(const char* s) {
    while (s) {
        if (s == '%' && ++s != '%')
           throw std::runtime error("missing arguments");
        std::cout << s++;
    }
}
```

rvalue references

non-const Referenzen kann man an *lvalues*, const Referenzen an *lvalues* oder *rvalues*,

An non-const *rvalues* kann man keinerlei Referenzen binden. (Damit niemand den Wert von temporären Variablen ändert, die u.U. verschwinden, bevor der Wert benutzt werden kann)

```
void incr(int& a) { ++a; }
int i = 0; incr(i); // i becomes 1
incr(0); // error: 0 is not an lvalue
```

Aber was ist mit:

```
template<class T> swap(T& a, T& b) { // old style swap
  T tmp(a); // now we have two copies of a
  a = b;    // now we have two copies of b
  b = tmp;  // now we have two copies of tmp (aka a)
}
```

rvalue references

Wenn Kopieren für **T** teuer ist (z.B. string und vector), wird swap ebenfalls teuer (deshalb gibt es in std spezialisierte swap-Versionen)

In Wahrheit soll am **besten gar nichts** kopiert werden – die Werte von **a**, **b**, und **tmp** sollten nur **bewegt** werden

In C++11 kann man "move constructors" and "move assignments" definieren:

```
template<class T>
class vector { // ...
   vector(const vector&);             // copy constructor
   vector(vector&&);                  // move constructor
   vector& operator=(const vector&); // copy assignment
   vector& operator=(vector&&);       // move assignment
};
// note: move constructor and move assignment takes non-const &&
// they can, and usually do, write to their argument
```

&& bezeichnet eine "*rvalue reference*". Eine *rvalue reference* kann man an einen *rvalue* (aber nicht an einen *lvalue*) binden

rvalue references


```
X a;
X f();
X& r1 = a;        // bind r1 to a (an lvalue)
X& r2 = f();      // error: f() is an rvalue; can't bind
X&& rr1 = f();    // fine: bind rr1 to temporary
X&& rr2 = a;      // error: bind a is an lvalue
```

Ist dies das perfekte swap?

```
template<class T> void swap(T&& a, T&& b) { // perfect swap ?
  T tmp = a;
      a = b;
      b = tmp
}
```

NEIN, weil

1. dann keine *lvalues* übergeben werden könnten :-(und
2. alles, was einen Namen hat, bei der Benutzung kein *rvalue* ist :-(

rvalue references


```
// perfect swap:
template<class T> void swap(T& a, T& b) {
  T tmp = std::move(a);   // could invalidate a
      a = std::move(b);   // could invalidate b
      b = std::move(tmp); // could invalidate tmp
}
```

move(x) bewegt **NICHTS**, sondern bedeutet “man kann **x** als *rvalue* verwenden".

Wer bewegt den inneren Zustand des Objektes? Das *move assignment* `T& T::operator=(T&&);` (wenn vorhanden), sonst wird mit dem *copy assignment* kopiert.

move() wiederum kann als template function mit einem rvalue reference Parameter implementiert werden.

rvalue references können auch für *perfect forwarding* benutzt werden.

In der C++11 *standard library* haben alle Container *move constructors* und *move assignments*. Operationen, die Elemente einfügen (wie **insert()** und **push_back()**) haben Versionen, die auf *rvalue references* arbeiten, es gibt die Platzierungsfunktion **emplace()**

rvalue references

```
#include <vector>

class X   {
    std::vector<double> data;
public:
    X():   data(100000) {}                  // lots of data

    X(X const& other):                      // copy constructor
        data(other.data) {}                 // duplicate all that data

    X(X&& other):  // move constructor
        data(std::move(other.data)) {}  // move the data: no copies

    X& operator=(X const& other) {          // copy-assignment
        data=other.data;                    // copy all the data
        return *this;   }

   X& operator=(X && other) {               // move-assignment
        data=std::move(other.data);         // move the data: no copies
        return *this;    }
};
```

rvalue references


```
// kanonische Implementation (nicht immer sie schnellste)
class X   {
    Data data;
public:
    friend void swap(X& x1, X& x2) { swap(a.data, b.data); }

    X(X const& other): data(other.data) {}
    X& operator=(X other) {   // by value: save tmp and if(this==&other)
        swap(*this, other);
        return *this;
    }

    X(X&& other): X{} { swap(*this, other); }
    X& operator=(X && other) {
        swap(*this, other);
        return *this;
    }
};
```