Anwendungen von C++ - Templates

Inhalt:

- *Spezialisierung* von Templates;
- *Traits;*
- *Konfiguration von Klassen* mittels Template-Techniken;
- *Expression - Templates,* Vorstellung des zugrundeliegenden Prinzips;
- *Zeitvergleich* anhand eines Runge-Kutta-Verfahrens, implementiert
  - „zu Fuß“,
  - mit einer „normalen“ Vektor-Klasse,
  - mit der Blitz-Library,
  - mit einer eigenen (Expression -Template-) Vektor-Klasse;
- *Diskussion*

spez1.cpp Rückgabetyp überall `bool` !

```
// Spezialisierung von Funktions-Templates
// A.Voss, (C) 1999, 28.06.99
#include <iostream.h>
#include <string.h>

template <class T>
bool le(const T t1, const T t2)
{ cout<<" (1) "; return (t1<=t2); }

template <class T>
bool le(const T* t1, const T* t2)
{ cout<<" (2) "; return ((*t1)<=(*t2)); }

template <>
bool le(const char* t1, const char* t2)
{ cout<<" (3) "; return (strcmp(t1,t2)<=0); }

void main()
{
  const double  d1(42.1),    d2(12.3);
  const char   *p1("HiHo"), *p2("Oops");

  le(d1,d2);   // (1)
  le(&d1,&d2); // (2)
  le(p1,p2);   // (3)
}
```

spez3.cpp Rückgabetyp unterschiedlich!

```
// Spezialisierung von Funktions-Templates, versch. RC !!!
// A.Voss, (C) 1999, 28.06.99
#include <iostream.h>
#include <string.h>

template <class T>
const T min(const T t1, const T t2)
{ cout<<" (1) "; return (t1<=t2) ? t1 : t2; }

template <class T>
const T min(const T* t1, const T* t2)
{ cout << " (2) "; return ((*t1)<=(*t2)) ? *t1 : *t2; }

template <>
const char* min(const char* t1, const char* t2)
{ cout << " (3) "; return (strcmp(t1,t2)<=0) ? t1 : t2; }

void main()
{
  const double  d1(42.1),    d2(12.3);
  const char   *p1("HiHo"), *p2("Oops");

  cout << min(d1,d2) << endl;   // (1)
  cout << min(&d2,&d1) << endl; // (2)
  cout << min(p1,p2) << endl;   // ???

  // g++ -Wall -pedantic spez3.cpp
  // spez3.cpp:16: template-id `min<>' for
  // `min<>(const char *, const char *)'
  // does not match any template declaration
}
```

spez4.cpp

```
// Spezialisierung von Funktions-Templates, Traits
// A.Voss, (C) 1999, 28.06.99
#include <iostream.h>
#include <string.h>

template <class T>
const T min(const T t1, const T t2)
{ cout<<" (1) "; return (t1<=t2) ? t1 : t2; }

template <class T> struct RC {
  typedef const T rc;
  inline static rc min(const T* t1, const T* t2)
  { cout<<" (2) "; return ((*t1)<=(*t2)) ? *t1 : *t2; }
};

template <> struct RC<char> {
  typedef const char* rc;
  inline static rc min(const char* t1, const char* t2)
  { cout<<" (3) "; return (strcmp(t1,t2)<=0) ? t1 : t2; }
};

template <class T>
RC<T>::rc min(const T* t1, const T* t2)
{ return RC<T>::min(t1,t2); }

void main()
{
  const double  d1(42.1),    d2(12.3);
  const char   *p1("HiHo"), *p2("Oops");

  cout << min(d1,d2) << endl;   // (1)
  cout << min(&d2,&d1) << endl; // (2)
  cout << min(p1,p2) << endl;   // (3)
}
```

kurzer Stop:

- Spezialisierung der Klasse `RC` zur Definition eines vom Template-Parameter abhängigen Rückgabetypes rc;

- Benutzung dieses Types: `RC<Type>::rc`;

- Implementation weiterer Funktionalität (hier `min`) über *statische* Funktionen (bevorzugt `inline`);

- Benutzung dieser Funktion: `RC<Type>::min()`;

spez5.cpp

```
// Traits
// A.Voss, (C) 1999, 28.06.99

template <class T1, class T2> struct BESTOF;

template <class T> struct BESTOF<T,T> {
  typedef T rc;
};

template <> struct BESTOF<double,int> {
  typedef double rc;
};

template <> struct BESTOF<double,long double> {
  typedef long double rc;
};

// ...

void main()
{
  typedef BESTOF<double,int>::rc          dbl1;
  typedef BESTOF<double,long double>::rc  dbl2;

// ...
}
```

spez6.cpp

```
// Traits, cf. Numeric_Limits<>
// A.Voss, (C) 1999, 28.06.99
#include <iostream.h>
#include <math.h>

template <class T> struct NUM_TYPE { };

template <> struct NUM_TYPE<int> {
  static inline bool IsInteger() { return true; }
  static inline int Eps() { return 0; }
// ...
};

template <> struct NUM_TYPE<double> {
  static inline bool IsInteger() { return false; }
  static inline double Eps() { return DBL_EPSILON; }
// ...
};

template <class T>
inline bool IsInteger(const T& d)
{ return NUM_TYPE<T>::IsInteger(); }

void main()
{
  int n(1);
  double d(12.0);

  cout << IsInteger(n) << endl;             // 1
  cout << NUM_TYPE<int>::Eps() << endl;     // 0
  cout << IsInteger(d) << endl;             // 0
  cout << NUM_TYPE<double>::Eps() << endl; // DBL_EPS
}
```

Funktoren, Beispielproblem: *Quicksort*

- funktioniert prinzipiell auf geordneten Mengen;

klassischer Ansatz: `qsort(...)` (stdlib)

- Verbindung zwischen Algorithmus und Elementen: Übergabe eines Funktionszeigers auf Vergleichsfunktion;
- Vorteil: Sortier-Code existiert nur einmal;
- Nachteil: Vergleich zweier Elemente „extern“;

Template-Ansatz: `sort<...>(...)` (STL)

- „Konfiguration“ eines Quicksort-Templates mittels einer Klasse, die Operator zum Vergleich enthält (häufig `Op()`);
- Vorteil: Sortier-Code wird für diese Elemente `inline` generiert, insbesondere Vergleich zweier Elemente *ohne* Aufruf einer Funktion;
- Nachteil: es wird je verwendetem Datentyp Sortier-Code erzeugt;

conf2.cpp

```
// Konfiguration von Templates
// A.Voss, (C) 1999, 28.06.99
#include <iostream.h>

template <class funcClass> class rml {
public:
  typedef typename funcClass::dbl dbl;

  void Solve() { cout<<"RML "<<funcClass::name()<<endl; }
//dbl p(...) { return funcClass::p(...); }
};

template <class Tvalue_type> struct CEuler {
  typedef Tvalue_type dbl;

  static inline char* name() { return "convex"; }
//static inline dbl p(...) { ... }
};

template <class Tvalue_type> struct NCEuler {
  typedef Tvalue_type dbl;

  static inline char* name() { return "non-convex"; }
//static inline dbl p(...) { ... }
};

void main()
{
  rml< CEuler<double> >   rml_c;
  rml< NCEuler<double> >  rml_nc;

  rml_c.Solve();
  rml_nc.Solve();
}
```

Expression-Templates:

Ausgangsproblem: Erzeugung *temporärer Vektor-Objekte* bei Operationen in Vektorklassen

$$\underbrace{\underbrace{\underbrace{v = \underbrace{(v1+v2)}_{Op+(V,V)} * 3.14}_{Op*(V,D)}}_{Op=(V)}}$$

```
// Vektor-Klasse V (mit fester Dim.)
inline V operator+(const V& v1, const V& v2)
{ V v;
  for (int i=v.length(); i--; )
    v[i]=v1[i]+v2[i];
  return v;
}
```

- es existieren verschiedene Optimierungstechniken, um den Aufwand zu *reduzieren,* ganz zu vermeiden ist er nicht;
- die Schleifen werden *in* den Operatoren ausgeführt;

Expression-Templates:

Idee: Anwendung der Operationen jeweils auf das i'te Vektorelement

```
v[i] = (v1[i]+v2[i]) * 3.14   ∀i
```

und Ausführung der Schleife über `i` z.B. in `Op=`

- es existiert nur *eine* „äußere“ Schleife;
- es treten auch temporäre Objekte auf, diese sind jedoch *skalar;*
- benötigt wird ein Mechanismus, der den Ausdruck für Vektoren in einen Ausdruck für Skalare umwandelt;
- dieser skalare Ausdruck muß ausgewertet und „inkrementiert“ werden können;

```
// Vektor-Klasse V (Member Op=(Ex) )
inline V& operator=(Expression ex)
{ V::iterator it=begin(), itE=end();
  for (; it<itE; ++it, ++ex)
    *it = *ex;
  return *this;
}
```

ex_main.cpp

```
// Expression-Templates, analog Todd Veldhuizen
// A.Voss, 28.06.99
#include <iostream.h>
#include <math.h>
#include "ex_all.h"   // Vektor-Klasse und Exp.Templ.

void main()
{
  const int n=3;
  DVec<double> v1(n), v2(n), v3(n);

  v1[0]=1;v1[1]=2;v1[2]=3;  v2[0]=2;v2[1]=3;v2[2]=5;
  cout<<"v1 = ("<<v1<< ")"<<endl;
  cout<<"v2 = ("<<v2<< ")"<<endl<<endl;

// expl1_1/2.h   V+V
  v3 = v1+v2;    cout<<"v3 = v1+v2 = ("<<v3<<")"<<endl;

// expl2.h       V+D
  v3 = v1+2.5;   cout<<"v3 = v1+2.5 = ("<<v3<<")"<<endl;

// expl3.h       Op+Op
  v3 = (v1+v2)-(v1+2.5);
  cout<<"v3 = (v1+v2)-(v1+2.5) = ("<<v3<<")"<<endl;

// expl4_1.h     -Op
  v3 = -(v1+v2); cout<<"v3 = -(v1+v2) = ("<<v3<<")"<<endl;

// expl4_2.h     f(V)
  v3[0]=0.0;v3[1]=M_PI/2.0;v3[2]=M_PI/2.0*3.0;
  v3 = sin(v3);
  cout<<"v3 = sin(0,Pi/2,3/2*Pi) = ("<<v3<<")"<<endl;
}
```

ex_expl1_1.h **Ziel:** `v = v1 + v2`

```
// Expression-Templates, analog Todd Veldhuizen
// A.Voss, 28.06.99

template<class E> class ExpWrap {
public:
  typedef typename E::value_type value_type;

  inline ExpWrap(const E& e) : m_E(e) { }
  inline value_type operator*() const { return *m_E; }
  inline void operator++() { ++m_E; }
private:
  E m_E;
};

template<class EL, class ER, class OP> class ExpBinOp {
public:
  typedef typename OP::value_type value_type;

  inline ExpBinOp(const EL& eL, const ER& eR)
    : m_eL(eL), m_eR(eR) { }
  inline value_type operator*() const
    { return OP::apply(*m_eL,*m_eR); }
  inline void operator++() { ++m_eL; ++m_eR; }
private:
  EL m_eL;
  ER m_eR;
};

template <class T> struct ExpOpAdd {
  typedef T value_type;
  static inline T apply(T a, T b) { return a+b; }
};
```

ex_expl1_2.h **Ziel:** `v = v1 + v2`

```
// Expression-Templates, analog Todd Veldhuizen
// A.Voss, 28.06.99

template <class T>
inline ExpWrap<ExpBinOp<typename DVec<T>::iterator,
                        typename DVec<T>::iterator,
                        ExpOpAdd<T> > >
operator+(const DVec<T>& a, const DVec<T>& b)
{ typedef ExpBinOp< typename DVec<T>::iterator,
                    typename DVec<T>::iterator,
                    ExpOpAdd<T> > ExprT;
  return ExpWrap<ExprT>(ExprT(a.begin(),b.begin()));
}
```

`Op=` **aus** `DVec`

```
template <class A>
inline DVec& operator=(ExpWrap<A> ex)
{ for (DVec::iterator it=begin(); it<end(); ++it, ++ex)
    *it=*ex;
  return *this;
}
```

- `Op+` wird *vor* `Op=` aufgerufen, d.h. der Ausdruck (Baum) wird erstellt und dann ausgewertet;
- Klasse `ExpWrap` zunächst unwichtig;
- Klasse `ExpBinOp` konfiguriert mit *Ptr,Ptr,Op+*
  `Op*` wird zu { `return (*Ptr + *Ptr);` }
  `Op++` wird zu { `++Ptr; ++Ptr;` }

ex_expl2.h **Ziel:** `v = v1 + 2.5`

```
// Expression-Templates, analog Todd Veldhuizen
// A.Voss, 28.06.99

template<class E> class ExpConst {
public:
  typedef E value_type;

  inline ExpConst(const E& e) : m_E(e) { }
  inline value_type operator*() const { return m_E; }
  inline void operator++() { }
private:
    E m_E;
};

template<class T>
inline ExpWrap<ExpBinOp<typename DVec<T>::iterator,
                        ExpWrap<ExpConst<T> >,
                        ExpOpAdd<T> > >
operator+(const DVec<T>& a, const T& b )
{ typedef ExpBinOp< typename DVec<T>::iterator,
                    ExpWrap<ExpConst<T> > ,
                    ExpOpAdd<T> > ExprT;
  return ExpWrap<ExprT>(ExprT(a.begin(),
                              ExpWrap<ExpConst<T> >(b) ));
}
```

- Klasse `ExpBinOp` konfiguriert mit *Ptr,Cnst,Op+*

  `Op*` wird zu { `return (*Ptr + Const);` }

  `Op++` wird zu { `++Ptr;` }

ex_expl3.h **Ziel:** `v = (v1+v2)-(v1+2.5)`

```
// Expression-Templates, analog Todd Veldhuizen
// A.Voss, 28.06.99

template <class T>
struct ExpOpMin {
  typedef T value_type;
  static inline T apply(T a, T b) { return a-b; }
};

template<class A, class B>
inline ExpWrap<ExpBinOp<ExpWrap<A>,
                        ExpWrap<B>,
                        ExpOpMin<
                          ExpWrap<A>::value_type> > >
operator-(const ExpWrap<A>& a, const ExpWrap<B>& b)
{ typedef ExpBinOp< ExpWrap<A>,
                    ExpWrap<B>,
                    ExpOpMin<
                      ExpWrap<A>::value_type> > ExprT;
  return ExpWrap<ExprT>(ExprT(a,b));
}
```

- Klasse `ExpWrap` sinnvoll, damit Operatoren nur für diese Klasse zu schreiben sind;

ex_expl4_1.h **Ziel:** `v = -(v1+v2)`

```
// Expression-Templates
// A.Voss, 28.06.99

template<class EL, class OP> class ExpOneOp {
public:
  typedef typename OP::value_type value_type;

  inline ExpOneOp(const EL& eL) : m_eL(eL) { }
  inline value_type operator*() const
    { return OP::apply(*m_eL); }
  inline void operator++() { ++m_eL; }
private:
  EL m_eL;
};

template <class T> struct ExpOpPreMin {
  typedef T value_type;
  static inline T apply(T a) { return -a; }
};

template<class A>
ExpWrap<ExpOneOp<ExpWrap<A>,
                 ExpOpPreMin<ExpWrap<A>::value_type> > >
inline operator-(const ExpWrap<A>& a)
{
    typedef ExpOneOp<ExpWrap<A>,
                     ExpOpPreMin<
                       ExpWrap<A>::value_type> > ExprT;
    return ExpWrap<ExprT>(ExprT(a));
}
```

ex_expl4_2.h **Ziel:** `v = sin(v3)`

```
// Expression-Templates
// A.Voss, 28.06.99

template <class T>
struct ExpOpSin {
  typedef T value_type;
  static inline T apply(T a) { return sin(a); }
};

template <class T>
ExpWrap<ExpOneOp<typename DVec<T>::iterator,
                 ExpOpSin<T> > >
inline sin(const DVec<T>& a)
{ typedef ExpOneOp<typename DVec<T>::iterator,
                   ExpOpSin<T> > ExprT;
  return ExpWrap<ExprT>(ExprT(a.begin()));
}
```

ex_main.out

```
v1 = ( 1  2  3 )
v2 = ( 2  3  5 )

v3 = v1+v2 = ( 3  5  8 )
v3 = v1+2.5 = ( 3.5  4.5  5.5 )
v3 = (v1+v2)-(v1+2.5) = ( -0.5  0.5  2.5 )
v3 = -(v1+v2) = ( -3  -5  -8 )
v3 = sin(0,Pi/2,3/2*Pi) = ( 0  1  -1 )
```

Expression-Templates: Zusammenfassung

- Expression-Templates wandeln Ausdrücke in skalare Ausdrücke um; dazu wird ein Ausdrucksbaum aus Operator-Objekten aufgebaut;
- alle Operator-Objekte besitzen einen Auswertungs- (`Op*`) und einen Inkrementoperator (`Op++`); die Auswertung der eigentlichen skalaren Operation wird bei optimalem Code `inline` ausgeführt;
- optimal für das Laufzeitverhalten ist eine Elimination der temporären Operator-Objekte;
- die Klasse `ExpWrap` dient dazu, Operatoren nicht für alle Operatorklassen schreiben zu müssen;
- diese Technik ist nicht auf Vektoren beschränkt;

sm_main.cpp Ziel: Funktionen als Parameter

```
// Expression-Templates
// A.Voss, 28.06.99

#include <iostream.h>
#include <math.h>
#include "sm_expl1.h"

template<class A>
void evaluate(ExpWrap<A> f,
              double dStart, double dStep, int nCount)
{ for (int i=0; i<nCount; i++, dStart+=dStep)
    cout<<"x="<<dStart<<" -> f(x)="<<f(dStart)<<"   ";
  cout<<endl;
}

template<class A>
double simpson(ExpWrap<A> f, double a, double b)
{ return (b-a)/6.*(f(a)+4.*f((a+b)/2.)+f(b)); }

void main()
{
  ExpIdentity<double>           id;
  ExpWrap<ExpIdentity<double> > x(id);

  evaluate(x*x + 2.*x + 1.,0.0,1.0,3);  // (x+1)^2

  cout << "I: " << simpson(sin(x)+1.0,0,M_PI) << endl;
}
```

sm_main.out

```
x=0 -> f(x)=1   x=1 -> f(x)=4   x=2 -> f(x)=9
I: 5.23599
```

sm_expl2.h Erweiterung um `Op()`

```
// Expression-Templates
// A.Voss, 28.06.99

template<class E> class ExpWrap {
    E m_E;
public:
  typedef typename E::value_type value_type;
  inline ExpWrap(const E& e) : m_E(e) { }
  inline value_type operator()(value_type d) const
    { return m_E(d); }
};

template<class EL, class OP> class ExpOneOp {
    EL m_eL;
public:
  typedef typename OP::value_type value_type;
  inline ExpOneOp(const EL& eL) : m_eL(eL) { }
  inline value_type operator()(value_type d) const
    { return OP::apply(m_eL(d)); }
};

template <class T> struct ExpIdentity {
  typedef T value_type;
  inline T operator()(T x) const { return x; }
};

template<class E> class ExpConst {
    E m_E;
public:
  typedef E value_type;
  inline ExpConst(const E& e) : m_E(e) { }
  inline value_type operator()(value_type d) const
    { return m_E; }
};
```

Expression-Templates: Zeitbeispiel (!)

- Verfahren: Klassisches RK-Verfahren 4. Ordnung, auf $f' = f$ über Intervall $[0, 1]$ mit $h = 0.001$, d.h. 1000 Schritte;
- compiliert für EGCS mit Optimierung:
  `g++ -O3 -fstrict-aliasing ...`
- die Zeit wird gemittelt aus 1000 Durchläufen;
- der Faktor gibt an: Zeit je Vektorklasse div. durch Zeit des „zu Fuß“-Verfahrens, je Dimension des Vektors (1..9)
- Vektorklassen:

  (1) Vektorklasse mit temporären Objekten,

  (2) Blitz-Library (TinyVector),

  (3) Eigene Expression-Vektorklasse,

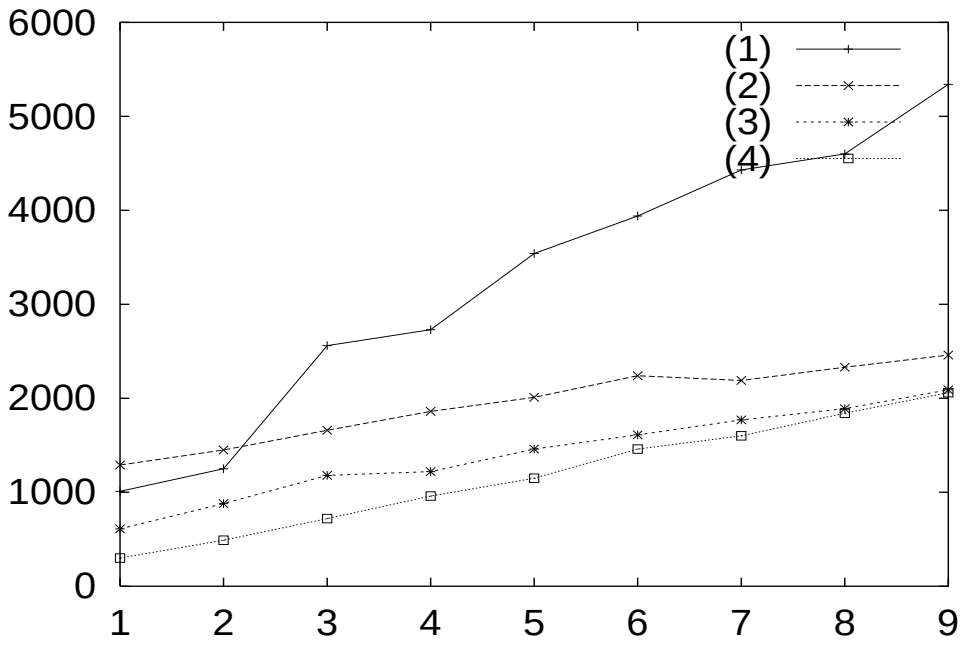
  (4) zu Fuß-Verfahren;

rk_test1.cpp

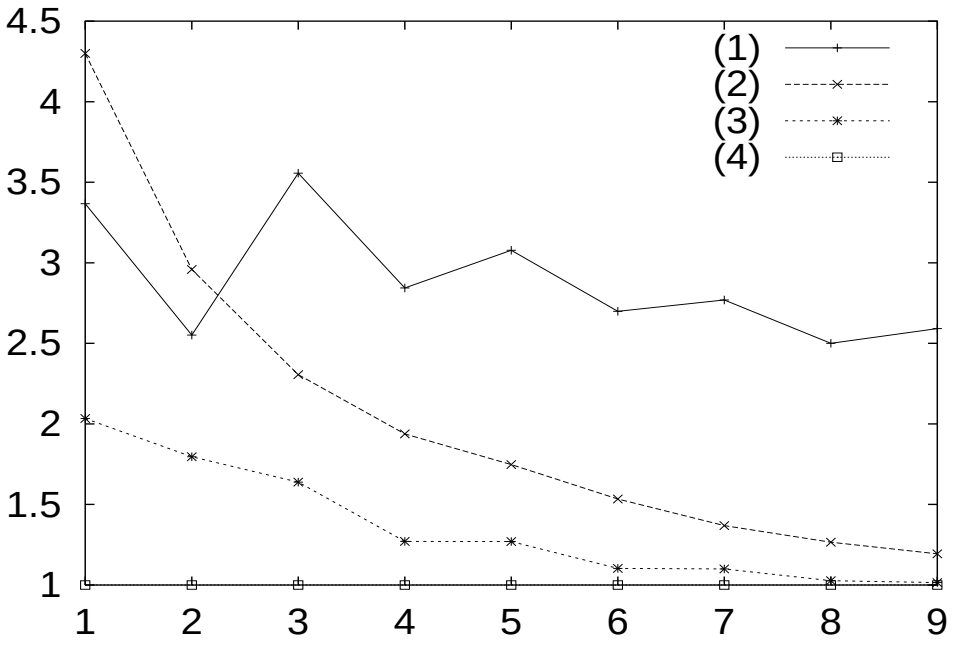
```
// x,y,ki Vektoren jeweils aus getesteter Vektorklasse
for (int i,j,k=1; k<=nLoop; k++) {
  // init x,y
  for (i=0; i<nCnt; i++) {
    f(y,k1);
    f(y+dh*0.5*k1,k2);
    f(y+dh*0.5*k2,k3);
    f(y+dh*k3,k4);
    y=y+dh*(1./6.*k1+1./3.*k2+1./3.*k3+1./6.*k4);
    x=x+dh;
  }
}

// zu Fuss, x,y,ki n-dim. double-Felder
for (int i,j,k=1; k<=nLoop; k++) {
  // init x,y
  for (i=0; i<nCnt; i++) {
    for (j=nDim; j--; ) {
      f(y[j],k1[j]);
      f(y[j]+dh*0.5*k1[j],k2[j]);
      f(y[j]+dh*0.5*k2[j],k3[j]);
      f(y[j]+dh*k3[j],k4[j]);
      y[j]+=dh*(1./6.*k1[j]+1./3.*k2[j]+1./3.*k3[j]+
                1./6.*k4[j]);
      x[j]+=dh;
    }
  }
}
```

Expression-Templates: Zeitbeispiel (!)

Dim - Zeit:



Dim - Faktor:



Expression-Templates: Tuning

- Vorab: Ansatz ist so gut wie die Optimierungsmöglichkeiten des verwendeten Compilers;
- Problem: bei kleinen Vektoren übermäßig hoher Aufwand zum Kopieren der Operator- und Wrapper-Objekte;
- direkt auf Referenzen zu arbeiten funktioniert nicht, da temporäre Objekte beliebig vom Compiler aufgeräumt werden können;
- bei festgelegter Reihenfolge der Auswertung der Operatoren bzw. Argumente einer Funktion, kommt man (fast) ohne Daten in den Objekten aus;
- ein Ansatz zur Verbesserung ist die vorherige Reservierung von Speicher für die temporären Objekte; dies geht auch ohne virtuelle Funktionen; Arbeiten mit *Placement-new*;

smplex_op.h

```
// Expression-Stack-Templates
// A.Voss, (C) 1999, 28.06.99

// V+V
inline friend
const BINOP<EXPTEMP::ePlus,
            const_dbl*,const_dbl*,
            V,EXPTEMP::eBoth>&
operator+(const V& lhs, const V& rhs)
{ return *new (&(g_opList[g_opN++]))
    const BINOP<EXPTEMP::ePlus,
                const_dbl*,const_dbl*,
                V,EXPTEMP::eBoth>
                (lhs.m_d+m_nDim-1,rhs.m_d+m_nDim-1); }

// o+o
template < int O1, class LHS1, class RHS1, int nID1,
           int O2, class LHS2, class RHS2, int nID2>
inline friend
const BINOP<EXPTEMP::ePlus,
            const BINOP<O1,LHS1,RHS1,V,nID1>&,
            const BINOP<O2,LHS2,RHS2,V,nID2>&,
            V,EXPTEMP::eBoth >&
operator+( const BINOP<O1,LHS1,RHS1,V,nID1>& lhs,
           const BINOP<O2,LHS2,RHS2,V,nID2>& rhs)
{ return *new (&(g_opList[g_opN++]))
    const BINOP<EXPTEMP::ePlus,
                const BINOP<O1,LHS1,RHS1,V,nID1>&,
                const BINOP<O2,LHS2,RHS2,V,nID2>&,
                V,EXPTEMP::eBoth >(lhs,rhs); }
```