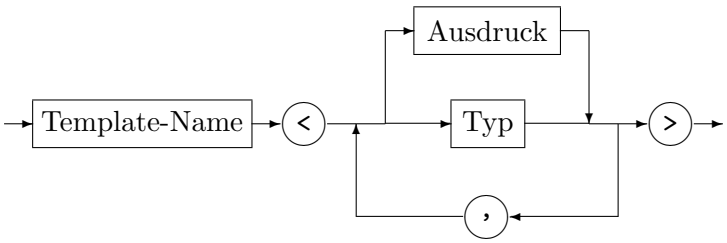
**Template-Auswertung**

Durch Einsetzen eines Typs bzw. eines *konstanten* Ausdrucks in




wird die entsprechende Klasse, Funktion oder Komponentenfunktion erzeugt, wenn ihre Definition tatsächlich benötigt wird (template instantiation).

Die Templateargumente müssen denselben Typ wie die Templateparameter haben, Integerweiterungen finden jedoch statt.

*Bsp.:*

```
#include <iostream>
#include <cmath>
#include <array>
#include <typeinfo>

using namespace std;

// Abgeleitete Klasse myarray erbt Komponenten von array (ausser Konstr./Destr.)
template<typename T, size_t N> struct myarray : array<T,N>
{
   // Standardkonstruktor zur Ausgabe der Typinformation
   myarray() { cout << "myarray<" << typeid(T).name() << "," << N << ">" << endl; }
};

template<typename T, size_t N> T two_norm(const myarray<T,N>& x)
{
  T s = 0;
  for (typename myarray<T,N>::const_iterator pos=x.begin(); pos!=x.end(); ++pos)
    s += (*pos)*(*pos);
  return sqrt(s);
};


int main()
{
  // myarray<float> x = {3,4 };  // unzulaessig, weil Konstruktor nicht geerbt
  myarray<float,2>   x;          // Konstrukt.aufruf erfolgt nur fuer myarray<float,2>
  myarray<double,4> *y;          // Als einzige Fkt. in array/myarray werden ein
  x[0]=3; x[1]=4;                // Konstruktor und eine Komp.funktion operator[] erzeugt
                                 // Insb. wird keine Fkt. twonorm generiert
                                 // (Unix/Linux: Ueberpruefen mit nm)
  return 0;
}
```

*Ausgabe:*

```
myarray<f,2>
```

Bei Funktionen kann die Templateargumentliste entfallen, wenn durch geeignete Wahl eine Funktion erzeugt werden kann, deren Parametertypen zu den Datentypen der eingesetzten Funktionsargumente passen (template argument deduction).

*Bsp.:* `two_norm(x)` statt `two_norm<float,2>(x)`

### Template-Spezialisierung

Wird in einem Template ein Template-Parameter durch einen Template-Ausdruck ersetzt, so entsteht daraus ein Template mit kleinerem Anwendungsbereich. (Template-Spezialisierung) Speziellere Templates haben Vorrang vor allgemeiner definierten Templates. Damit sind dann Sonderbehandlungen für einzelne Datentypen oder Gruppen davon möglich.

*Bsp.: Dekl. für* `vector<bool>`

```
template <class T, class Allocator = allocator<T>> class vector {
  public:
                :
    typedef typename Allocator::size_type size_type;
                :
    explicit vector(size_type n, const T& value = T(),
                    const Allocator& = Allocator());
                :
};


template <class Allocator = allocator<bool>> class vector<bool, Allocator> {
  public:
                :
    typedef typename Allocator::size_type size_type;
                :
    explicit vector(size_type n, const bool& value = bool(),
                    const Allocator& = Allocator());
                :
    void flip();    // Umklappen der Bits
                :
};
```

Auch Funktionstemplates lassen sich spezialisieren.

*Bsp.: Spezialisierung für swap*

```
  template <class T>                      // Definition in <algorithm> (C++98)
  inline void swap (T& a, T& b)
  {
    T tmp = a;
    a = b;
    b = tmp;
  }

  template <class T, class Allocator>     // Definition in <vector>
  inline void swap(vector<T,Allocator>& a, vector<T,Allocator>& b)
  {
    a.swap(b);
  }
```

*Bsp.: Konjugierte Gradienten*

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

template<class T> class matrix;

template<class T> class vektor {    // vektor, nicht vector !!
  private:
    valarray<T> v;
  public:
    vektor<T>(int n=0): v(n) {}
    vektor<T>& operator=(T lambda) { v=lambda; return *this; }
    vektor<T>& operator+=(const vektor<T>& b) { v+=b.v; return *this; }
    vektor<T>& operator-=(const vektor<T>& b) { v-=b.v; return *this; }
    vektor<T> operator+(const vektor<T>& b) const
      { vektor<T> temp(*this); temp.v+=b.v; return temp; }
    vektor<T> operator-(const vektor<T>& b) const
      { vektor<T> temp(*this); temp.v-=b.v; return temp; }
    vektor<T> operator-() const { vektor<T> temp(*this); temp.v=-v; return temp; }
    vektor<T>& operator*=(T lambda) { v*=lambda; return *this; }
    vektor<T> operator*(T lambda)
      { vektor<T> temp(*this); temp.v*=lambda; return temp; }
    vektor<T>& operator/=(T lambda) { v/=lambda; return *this; }
    vektor<T> operator/(T lambda)
      { vektor<T> temp(*this); temp.v/=lambda; return temp; }
    friend inline T skalar(const vektor<T>& a, const vektor<T>& b)
      { return (a.v*b.v).sum(); }
    T& operator[] (int i)       { return v[i]; }
    T  operator[] (int i) const { return v[i]; }
    int dim() const { return v.size(); }
    friend inline vektor<T> operator*(const matrix<T>& a, const vektor<T>& b);
};

template<class T> class myslice_array {   // Hilfsklasse
  private:
    slice s; valarray<T>& v;
  public:
    myslice_array<T>(int i0, int n, int h, valarray<T>& v_):
      s(slice(i0,n,h)), v(v_) {}
    T& operator[](int j) { return v[s.start()+j*s.stride()]; }
};

template<class T> class mycslice_array {   // Hilfsklasse
  private:
    slice s; const valarray<T>& v;
  public:
    mycslice_array<T>(int i0, int n, int h, const valarray<T>& v_):
      s(slice(i0,n,h)), v(v_) {}
    T operator[](int j) { return v[s.start()+j*s.stride()]; }
};
```

```
template <class T> class matrix {
  private:
    int n; valarray<T> v;
  public:
    matrix<T>(int n_=0): n(n_),v(n_*n_) {}
    myslice_array<T>  operator[](int i)       { return myslice_array<T>(i*n,n,1,v);  }
    mycslice_array<T> operator[](int i) const { return mycslice_array<T>(i*n,n,1,v); }
    friend inline vektor<T> operator*(const matrix<T>& a, const vektor<T>& b) {
      int n=a.n;
      vektor<T> temp(n);
      for (int i=0; i<n; ++i) temp[i]=(a.v[slice(i*n,n,1)]*b.v).sum();
      return temp;
    }
    int dim() const { return n; }
};

template <class Matrix, class Vektor, class T>
  void konjgrad(const Matrix& A, const Vektor& b, Vektor& x, T eps)
{
  int n=b.dim(), k;
  Vektor r(n), p(n), q(n);
  r = b-A*x; p = r;
  T alpha, beta, rhoalt=skalar(r,r), rho;

  for (k=0; k<n && rhoalt>=eps*eps; ++k) {
    q = A*p;
    alpha = rhoalt/skalar(p,q);
    x += p*alpha;
    r -= q*alpha;
    rho = skalar(r,r);
    beta = rho/rhoalt;
    p = r+p*beta;
    rhoalt = rho;
  }
  cout << "Iterationszahl: " << k << endl;
  r = A*x-b;
  cout << "Residuum:       " << sqrt(skalar(r,r)) << endl;
}

int main()
{
  int n; cout << "n : "; cin >> n;
  vektor<double> b(n),x(n);
  matrix<double> A(n);
  for (int i=0; i<n; ++i) b[i]=sin(2*M_PI*i/n);
  // Hilbertmatrix (schlecht konditioniert!)
  for (int i=0; i<n; ++i) for (int j=0; j<n; ++j) A[i][j]=1.0/(i+j+1);
  for (int i=0; i<n; ++i) A[i][i]+=1.0;   // A = I+Hilbertmatrix (gut konditioniert)
  konjgrad(A, b, x, 1e-15);
  for (int i=0; i<n; ++i) cout << "x[" << i << "] = " << x[i] << endl;
  return 0;
}
```