C++ bietet eine Vielfalt weiterer Techniken für Templates:

- Templates können auch außerhalb von Klassen für einzelne Funktionen verwendet werden.
- Templates können implizit in Zuge von Überladungen instantiiert werden. Dabei können sie auch in Konkurrenz zu Nicht-Template-Funktionen des gleichen Namens stehen.
- Templates können innerhalb von Klassen definiert werden.
- Templates können neben Typen auch ganze Zahlen, Zeiger, Referenzen oder andere Templates als Parameter haben. Parameter können optional sein.
- Unterstützung von Spezialfällen und rekursive Templates. (Damit erreichen wir die Mächtigkeit einer Turing-Maschine zur Übersetzzeit!)
- Literatur: David Vandevoorde und Nicolai M. Josuttis: *C++ Templates*

```
template<typename T>
inline void exchange(T& v1, T& v2) {
   T tmp(v1); v1 = v2; v2 = tmp;
}
```

- *exchange* ist hier eine generelle Funktion zum Austausch zweier Variableninhalte.
- (Eine entsprechende Funktion namens *swap* existiert in der Standardbibliothek.)
- Die Funktion kann *ohne* Template-Parameter verwendet werden. In diesem Falle sucht der Übersetzer zunächst nach einer entsprechenden Nicht-Template-Funktion und, falls sich kein entsprechender Kandidat findet, nach einem Template, das sich passend instantiieren lässt.

```
  int i, j;
  // ...
  exchange(i, j);
```

```
template<typename R, typename T>
R eval(R (*f)(T), T x) {
   static map<T, R> cache;
   if (cache.find(x) != cache.end()) {
      return cache[x];
   }
   R result = f(x);
   cache[x] = result;
   return result;
}
```

- Die Typen eines Parameters eines Funktions-Templates können von den Template-Typenparametern in beliebiger Weise abgeleitet werden.
- Hier erwartet *eval* zwei Parameter: Eine (möglicherweise nur aufwendig auszuwertende) Funktion f und ein Argument x. In der Variablen *cache* werden bereits ausgerechnete Werte $fx(x)$ notiert, um wiederholte Berechnungen zu vermeiden.

```
int fibonacci(int i) {
   if (i <= 2) return 1;
   return eval(fibonacci, i-1) + eval(fibonacci, i-2);
}

//
cout << eval(fibonacci, 10) << endl;
```

- Hier wird F_i für jedes i nur ein einziges Mal berechnet.
- Die Template-Parameter R und T werden hier ebenfalls vollautomatisch abgeleitet.

```
template<int N, typename T>
T sum(T a[N]) {
   T result = T();
   for (int i = 0; i < N; ++i) {
      result += a[i];
   }
   return result;
}
```

- Ganzzahlige Template-Parameter sind zulässig einschließlich Aufzählungstypen (**enum**).
- Diese können beispielsweise bei Vektoren eingesetzt werden.
- Wenn der Typ unbekannt ist, aber eine explizite Initialisierung gewünscht wird, kann dies durch die explizite Verwendung des Default-Constructors geschehen. Dieser liefert hier auch korrekt auf 0 initialisierte Werte für elementare Datentypen wie **int** oder **float**.

```
   int a[] = {1, 2, 3, 4};
   cout << sum<4>(a) << endl;
```

```
template<typename CONTAINER>
bool is_palindrome(const CONTAINER& cont) {
   if (cont.empty()) return true;
   typename CONTAINER::const_iterator forward(cont.begin());
   typename CONTAINER::const_iterator backward(cont.end());
   --backward;
   for(;;) {
      if (forward == backward) return true;
      if (*forward != *backward) return false;
      ++forward;
      if (forward == backward) return true;
      --backward;
   }
}
```

- Da die Zugriffs-Operatoren für Iteratoren für alle Container einheitlich sind, ist es diesem Template egal, ob es sich um eine *list*, eine *deque*, einen *string* oder was auch immer handelt, sofern alle verwendeten Operatoren unterstützt werden.
- Bei zusammengesetzten Typnamen ist innerhalb von Templates das Schlüsselwort **typename** wichtig, damit eine syntaktische Analyse auch von noch nicht instantiierten Templates möglich ist.

- Häufig ist bei Templates für Container und/oder Iteratoren die Deklaration abgeleiteter Typen notwendig wie etwa bei dem Elementtyp des Containers. Hier ist es hilfreich, dass sowohl Container als auch Iteratoren folgende namenstechnisch hierarchisch untergeordnete Typen anbieten:

value_type	Zugehöriger Elemente-Typ
reference	Referenz-Typ zu *value_type*
difference_type	Datentyp für die Differenz zweiter Iteratoren; sinnvoll etwa bei *vector*, *string* oder *deque*
size_type	Passender Typ für die Zahl der Elemente

- Wenn der übergeordnete Typ ein Template-Parameter ist, dann muss jeweils **typename** vorangestellt werden.

```
template<typename ITERATOR>
inline typename ITERATOR::value_type sum(ITERATOR from, ITERATOR to) {
   typename ITERATOR::value_type s = typename ITERATOR::value_type();
   while (from != to) {
      s += *from++;
   }
   return s;
}
```

- Die Template-Funktion *sum* erwartet zwei Iteratoren und liefert die Summe aller Elemente, die durch diese beiden Iteratoren eingegrenzt werden (inklusiv bei dem ersten Operator, exklusiv bei dem zweiten).

```
#include <iterator>
#include <iostream>
// ...
int main() {
   typedef int ELEMENT;
   cout << "sum = " <<
      sum(istream_iterator<ELEMENT>(cin),
         istream_iterator<ELEMENT>()) <<
      endl;
}
```

- Praktischerweise gibt es Stream-Iteratoren, die wie die bekannten Iteratoren arbeiten, jedoch die gewünschten Elemente jeweils auslesen bzw. herausschreiben.
- *istream_iterator* ist ein Iterator für einen beliebigen *istream*. Der Default-Konstruktor liefert hier einen Endezeiger.

- Polymorphismus bedeutet, dass die jeweilige Methode bzw. Funktion in Abhängigkeit der Parametertypen (u.a. auch nur von einem einzigen Parametertyp) ausgewählt wird.
- Dies kann statisch (also zur Übersetzzeit) oder dynamisch (zur Laufzeit) erfolgen.
- Ferner lässt sich unterscheiden, ob die Typen irgendwelchen Beschränkungen unterliegen oder nicht.
- Dies lässt sich prinzipiell frei kombinieren. C++ unterstützt davon jedoch nur zwei Varianten:

	statisch	dynamisch
beschränkt	(z.B. in Ada)	virtuelle Methoden in C++
unbeschränkt	Templates in C++	(z.B. in Smalltalk oder Perl)

Statischer vs. dynamischer Polymorphismus in C++ 305

Vorteile dynamischen Polymorphismus in C++:

- Unterstützung heterogener Datenstrukturen, etwa einer Liste von Widgets oder graphischer Objekte.
- Die Schnittstelle ist durch die Basisklasse klarer definiert, da sie dadurch beschränkt ist.
- Der generierte Code ist kompakter.

Vorteile statischen Polymorphismus in C++:

- Erhöhte Typsicherheit.
- Die fehlende Beschränkung auf eine Basisklasse erweitert den potentiellen Anwendungsbereich. Insbesondere können auch elementare Datentypen mit unterstützt werden.
- Der generierte Code ist effizienter.

Statischer Polymorphismus für nicht-verwandte Klassen 306

```
class StdRand {
   public:
      void seed(long seedval) { srand(seedval); }
      long next() { return rand(); }
};

class Rand48 {
   public:
      void seed(long seedval) { srand48(seedval); }
      long next() { return lrand48(); }
};
```

- Gegeben seien zwei Klassen, die nicht miteinander verwandt sind, aber bei einigen relevanten Methoden die gleichen Signaturen offerieren wie hier etwa bei *seed* und *next*.

Statischer Polymorphismus für nicht-verwandte Klassen 307

```
template<class Rand>
int test_sequence(Rand& rg) {
   const int N = 64;
   int hits[N][N][N] = {{{0}}};
   rg.seed(0);
   int r1 = rg.next() / N % N;
   int r2 = rg.next() / N % N;
   int max = 0;
   for (int i = 0; i < N*N*N*N; ++i) {
      int r3 = rg.next() / N % N;
      int count = ++hits[r1][r2][r3];
      if (count > max) {
         max = count;
      }
      r1 = r2; r2 = r3;
   }
   return max;
}
```

- Dann können beide von der gleichen Template-Funktion behandelt werden.

Statischer Polymorphismus für nicht-verwandte Klassen 308

```
int main() {
   StdRand stdrand;
   Rand48 rand48;
   cout << "result of StdRand: " << test_sequence(stdrand) << endl;
   cout << "result of Rand48: " << test_sequence(rand48) << endl;
}
```

- Hier verwendet *test_sequence* jeweils die passenden Methoden *seed* und *next* in Abhängigkeit des statischen Argumenttyps.
- Die Kosten für den Aufruf virtueller Methoden entfallen hier. Dafür wird hier der Programmtext für *test_sequence* für jede Typen-Variante zusätzlich generiert.

```
template<typename T>
T mod(T a, T b) {
   return a % b;
}

double mod(double a, double b) {
   return fmod(a, b);
}
```

- Explizite Spezialfälle können in Konkurrenz zu implizit instantiierbaren Templates stehen. Sie werden dann, falls sie irgendwo passen, bevorzugt verwendet.
- Auf diese Weise ist es auch möglich, effizientere Algorithmen für Spezialfälle neben dem allgemeinen Template-Algorithmus zusätzlich anzubieten.

```
template<typename T>
char* tell_type(T* p) { return "is a pointer"; }

template<typename T>
char* tell_type(T (*f)()) { return "is a function"; }

template<typename T>
char* tell_type(T v) { return "is something else"; }

int main() {
   int* p; int a[10]; int i;
   cout << "p " << tell_type(p) << endl;
   cout << "a " << tell_type(a) << endl;
   cout << "i " << tell_type(i) << endl;
   cout << "main " << tell_type(main) << endl;
}
```

- Speziell konstruierte Typen können separat behandelt werden, so dass sich etwa Zeiger von anderen Typen unterscheiden lassen.

```
clonard$ g++ -c -DLAST=30 Primes.C 2>&1 | fgrep 'In member'
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 29]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 23]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 19]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 17]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 13]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 11]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 7]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 5]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 3]':
Primes.C: In member function 'void Prime_print<i>::f() [with int i = 2]':
clonard$
```

- Auf einer Sitzung des ISO-Standardisierungskommitees im Jahr 1994 demonstrierte Erwin Unruh die Möglichkeit, Templates zur Programmierung zur Übersetzungszeit auszunutzen.
- Sein Beispiel berechnete die Primzahlen. Die Ausgabe erfolgte dabei über die Fehlermeldungen des Übersetzers.
- Siehe http://www.erwin-unruh.de/Prim.html

```
template<int N>
class Fibonacci {
   public:
      enum {
         result = Fibonacci<N-1>::result +
                  Fibonacci<N-2>::result
      };
};

template<>
class Fibonacci<1> {
   public: enum { result = 1 };
};

template<>
class Fibonacci<2> {
   public: enum { result = 1 };
};
```

- Templates können sich selbst rekursiv verwenden. Die Rekursion lässt sich dann durch die Spezifikation von Spezialfällen begrenzen.

```
template<int N>
class Fibonacci {
   public:
      enum {
         result = Fibonacci<N-1>::result +
                  Fibonacci<N-2>::result
      };
};
```

- Dabei ist es sinnvoll, mit **enum**-Konstantenberechnungen zu arbeiten, weil diese zwingend zur Übersetzzeit erfolgen.
- Alternativ sind auch statische Konstanten denkbar, die aber im Falle von Referenzierungen zu impliziten Instantiierungen führen und somit nicht mehr reine Objekte zur Übersetzzeit sind.

```
int a[Fibonacci<6>::result];
int main() {
   cout << sizeof(a)/sizeof(a[0]) << endl;
}
```

- Zur Übersetzzeit berechnete Werte können dann auch selbstverständlich zur Dimensionierung globaler Vektoren verwendet werden.

```
clonard$ make
gcc-makedepend  Fibonacci.C
g++ -Wall -g   -c -o Fibonacci.o Fibonacci.C
g++ -o Fibonacci -R/usr/local/lib Fibonacci.o
clonard$ ./Fibonacci
8
clonard$
```

Vermeidung von Schleifen mit rekursiven Templates 315

```
template <int N, typename T>
class Sum {
   public:
      static inline T result(T* a) {
         return *a + Sum<N-1, T>::result(a+1);
      }
};

template <typename T>
class Sum<1, T> {
   public:
      static inline T result(T* a) {
         return *a;
      }
};
```

- Rekursive Templates können verwendet werden, um **for**-Schleifen mit einer zur Übersetzzeit bekannten Zahl von Iterationen zu ersetzen.

Vermeidung von Schleifen mit rekursiven Templates 316

```
template <int N, typename T>
inline T sum(T* a) {
   return Sum<N, T>::result(a);
}

int main() {
   int a[] = {1, 2, 3, 4, 5};
   cout << sum<5>(a) << endl;
}
```

- Die Template-Funktion *sum* vereinfacht hier die Nutzung.

- Traits sind Charakteristiken, die mit Typen assoziiert werden.
- Die Charakteristiken selbst können durch Klassen repräsentiert werden und die Assoziationen können implizit mit Hilfe von Templates oder explizit mit Template-Parametern erfolgen.

Sum.h

```
#ifndef SUM_H
#define SUM_H

template <typename T>
inline T sum(T const* begin, T const* end) {
   T result = T();
   for (T const* it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Die Template-Funktion *sum* erhält einen Zeiger auf den Anfang und das Ende eines Arrays und liefert die Summe aller enthaltenen Elemente.
- (Dies ließe sich auch mit Iteratoren lösen, darauf wird hier jedoch der Einfachheit halber verzichtet.)

TestSum.C

```
#include "Sum.h"
#include <iostream>
#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))
using namespace std;
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   float floats[] = {1.2, 3.7, 4.8};
   cout << "sum of floats[] = " <<
      sum(floats, floats + DIM(floats)) << endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   cout << "sum of text[] = " << sum(text, text + DIM(text)) << endl;
}
```

- Bei den ersten beiden Arrays funktioniert das Template recht gut. Weswegen scheitert es im dritten Fall?

```
thales$ testsum
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = ,
thales$
```

SumTraits.h

```
#ifndef SUM_TRAITS_H
#define SUM_TRAITS_H

// by default, we use the very same type
template <typename T>
class SumTraits {
   public:
      typedef T SumValue;
};

// special case for char
template <>
class SumTraits<char> {
   public:
      typedef int SumValue;
};

#endif
```

- Die Template-Klasse *SumTraits* liefert als Charakteristik den jeweils geeigneten Datentyp für eine Summe von Werten des Typs T.
- Per Voreinstellung ist das T selbst, aber es können Ausnahmen definiert werden wie hier zum Beispiel für *char*.

TestSum2.C

```
#include "Sum2.h"
#include <iostream>

using namespace std;

#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))

int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   float floats[] = {1.2, 3.7, 4.8};
   cout << "sum of floats[] = " <<
      sum(floats, floats + DIM(floats)) << endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   cout << "sum of text[] = " << sum(text, text + DIM(text)) << endl;
}
```

```
thales$ testsum2
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = 3372
thales$
```

Sum3.h

```
#ifndef SUM3_H
#define SUM3_H

#include "SumTraits.h"

template <typename T, typename ST = SumTraits<T> >
class Sum {
   public:
      typedef typename ST::SumValue SumValue;
      static typename SumValue sum(T const* begin, T const* end) {
         SumValue result = SumValue();
         for (T const* it = begin; it < end; ++it) {
            result += *it;
         }
         return result;
      }
};

template <typename T>
inline typename SumTraits<T>::SumValue sum(T const* begin, T const* end) {
   return Sum<T>::sum(begin, end);
}

template <typename ST, typename T>
inline typename ST::SumValue sum(T const* begin, T const* end) {
   return Sum<T, ST>::sum(begin, end);
}

#endif
```

Sum3.h

```
template <typename T, typename ST = SumTraits<T> >
class Sum {
   public:
      typedef typename ST::SumValue SumValue;
      static typename SumValue sum(T const* begin, T const* end) {
         SumValue result = SumValue();
         for (T const* it = begin; it < end; ++it) {
            result += *it;
         }
         return result;
      }
};
```

- C++ unterstützt voreingestellte Template-Parameter.
- Leider nur bei Template-Klassen und nicht bei Template-Funktionen. Deswegen muss die Template-Funktion hier in eine Klasse mit einer statischen Funktion verwandelt werden.
- Diese Konstruktion ermöglicht dann einem Nutzer dieser Konstruktion die Voreinstellung zu übernehmen oder bei Bedarf eine eigene Traits-Klasse zu spezifizieren.

Sum3.h

```
template <typename T>
inline typename SumTraits<T>::SumValue sum(T const* begin, T const* end) {
   return Sum<T>::sum(begin, end);
}

template <typename ST, typename T>
inline typename ST::SumValue sum(T const* begin, T const* end) {
   return Sum<T, ST>::sum(begin, end);
}
```

- Mit diesen beiden Template-Funktionen wird wieder die zuvor gewohnte Bequemlichkeit hergestellt.
- Die erste Variante entspricht der zuvor gewohnten Funktionalität (implizite Wahl der Charakteristik).
- Die zweite Variante erlaubt die Spezifikation der Traits-Klasse. Dies erfolgt praktischerweise über den ersten Template-Parameter, damit der zweite implizit über den Aufruf bestimmt werden kann.

TestSum3.C

```
#include "Sum3.h"
#include <iostream>
#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))

using namespace std;

class MyTraits {
   public:
      typedef double SumValue;
};

int main() {
   int numbers[] = {2147483647, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   cout << "sum of numbers[] = " <<
      sum<MyTraits>(numbers, numbers + DIM(numbers)) << endl;
}
```

```
thales$ testsum3
sum of numbers[] = -2147483639
sum of numbers[] = 2.14748e+09
thales$
```