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 std::map<T, R> cache;
   // attempt to insert default value
   typename std::map<T, R>::iterator it; bool inserted;
   std::tie(it, inserted) = cache.insert(std::make_pair(x, R()));
   // if this was successful, it wasn't computed before
   if (inserted) {
      it->second = f(x); // replace default value by actual value
   }
   return it->second;
}
```

- 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 $f(x)$ notiert, um wiederholte Berechnungen zu vermeiden.
- Bei zusammengesetzten Typnamen ist innerhalb von Templates das Schlüsselwort **typename** wichtig, damit eine syntaktische Analyse auch von noch nicht instantiierten Templates möglich ist.

```
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;
   auto forward(cont.begin());
   auto backward(cont.end());
   --backward;
   for(;;) {
      if (forward == backward) return true;
      if (*forward != *backward) return false;
      ++forward;
      if (forward == backward) return true;
      --backward;
   }
}
```

- Da fast alle Container bidirektionale Iteratoren mit einheitlichen Operationen anbieten, ist es diesem Template egal, ob es sich um eine *list*, eine *deque*, einen *string* oder was auch immer handelt, sofern der Iterator bidirektional 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 auto
sum(ITERATOR from, ITERATOR to) -> decltype(*from + *from) {
   typedef decltype(*from + *from) Value;
   Value s = Value();
   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).
- **decltype** kam durch C++11 hinzu und erlaubt es, einen Datentyp für eine Deklaration aus einem Ausdruck abzuleiten.
- Da dies von den Parametern abhängt, wurde der Rückgabetyp hinter die Parameter verschoben und zum Ausgleich zu Beginn **auto** angegeben.

```
#include <iterator>
#include <iostream>
// ...
int main() {
   using ELEMENT = int;
   std::cout << "sum = " <<
      sum(std::istream_iterator<ELEMENT>(cin), std::istream_iterator<ELEMENT>()
         << std::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.

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

- 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:
      static constexpr int
         result = Fibonacci<N-1>::result +
                  Fibonacci<N-2>::result;
};

template<>
class Fibonacci<1> {
   public: static constexpr int result = 1;
};

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

- Dabei ist es sinnvoll, mit **constexpr** zu arbeiten, weil die hier angegebenen Ausdrücke zur Übersetzzeit berechnet werden müssen.
- Da **constexpr** erst mit C++11 eingeführt wurde, wurde früher auf **enum** zurückgegriffen. Auch einfache Funktionen mit einer **return**-Anweisung können mit **constexpr** deklariert werden.
- Beginnend mit C++14 sind lokale Variablen und Schleifen in **constexpr**-Funktionen zugelassen. (Aber der *gcc* unterstützt dies erst ab Version 5.0.)

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

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

```
thales$ make
gcc-makedepend -std=gnu++11 Fibonacci.cpp
g++ -Wall -g -std=gnu++11  -c -o Fibonacci.o Fibonacci.cpp
g++ -o Fibonacci  Fibonacci.o
thales$ Fibonacci
8
thales$
```

Vermeidung von Schleifen mit rekursiven Templates 399

```
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 400

```
template <typename T>
inline auto sum(T& a) -> decltype(a[0] + a[0]) {
   return Sum<std::extent<T>::value,
      typename std::remove_extent<T>::type>::result(a);
}

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

- Die Template-Funktion *sum* vereinfacht hier die Nutzung.
- Da der Parameter per Referenz übergeben wird, bleibt hier die Typinformation einschließlich der Dimensionierung erhalten.
- *std*::*extent*<*T*>::*value* liefert die Dimensionierung, *std*::*remove_extent*<*T*>::*type* den Element-Typ des Arrays.

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Beginnend mit C++11 werden auch Templates mit variabel langen Parameterlisten unterstützt. Dies geht hier unter Verwendung von *Values... values*.

```
for_values([](unsigned int i) {
   std::cout << i << std::endl;
}, 4, 6, 7);
```

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Diese Template-Funktion ist rekursiv organisiert, wobei die Rekursion zur Übersetzzeit aufgelöst wird.
- Der erste Fall dient dem Ende der Rekursion, *body* wird hier nur für einen einzigen Wert *value* aufgerufen.
- Der zweite Fall ist für den Induktionsschritt. Der erste Wert wird herausgegriffen, *body* dafür aufgerufen und der Rest der Rekursion überlassen.

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Im Kontext eines Templates können auch Funktionen variable Zahlen von Argumenten haben. Diese ist aber nur zur Übersetzzeit variabel.
- Für jede vorkommende Parameterzahl wird eine entsprechende Funktion erzeugt. Normalerweise wird das alles aber per **inline** zur Übersetzzeit aufgelöst.
- Am Ende bleibt hier nur eine entsprechende Sequenz übrig.

```
        movl    $4, %eax
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        call    _ZZ4mainENKUljE_clEj.isra.0
        movl    $6, %eax
        call    _ZZ4mainENKUljE_clEj.isra.0
        movl    $7, %eax
        call    _ZZ4mainENKUljE_clEj.isra.0
        addl    $4, %esp
        xorl    %eax, %eax
        popl    %ecx
        popl    %ebp
```

- Dies ist der von *g++* erzeugte Assemblertext für den Aufruf von *for_values* mit den Werten 4, 6 und 7.
- Beim Label *_ZZ4mainENKUljE_clEj.isra.0* ist der Programmtext des Lambda-Ausdrucks. Der erste Aufruf ist noch etwas aufwendiger, da der Stack vorbereitet werden muss. Die weiteren Aufrufe sind aber vereinfacht.

```
for_values([](auto value) {
   std::cout << value << std::endl;
}, 4, "Huhu", 9.3);
```

- Ab C++14 dürfen auch Lambda-Ausdrücke polymorph sein.
- Entsprechend darf hier der *value*-Parameter **auto** deklariert werden, so dass dieser jeweils von jedem der Argumente individuell abgeleitet werden kann.
- Nun zahlt sich aus, dass die *for_values*-Template-Funktion nicht auf einheitlichen Parametertypen bei dem Aufruf von *body* besteht.

```
template <typename Derived>
class Base {
   // ...
};

class Derived: public Base<Derived> {
   // ...
};
```

- Es ist möglich, eine Template-Klasse mit einer von ihr abgeleiteten Klasse zu parametrisieren.
- Der Begriff geht auf James Coplien zurück, der diese Technik immer wieder beobachtete.
- Diese Technik nützt aus, dass die Methoden der Basisklasse erst instantiiert werden, wenn der Template-Parameter (d.h. die davon abgeleitete Klasse) dem Übersetzer bereits bekannt sind. Entsprechend kann die Basisklasse von der abgeleiteten Klasse abhängen.

```
// nach Michael Lehn
template <typename Implementation>
class Base {
   public:
      Implementation& impl() {
         return static_cast<Implementation&>(*this);
      }
      void aMethod() {
         impl().aMethod();
      }
};

class Implementation: public Base<Implementation> {
   public:
      void aMethod() {
         // ...
      }
};
```

- Das CRTP ermöglicht hier die saubere Trennung zwischen einem herausfaktorisierten Teil in der Basisklasse von einer Implementierung in der abgeleiteten Klasse, wobei keine Kosten für virtuelle Methodenaufrufe zu zahlen sind.

```
Implementation& impl() {
   return static_cast<Implementation&>(*this);
}
```

- Mit **static_cast** können Typkonvertierungen ohne Überprüfungen zur Laufzeit vorgenommen werden. Insbesondere ist eine Konvertierung von einem Zeiger oder einer Referenz auf einen Basistyp zu einem passenden abgeleiteten Datentyp möglich. Der Übersetzer kann dabei aber nicht sicherstellen, dass das referenzierte Objekt den passenden Typ hat. Falls nicht, ist der Effekt undefiniert.
- In diesem Kontext ist **static_cast** genau dann sicher, wenn es sich bei dem Template-Parameter tatsächlich um die richtige abgeleitete Klasse handelt.
- Die Verwendung von **static_cast** in Verbindung mit CRTP geht auf ein 1994 veröffentlichtes Buch von John J. Barton und Lee R. Nackman zurück.

Einige Anwendungen, die durch CRTP möglich werden:

- Statische Klassenvariablen für jede abgeleitete Klasse. (Beispiel: Zähler für erzeugte bzw. noch lebende Objekte. Bei klassischer OO-Technik würde dies insgesamt gezählt werden, bei CRTP jedoch getrennt nach den einzelnen Instantiierungen.)
- Die abgeleitete Klasse implementiert einige implementierungsspezifische Methoden, die darauf aufbauenden weiteren Methoden kommen durch die Basis-Klasse. (Beispiel: Wenn die abgeleitete Klasse den Operator == unterstützt, kann die Basisklasse darauf basierend den Operator != definieren.
- Verbessertes Namensraum-Management auf Basis des *argument-dependent lookup* (ADL). In der Basisklasse definierte **friend**-Funktionen können so von den abgeleiteten Klassen importiert werden. (Technik von Abrahams und Gurtovoy.)
- Zur Konfliktauflösung bei überladenen Funktions-Templates als Alternative zu *std*::*enable_if*. (Technik von Abrahams und Gurtovoy.)

Allzu leicht geraten Template-Funktionen zu allgemein:

```
template <typename Alpha, typename Matrix>
void scale(const Alpha& alpha, Matrix& A) {
   // scale a matrix
}

template <typename Alpha, typename Vector>
void scale(const Alpha& alpha, Vector& x) {
   // scale a vector
}
```

Hier stehen beide Definition zueinander in Konflikt. Wie lässt sich dieser lösen?

```
template <typename Alpha, typename Matrix>
typename std::enable_if<IsMatrix<Matrix>::value, void>::type
void scale(const Alpha& alpha, Matrix& A) {
   // scale a matrix
}

template <typename Alpha, typename Vector>
typename std::enable_if<IsVector<Vector>::value, void>::type
void scale(const Alpha& alpha, Vector& x) {
   // scale a vector
}
```

- Mit Hilfe der SFINAE-Technik können wir den Konflikt auflösen.
- Wir müssen hier nur für jeden weitere polymorphe *Matrix*- oder *Vector*-Variante ein entsprechendes *IsMatrix*- bzw. *IsVector*-Konstrukt ergänzen.

```
template <typename Derived> struct Matrix {};
template <typename Derived> struct Vector {};

template <typename Alpha, typename Matrix>
void scale(const Alpha& alpha, Matrix<MA>& A_) {
   MA& A = static_cast<MA&>(A_);
   // scale matrix A
}

template <typename Alpha, typename Vector>
void scale(const Alpha& alpha, Vector<VX>& x_) {
   VX& x = static_cast<VX&>(x_);
   // scale vector x
}
```

- Alternativ können wir mit der von Abrahams und Gurtovoy vorgeschlagenen Technik darauf bestehen, dass alle Matrix- und Vektorklassen via CRTP von den entsprechenden Basisklassen abgeleitet werden.
- Dann lassen sich die Konflikte ohne SFINAE auflösen.