```
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.

```
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$
```

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

- Klassen-Templates können teilweise oder vollständig spezialisiert werden. (Bei Template-Funktionen ist das nicht sinnvoll, da das im Konflikt zum Überladen ist.)
- In jedem Fall müssen sie dann als Template deklariert werden, selbst wenn wie hier die Template-Parameter-Liste leer ist.
- Hinter dem deklarierten Namen (hier *Fibonacci*) werden dann alle Template-Parameter aufgezählt, die dann fest vorgegeben werden können bzw. von den verbliebenen Template-Parametern abhängen können.
- Vollständig oder partiell spezialisierte Templates ermöglichen es, eine Rekursion bei Templates zu beenden.

Rekursion mit using

```
template<std::size_t N>
struct Counter {
   using next = Counter<N+1>;
   static constexpr std::size_t value = N;
};
```

- Die Rekursion kann auch indirekt erfolgen mit Hilfe einer **using** bzw. **typedef**-Deklaration, bei der auf eine andere Template-Instanz verwiesen wird.
- Der Übersetzer wertet das nur aus, wo dies notwendig ist. Somit haben wir hier keine Endlos-Rekursion.

```
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.

```
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<unsigned int... Is> struct seq {
   using next = seq<Is..., sizeof...(Is)>;
};
template<unsigned int N> struct gen_seq {
   using type = typename gen_seq<N-1>::type::next;
};
template<> struct gen_seq<0> {
   using type = seq<>;
};
template<unsigned int N>
using make_seq = typename gen_seq<N>::type;
```

- Bei der Metaprogrammierung kann es sinnvoll sein, mit dynamischen Listen zu hantieren. Hierfür bieten sich variabel lange Template-Parameterlisten an.
- **sizeof**...(*Is*) liefert die Zahl der Elemente des Template-Parameter-Packs *Is*.
- Das Konstrukt dient dazu Template-Parameterlisten mit den Zahlen 0 bis zu einem gewünschten Limit aufzubauen.

```
template<typename F>
void process_values(F&& f) {
}
template<typename F, typename Arg, typename... Args>
void process_values(F&& f, Arg arg, Args... args) {
   f(arg); process_values(f, args...);
}
template<typename F, unsigned int... Is>
void do_values(F&& f, seq<Is...>) {
   process_values(std::forward<F>(f), Is...);
}

int main() {
   do_values([](unsigned int i) -> void {
      std::cout << " " << i;
   }, make_seq<20>());
   std::cout << std::endl;
}
```

- Mit Hilfe einer Template-Funktion lässt sich dann die Sequenz extrahieren, um z.B. sie in eine Parameterliste zu verwandeln.

```
template<unsigned int... Is>
constexpr auto make_array(seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {Is...};
}

auto values = make_array(make_seq<20>());

int main() {
   for (auto val: values) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Alternativ kann das auch genutzt werden, um damit ein Array zu initialisieren.
- Man beachte, dass das Array global ist und bereits zur Übersetzzeit vollständig mit Werten gefüllt wird.

```
template<unsigned int... Is>
constexpr auto make_array_of_squares(seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {Is * Is...};
}

auto squares = make_array_of_squares(make_seq<20>());

int main() {
   for (auto val: squares) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Wenn sogenannte *template parameter packs* mit Hilfe von ... expandiert werden, kann auch ein Konstrukt angegeben werden, das für jeden einzelnen Parameter expandiert wird und – durch Kommata getrennt – zusammengefügt wird.

```
template<typename Map, unsigned int... Is>
constexpr auto make_array(Map&& map, seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {map(Is)...};
}

auto squares = make_array([](unsigned int val) constexpr {
   return val * val;
}, make_seq<20>());

int main() {
   for (auto val: squares) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Geht dies auch mit **constexpr**-Lambda-Ausdrücken? Ja, ab C++17!

```
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.