Beginnend mit C++11 wird

for *(for-range-declaration : expression) statement*

durch den Übersetzer zu folgendem Programmtext expandiert:

```
{
    auto&& range = (expression);
    for (auto begin = range.begin(), end = range.end();
            begin != end; ++begin) {
        for-range-declaration = *begin;
        statement
    }
}
```

Bei Zeigern wird stattdessen auf die Adressarithmetik zurückgegriffen und statt den Methoden *begin*() und *end*() können auch entsprechende polymorphe Funktionen zum Einsatz kommen, wenn die Methoden nicht existieren.

Idee: Ist es möglich,

```
for (int i = 1; i < 10; ++i) {
    // ...
}
```

zu

```
for (auto i: range(1, 10)) {
    // ...
}
```

zu vereinfachen? Prinzipiell ja: Wir benötigen eine entsprechende Klasse mit einem zugehörigen Iterator-Typ, der die Operatoren $*$ und $++$ unterstützt.

range.hpp

```
template <typename T>
class IntegralRange {
   public:
      IntegralRange(T begin_val, T end_val) :
         begin_val{begin_val}, end_val{end_val} {
      }

      class Iterator {
         // ...
      };

      Iterator begin() const {
         return Iterator(begin_val);
      }
      Iterator end() const {
         return Iterator(end_val);
      }

   private:
      T begin_val;
      T end_val;
};
```

range.hpp

```
class Iterator {
   public:
      Iterator(T val) : val{val} {
      }
      T operator*() {
         return val;
      }
      Iterator& operator++() {
         ++val; return *this;
      }
      Iterator& operator++(int) {
         Iterator it = *this;
         ++val; return it;
      }
      bool operator==(const Iterator& other) const {
         return val == other.val;
      }
      bool operator!=(const Iterator& other) const {
         return val != other.val;
      }

   private:
      T val;
};
```

Das müsste nun in eine entsprechende Template-Funktion verpackt werden, die den Typparameter automatisch bestimmt:

```
template<typename T>
IntegralRange<T> range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Problem: Diese Template-Funktion akzeptiert zunächst prinzipiell beliebige T, geht dann aber schief, wenn T sich nicht wie ein ganzzahliger Datentyp verhält. Ist es möglich, hier eine Einschränkung vorzunehmen?

- Wenn der C++-Übersetzer alle in Frage kommenden Kandidaten einer Template-Funktion in Betracht zieht, werden zunächst die Template-Parameter nur in die Template-Parameter-Deklaration und die Funktionsdeklaration (Parameter und Return-Typ) eingesetzt und danach überprüft, ob das Resultat semantisch zulässig ist.
- Wenn die Antwort nein ist, dann ist das kein Fehler. Stattdessen wird nur ganz einfach der Kandidat aus dem Pool der Kandidaten entfernt.
- Das Prinzip nennt sich SFINAE: *Substitution Failure Is Not An Error*.

Angenommen, wir wollen bei *range* nur **int** und **unsigned int** zulassen. Dann könnten wir das so organisieren:

```
template <typename T> struct is_integer {};
template <> struct is_integer<int> {
    using type = IntegralRange<int>;
}
template <> struct is_integer<unsigned int> {
    using type = IntegralRange<unsigned int>;
}

template<typename T>
typename is_integer<T>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Hier wird nun *is_integer*<*T*>::*type* benutzt, das nur für die Fälle **int** und **unsigned int** definiert ist. Für alle anderen Typen gibt es keinen *type*, was zu einem Ausschluss per SFINAE führt.

Es lohnt sich, die Bedingungen (wie hier *is_integer*) von dem gewünschten Typ zu trennen (war hier *IntegralRange*<*T*>). Die Standard-Bibliothek bietet hierfür *std*::*enable_if*, das so definiert sein könnte:

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T; };
```

Der erste Parameter ist die **bool**-Bedingung, der zweite Parameter spezifiziert den gewünschten Typ.

Als nächstes wird eine Hilfsklasse benötigt, die auf der Template-Ebene einen konstanten Wert eines integralen Typs repräsentiert. Die C++-Bibliothek bietet hierfür *std*::*integral_constant* an, das wie folgt implementiert sein könnte:

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant;
    constexpr operator value_type() const noexcept { return value;
        }
    constexpr value_type operator()() const noexcept { return
        value; }
};
```

Darauf basierend lassen sich Hilfsklassen für **bool**-Werte definieren:

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

Nun lässt sich die Aufzählung der zugelassenen Typen unabhängig vom Resultat-Typ umsetzen:

```
template <typename T> struct is_integer : public std::
    false_type{};
template <> struct is_integer<int> : public std::true_type{};
template <> struct is_integer<unsigned int> : public std::
    true_type{};

template<typename T>
typename std::enable_if<
    is_integer<T>::value, // boolean-valued condition
    IntegralRange<T> // wanted type, if correct
>::type // is the wanted type, if correct
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

In <*type_traits*> finden sich aber bereits eine Vielzahl einzelner Tests, u.a. auch *std*::*is_integral*, der alle integralen Typen umfasst:

```
template<typename T>
typename std::enable_if<
    std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Wenn wir eine weitere Variante für Zeiger und Iteratoren zulassen wollen, könnten wir im einfachsten Falle die ganzzahligen Typen ausschließen:

```
template<typename T>
typename std::enable_if<
    !std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

Es wäre aber besser, die auf *IntegralRange* basierende Variante auf Typen zu beschränken, die wie Zeiger oder Iteratoren aussehen, d.h. die die unären Operatoren $*$ und $++$ unterstützen.
Wir benötigen hierzu etwas Handwerkszeug, um per SFINAE die Unterstützung von Operatoren und Methoden zu testen:

- Mit *std*::*declval* können wir aus einem Typ ein Objekt des Typs machen, ohne zu wissen, wie der Konstruktor aussieht.
- Mit **decltype** kann der Typ eines Ausdrucks wie ein Typname verwendet werden.
- Mit *std*::*remove_reference* werden wir && bzw. & los.

std::*declval* ist eine Template-Funktion aus <*utility*>, die nur deklariert, jedoch nie definiert wird:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval();
```

Wir dürfen *declval* somit nur in Konstruktionen einsetzen, die vollständig zur Übersetzzeit ausgewertet werden und bei denen nur der Typ relevant ist, jedoch nicht die (nicht vorhandene) Definition der Funktion.

Mit *add_rvalue_reference* bleiben Referenztypen so wie sie sind, nur Typen ohne Referenz (also weder & noch &&) werden mit && versehen.

Diese Ergänzung ist notwendig, um beispielsweise die Existenz von Methoden testen zu können, die nur für *rvalue*-Objekte zugänglich sind:

```
struct foo { void bar()&& { /*... */} };
```

Angenommen, wir haben einen Template-Parameter für einen Zeigertyp oder Iterator und wollen den Typ nach der Dereferenzierung haben:

```
template<typename T>
struct dereferenced {
    using type = decltype(*std::declval<T>());
};
```

Innerhalb von **decltype** wird nur der Typ eines Ausdrucks bestimmt, dieser jedoch nicht bewertet. Deswegen ist *std*::*declval*<*T*>() zulässig und liefert ein virtuelles Objekt des Typs *T*, das wir dann dereferenzieren können, um davon mit **decltype** den Typ zu bestimmen.

dereferenced<**int***>::*type* und *dereferenced*<*std*::*vector*<**int**>::*iterator*> entsprechen nun jeweils **int**.

Nun lässt sich das so kombinieren, dass wir abprüfen, ob die Operatoren $*$ und $++$ unterstützt werden:

```
template<typename T>
typename std::remove_reference<
    decltype(
        *std::declval<T&>(), // * supported?
        ++std::declval<T&>(), // ++ supported?
        /* wanted type, comes with &&: */
        std::declval<IteratorRange<T>>()
    )
>::type // now with && removed
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

- Statischer Polymorphismus basiert in C++ auf Templates und der Verlagerung der semantischen Überprüfung auf den Zeitpunkt der Instantiierung.
- Ein wesentliches Element ist die automatisierte Auswahl der am besten passenden Template-Klasse oder Template-Funktion zur Übersetzzeit.
- Die elegante Erweiterbarkeit ergibt sich aus der Möglichkeit, dass Varianten einfach per **#include** hinzugefügt werden können. Dann werden sie implizit überall berücksichtigt, wo sie anwendbar sind.
- Traits erlauben es, Eigenschaften von Typen zur Übersetzzeit zu spezifizieren und auszuwerten.
- Sowohl bei Template-Klassen als auch bei Template-Funktionen sind per SFINAE frei definierbare Einschränkungen möglich. Bei generischen Klassen ohne Einschränkungen besteht die Gefahr, dass die elegante Erweiterbarkeit für weitere Spezialfälle nicht mehr möglich ist.