- 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.
- Dynamischer Polymorphismus wird grundsätzlich von allen objekt-orientierten Programmiersprachen unterstützt.
- In einigen Fällen lässt sich die Auswahl der Methode oder Funktion auch im Rahmen der Optimierung zur Übersetzzeit treffen.
- Bei statischem Polymorphismus wird der Übersetzer gezwungen, die Auswahl zur Übersetzzeit durchzuführen.

- Für C++ werden seit einiger Zeit auch Optimierer angeboten, die beim Zusammenbau des Programms aktiv werden und Optimierungen über die einzelnen Übersetzungseinheiten hinweg vornehmen können.
- Bei neueren GCC-Versionen wird dies durch die Option *-flto* möglich (LTO = *link time optimization*). Die Option muss zur Übersetz- und Zusammenbauzeit angegeben werden.
- Zur Übersetzzeit wird dann die internen Datenstrukturen mit in der Ausgabe abgelegt, die dann zur Zusammenbauzeit ausgewertet werden kann.
- Dies eröffnet die Möglichkeit, dynamischen Polymorphismus durch statischen Polymorphismus zu ersetzen, wenn feststeht, dass bei dem Aufruf einer virtuellen Methode an einer Stelle immer die gleiche Implementierung aufgerufen wird.
- Seit dem GCC 5 konnten mit dieser Optimierung 50% der virtuellen Methodenaufrufe bei Firefox durch statische Aufrufe ersetzt werden.

- Neben der Frage, ob die Entscheidung zur Übersetzzeit oder Laufzeit fällt, 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 oder Eiffel)	virtuelle Methoden in C++
unbeschränkt	Templates in C++	(z.B. in Smalltalk oder Perl)

- Mit den *concepts* gibt es Bestrebungen, auch in C++ die Möglichkeit von Beschränkungen einzuführen. In C++17 wurden *concepts* noch nicht integriert, für C++20 sind sie aber geplant.
- Der *g++* unterstützt eine experimentelle Fassung der *concepts*, die über die Option „-fconcepts“ aktiviert werden kann.

Statischer vs. dynamischer Polymorphismus in C++ 222

Vorteile dynamischen Polymorphismus in C++:

- Unterstützung heterogener Datenstrukturen, etwa einer Liste von Widgets oder graphischer Objekte.
- Dynamisches Nachladen unbekannter Implementierungen ist möglich.
- 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.

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

class Rand48 {
   public:
      /* note srand48 & lrand48 are part of the
         POSIX standard but neither of the C or
         C++ standard and thereby not in std:: */
      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*.

Beispiel für statischen Polymorphismus


```
template<typename Rand>
unsigned int test_sequence(Rand& rg) {
   constexpr unsigned int N = 64;
   unsigned int hits[N][N][N] = {{{0}}};
   rg.seed(std::random_device()());
   unsigned int r1 = rg.next() / N % N;
   unsigned int r2 = rg.next() / N % N;
   unsigned int max = 0;
   for (unsigned int i = 0; i < N*N*N*N; ++i) {
      unsigned int r3 = rg.next() / N % N;
      unsigned 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.

```
int main() {
   StdRand stdrand;
   Rand48 rand48;
   std::cout << "result of StdRand: "
      << test_sequence(stdrand) << std::endl;
   std::cout << "result of Rand48: "
      << test_sequence(rand48) << std::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>
const char* tell_type(T* p) { return "is a pointer"; }

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

template<typename T>
const 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;
}
```

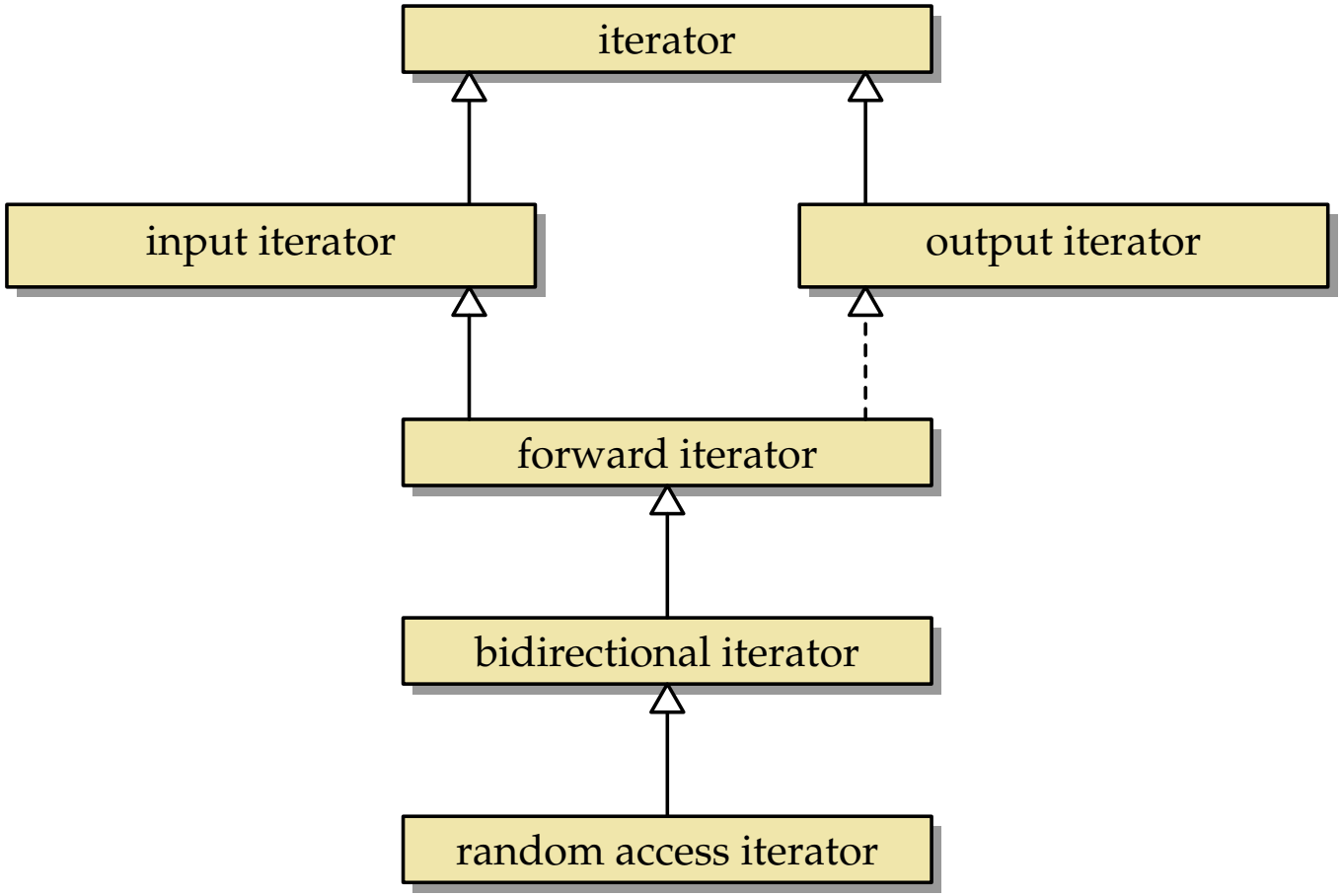
- Speziell konstruierte Typen können separat behandelt werden, so dass sich etwa Zeiger von anderen Typen unterscheiden lassen.

```
template<typename T>
constexpr std::size_t dim(const T& vec) {
   return sizeof(vec)/sizeof(*vec);
}
```

- Funktionen, die mit *constexpr* deklariert sind, werden zur Überzeitzeit ausgeführt, wenn sie mit zur Übersetzzeit bekannten Werten arbeiten.
- Hier wird die Dimensionierung eines Arrays abgefragt.
- Obige Variante lässt sich auf beliebige Typen anwenden, scheitert aber, wenn *vec* sich nicht dereferenzieren lässt. Es werden auch Zeigertypen zugelassen, bei denen dann eine 1 zurückgeliefert wird.
- Diese Fassung funktioniert nur für die Arrays, für die es gedacht ist:

```
template<typename T, std::size_t N>
constexpr std::size_t dim(const T (&vec)[N]) {
   return N;
}
```

- Iteratoren in C++ können als Verallgemeinerung von Zeigern gesehen werden, die einen universellen Zugriff auf Datenstrukturen erlauben.
- Durch die syntaktisch gleichartige Verwendung von Iteratoren und Zeigern können mit Hilfe des statischen Polymorphismus auch reguläre Zeiger als Iteratoren verwendet werden.
- Für die einzelnen Operationen eines Iterators gibt es einheitliche semantische Spezifikationen, die auch die jeweilige Komplexität angeben. Diese entsprechen denen der klassischen Zeiger-Operatoren.
- Dies sollte eingehalten werden, damit die auf Iteratoren arbeitenden Algorithmen semantisch korrekt sind und die erwartete Komplexität haben.
- Die auf Iteratoren basierenden Algorithmen sind immer generisch, d.h. es wird typischerweise mit entsprechenden impliziten Template-Parametern gearbeitet.



- Der C++-Standard spezifiziert eine (auf statischem Polymorphismus) beruhende semantische Hierarchie der Iteratoren-Klassen.

Alle Iteratoren erlauben das Dereferenzieren und das Weitersetzen:

Operator	Rückgabe-Typ	Beschreibung
$*it$	*Element&*	Zugriff auf ein Element; nur zulässig, wenn *it* dereferenzierbar ist
$++it$	*Iterator&*	Iterator vorwärts weitersetzen

Iteratoren unterstützen Kopierkonstruktoren, Zuweisungen und *std*::*swap*. Wieweit ein Iterator weitergesetzt werden darf bzw. ob jeweils eine Dereferenzierung zulässig ist, lässt sich diesen Operationen nicht entnehmen.

Diese Iteratoren erlauben es, eine Sequenz zu konsumieren, d.h. sukzessive auf die einzelnen Elemente zuzugreifen:

Operator	Rückgabe-Typ	Beschreibung
it1 != *it2*	**bool**	Vergleich zweier Iteratoren
$*it$	*Element*	Lesezugriff auf ein Element
$it->member$	Typ von *member*	Lesezugriff auf ein Datenfeld
$++it$	*Iterator&*	Iterator zuerst vorwärts weitersetzen
$it++$	*Iterator&*	Iterator danach vorwärts weitersetzen
$*it++$	*Element*	dereferenziert den Iterator und liefert diesen Wert; der Iterator wird danach weitergesetzt

avg.cpp

```
#include <iostream>
#include <iterator>

int main() {
   std::istream_iterator<double> it(std::cin);
   std::istream_iterator<double> end;
   unsigned int count = 0; double sum = 0;
   while (it != end) {
      sum += *it++; ++count;
   }
   std::cout << (sum / count) << std::endl;
}
```

- *std::istream_iterator* ist ein Input-Iterator, der den >>-Operator verwendet, um die einzelnen Werte des entsprechenden Typs von der Eingabe einzulesen.
- Charakteristisch ist hier der konsumierende Charakter, d.h. es ist nur ein einziger Durchlauf möglich.

avg.hpp

```
#ifndef AVG_HPP
#define AVG_HPP
#include <cstdlib>
#include <type_traits>

template<typename ForwardIterator>
auto avg(ForwardIterator it1, ForwardIterator it2)
      -> decltype(*it1 / sizeof(int)) {
   using T = decltype(*it1 + *it1);
   if (it1 == it2) {
      return T{};
   }
   T sum{}; std::size_t count = 0;
   while (it1 != it2) {
      sum += *it1++; ++count;
   }
   return sum / count;
}

#endif
```

- Die Ermittlung des Durchschnitts lässt sich auch als Template-Funktion formulieren.

avg.hpp

```
template<typename ForwardIterator>
auto avg(ForwardIterator it1, ForwardIterator it2)
      -> decltype(*it1 / sizeof(int)) {
   using T = decltype(*it1 + *it1);
   if (it1 == it2) {
      return T{};
   }
   T sum{}; std::size_t count = 0;
   while (it1 != it2) {
      sum += *it1++; ++count;
   }
   return sum / count;
}
```

- Mit **decltype** kann der Typ eines Ausdrucks wie ein Typname verwendet werden.
- Es ist darauf zu achten, dass *decltype*(∗*it1*) für einen *std*::*istream_iterator*<**double**> hier **const double**& liefern würde. So etwas könnte bei Bedarf mit Hilfe von *std*::*decay* auf **double** reduziert werden.

avg-test.cpp

```
#include <iostream>
#include <iterator>
#include "avg.hpp"

int main() {
   std::istream_iterator<double> it(std::cin);
   std::istream_iterator<double> end;
   std::cout << avg(it, end) << std::endl;
}
```

- Nun lässt sich *avg* auf eine beliebige mit Forward-Iteratoren spezifizierte Menge an Daten anwenden, bei denen die Template-Abhängigkeiten erfüllt sind.

Diese Iteratoren erlauben es, den Objekten einer Sequenz sukzessive neue Werte zuzuweisen oder eine Sequenz neu zu erzeugen:

Operator	Rückgabe-Typ	Beschreibung
$*it = element$	–	Zuweisung; *it* ist danach nicht notwendigerweise dereferenzierbar
$++it$	*Iterator&*	Iterator vorwärts weitersetzen
$it++$	*Iterator&*	Iterator vorwärts weitersetzen
$*it++ = element$	–	Zuweisung mit anschließendem Weitersetzen des Iterators

Zu beachten ist hier, dass Mehrfachzuweisungen auf $*it$ nicht notwendigerweise zulässig sind. Die Dereferenzierung ist nur auf der linken Seite einer Zuweisung zulässig.

inserter.cpp

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
int main() {
   std::list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::back_insert_iterator<std::list<double>> output(values);
   std::copy(input, input_end, output);
   for (auto value: values) { std::cout << value << std::endl; }
}
```

- Einfüge-Iteratoren sind Output-Iteratoren, die alle ihnen übergebenen Werte in einen Container einfügen.
- Zur Verfügung stehen *std::back_insert_iterator*, *std::front_insert_iterator* und *std::insert_iterator*.
- *std*::*copy* ist ein im Standard vorgegebener Algorithmus, der zwei einen Bereich definierende Input-Iteratoren und einen Output-Iterator als Parameter erhält.

Forward-Iteratoren unterstützen alle Operationen eines Input-Iterators. Im Vergleich zu diesen ist es zulässig, die Sequenz mehrfach zu durchlaufen. Entsprechend gilt:

- Aus $it1 == it2$ folgt $++it1 == ++it2$.
- Wenn $it1 == it2$ gilt, dann sind entweder beide Iteratoren dereferenzierbar oder keiner der beiden.
- Wenn die Iteratoren $it1$ und $it2$ dereferenzierbar sind, dann gilt $it1 == it2$ genau dann, wenn $*it1$ und $*it2$ das gleiche Objekt adressieren.

Forward-Iteratoren können auch die Operationen eines Output-Iterators unterstützen. Dann sind sie schreibbar und erlauben Mehrfachzuweisungen, ansonsten erlauben sie nur Lesezugriffe (*constant iterator*).

Beispiel für einen Forward-Iterator

forward.cpp

```
#include <iostream>
#include <forward_list>
#include <iterator>
#include <algorithm>
int main() {
   std::forward_list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::copy(input, input_end, front_inserter(values));
   // move all values < 0 to the front:
   auto middle = std::partition(values.begin(), values.end(),
      [](double val) { return val < 0; });
   std::ostream_iterator<double> out(std::cout, " ");
   std::cout << "negative values: " << std::endl;
   std::copy(values.begin(), middle, out); std::cout << std::endl;
   std::cout << "non-negative values: " << std::endl;
   std::copy(middle, values.end(), out); std::cout << std::endl;
}
```

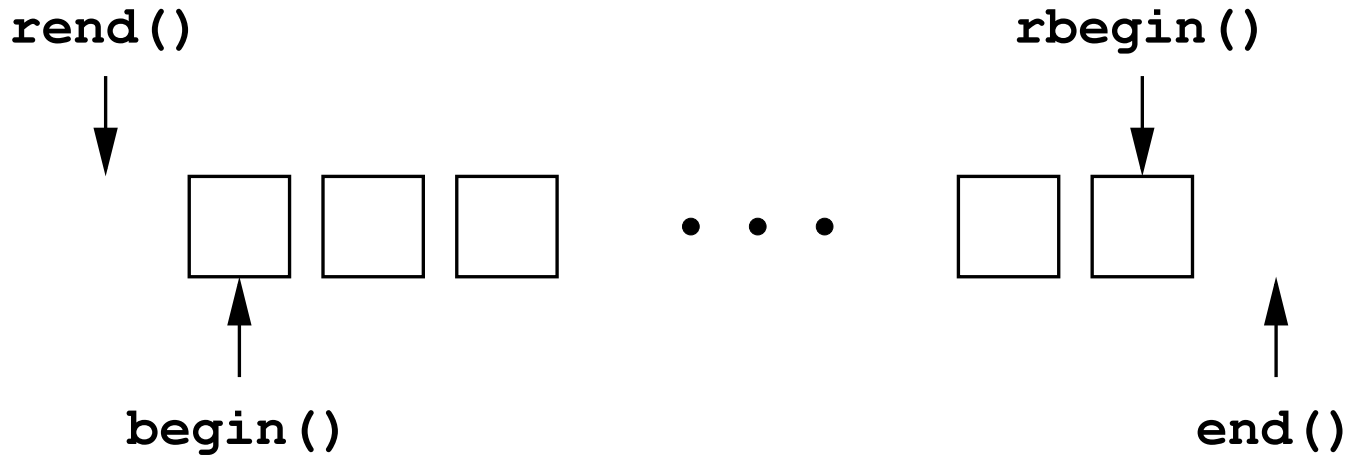
- *std*::*forward_list* ist eine einfach verkettete lineare Liste.

Bidirektionale Iteratoren sind Forward-Iteratoren, bei denen der Iterator in beide Richtungen versetzt werden kann:

Operator	Rückgabe-Typ	Beschreibung
$--it$	*Iterator&*	Iterator zuerst rückwärts weitersetzen
$it--$	*Iterator&*	Iterator danach rückwärts weitersetzen
$*it-- = element$	*Element&*	Zuweisung mit anschließendem Zurücksetzen des Iterators

Random-Access-Iteratoren sind bidirektionale Iteratoren, die die Operationen der Zeigerarithmetik unterstützen:

Operator	Rückgabe-Typ	Beschreibung
$it + n$	*Iterator*	liefert einen Iterator zurück, der n Schritte relativ zu it vorangegangen ist
$it - n$	*Iterator*	liefert einen Iterator zurück, der n Schritte relativ zu it zurückgegangen ist
$it[n]$	*Element&*	äquivalent zu $*(it+n)$
$it1 < it2$	**bool**	äquivalent zu $it2 - it1 > 0$
$it2 < it1$	**bool**	äquivalent zu $it1 - it2 > 0$
$it1 <= it2$	**bool**	äquivalent zu $!(it1 > it2)$
$it1 >= it2$	**bool**	äquivalent zu $!(it1 < it2)$
$it1 - it2$	*Distance*	Abstand zwischen $it1$ und $it2$; dies liefert einen negativen Wert, falls $it1 < it2$



iterator	bidirektionaler Iterator, der sich an der Ordnung des Containers orientiert (soweit eine Ordnung existiert)
const_iterator	analog zu *iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich
reverse_iterator	bidirektionaler Iterator, dessen Richtung der Ordnung des Containers entgegen gesetzt ist
const_reverse_iterator	analog zu *reverse_iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich

- Es ist zu beachten, dass ein Iterator *in* einen Container zeigen muss, damit auf ein Element zugegriffen werden kann, d.h. die Rückgabe-Werte von *end()* und *rend()* dürfen nicht dereferenziert werden.
- Analog ist es auch nicht gestattet, Zeiger mehr als einen Schritt jenseits der Container-Grenzen zu verschieben.
- $--it$ darf bei einem *iterator* oder *const_iterator* den Container nicht verlassen, nicht einmal um einen einzelnen Schritt.

Durch die polymorphen Iteratoren wurde beginnend mit C++11 eine entsprechende auf Iteratoren basierende polymorphe Form der **for**-Anweisung möglich. Dabei wird

for (⟨for-range-declaration⟩ : ⟨for-range-initializer⟩)
⟨statement⟩

durch den Übersetzer zu folgendem Programmtext expandiert:

```
{
    auto&& __range = ⟨for-range-initializer⟩;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for (; __begin != __end; ++__begin) {
        ⟨for-range-declaration⟩ = *__begin;
        ⟨statement⟩
    }
}
```

(Der expandierte Text wurde Abschnitt 9.5.4 im ISO-Standard 14882:2017 entnommen.)

```
{
    auto&& __range = ⟨for-range-initializer⟩;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for (; __begin != __end; ++__begin) {
        ⟨for-range-declaration⟩ = *__begin;
        ⟨statement⟩
    }
}
```

begin-expr und *end-expr* werden wie folgt bestimmt:

- Bei Arrays wird Zeigerarithmetik verwendet, d.h. *__range* bzw. *__range* + *__bound*, wobei *__bound* die Länge des Arrays ist.
- Wenn *__range* einer Klasse angehört mit den Methoden *begin* und *end*, werden diese verwendet.
- Zuletzt wird nach passenden Funktionen *begin*(*__range*) und *end*(*__range*) gesucht.

sample-lines.cpp

```
class Line: public std::string {
   friend std::istream& operator>>(std::istream& in, Line& line) {
      return std::getline(in, line);
   }
};

int main() {
   std::istream_iterator<Line> it(std::cin);
   std::istream_iterator<Line> it_end;
   ReservoirSampler<Line> r(10, it, it_end);
   for (auto& line: r) {
      std::cout << line << std::endl;
   }
}
```

- Wenn die *ReservoirSampler*-Klasse Iteratoren unterstützt, lässt sich die Anwendung deutlich vereinfachen.
- Um mit stream-basierten Input-Operatoren die Eingabe zeilenweise zu bearbeiten, benötigen wir hier eine abgeleitete Klasse *Line*, für die wir den Eingabe-Operator für *std*::*string* überdefinieren können.

reservoir-sampler.hpp

```
template <typename Iterator>
void add(Iterator it1, Iterator it2) {
   while (it1 != it2) {
      add(*it1++);
   }
}
```

- Mit einer weiteren *add*-Methode kann mit Hilfe von Iteratoren eine Menge von hinzuzufügenden Objekten spezifiziert werden.
- Diese Methode kann auch sogleich von einem entsprechenden Konstruktor verwendet werden:

reservoir-sampler.hpp

```
template<typename Iterator>
ReservoirSampler(std::size_t size, Iterator it1, Iterator it2) :
      ReservoirSampler(size) {
   add(it1, it2);
}
```

reservoir-sampler.hpp

```
using Iterator = const T*;
Iterator begin() const {
   return data;
}
Iterator end() const {
   return data + get_size();
}
```

- Dank der Methoden *begin* und *end* wird das Durchlaufen eines Reservoirs mit einer entsprechenden **for**-Anweisung möglich.
- Wenn die Objekte zusammenhängend als Array im Speicher liegen, dann können auch problemlos Zeiger als Iteratoren verwendet werden.