Vector – Beschreibung der Implementierung

Für den Test im Moped soll Ihre **Vector** Klasse in nur einem Header File (**vector.h**) definiert werden, d.h., eine Aufteilung .h und .cpp File, wie sonst bei Klassen üblich, wird hier nicht verlangt. Die Verwendung von nur einem Header File erleichtert im Weiteren den Übergang zu einer Template Version der **Vector** Klasse.

Zum Test dürfen Sie nur die im Folgenden genannte Basisfunktionalität bestehend aus Grundfunktionalität, Iteratoren und Templates, mitbringen. Insbesondere dürfen keine Lösungen von alten Testangaben, Probetests o. Ä. enthalten sein. Speichern Sie also gegebenenfalls einen Zwischenstand bevor Sie weitere Methoden zu Übungszwecken implementieren.

Beachten Sie die Deadline für den Upload Ihrer Vektorimplementierung

Zum Testen Ihrer Implementierung sollten Sie eigene Testroutinen (z. B. `main`-Files) schreiben. Als Ergänzung werden jede Woche Testfiles bereitgestellt. Beachten Sie, dass diese nur als Hilfestellung gedacht sind und **keine Garantie** darstellen, dass Ihr Vector korrekt funktioniert.

1 Grundfunktionalität

1.1 Instanzvariablen

Die Klasse `Vector` hat folgende **Instanzvariablen**.

size_t sz Enthält die Anzahl der Elemente im **Vector**.

size_t max_sz Enthält die maximale Anzahl an Elementen die möglich sind (Kapazität des **Vector**s).

double* values Zeigt auf ein Feld, welches die Elemente des **Vector**s beinhaltet.

Tipps:

Sie können eine zusätzliche Variable **static constexpr size_t min_sz**, die eine Mindestgröße (z. B. 5) für max_sz festlegt einführen. Ob man leere Vektoren zulässt, oder eine Mindestgröße definiert, ist in unserem Kontext weitestgehend eine Frage des persönlichen Geschmacks. Beide Vorgehensweisen haben ihre Vor- und Nachteile und es gibt spezifische Sonderfälle im Code, die behandelt werden müssen. Diese Sonderfälle treten nur an unterschiedlichen Orten auf.

Sie können ein Typalias **using value_type = double;** definieren und dann überall, wo es passt, **value_type** statt **double** verwenden. Das erleichtert Ihnen später den Übergang zu Templates.

1.2 Konstruktoren/Destruktor

Die Klasse `Vector` hat folgende **Konstruktoren und einen Destruktor**.

Defaultkonstruktor: Liefert einen leeren **Vector**.

Kopierkonstruktor: Liefert einen **Vector** mit demselben Inhalt.

Konstruktoren mit folgenden Parameterlisten

(size_t n): Liefert einen **Vector** mit Platz für n Elemente.

(std::initializer_list<double>): Liefert einen **Vector** mit spezifiziertem Inhalt.

Destruktor: Gibt allozierten Speicher wieder frei.

Vermeiden Sie Speicherlecks und beachten Sie Sonderfälle wie Vector(0) und Vector({}). Diesbezügliche Probleme in Ihrem Code können Sie z. B. mittels `valgrind` ermitteln, welches etwa auf der virtuellen Maschine einfach zu installieren ist.

1.3 Methoden

Die Klasse `Vector` hat folgende **Methoden**.

Kopierzuweisungsoperator: Das this-Objekt übernimmt die Werte vom Parameter. (Notwendig wegen der Verwendung von dynamisch alloziertem Speicher.)

size_t size() const: Retourniert Anzahl der gespeicherten Elemente.

bool empty() const: Retourniert `true` falls der **Vector** leer ist, ansonsten `false`.

void clear(): Löscht alle Elemente aus dem **Vector**.

void reserve(size_t n): Kapazität des **Vector**s wird auf n vergrößert, falls sie nicht schon mindestens so groß ist.

void shrink_to_fit(); Kapazität wird auf Anzahl der Elemente reduziert.

void push_back(double x): Fügt eine Kopie von x am Ende des **Vector**s hinzu.

void pop_back(): Entfernt das letzte Element im **Vector**. Wirft eine Exception, falls der **Vector** leer war.

double& operator[](size_t index) Abgesichert, retourniert das Element an der gegebenen Position (index).

const double& operator[](size_t index) const Abgesichert, retourniert das Element an gegebener Position (index).

size_t capacity() const Retourniert aktuelle Kapazität des **Vector**s.

1.4 Ausgabeformat

Die Klasse `Vector` hat folgendes **Ausgabeformat**, welches **unbedingt** einzuhalten ist.

ostream& operator<<(ostream&, const Vector&) Ausgabe in der Form: [Element1, Element2, Element3].

Beispiel Vector x({1,2,3,4}) → [1, 2, 3, 4]

Verwendung von **friend** für die Implementierung von **operator<<** ist erlaubt.

2 Iteratoren

Damit Iteratoren von STL-Algorithmen verwendet werden können, müssen einige Typ-Aliase für die Iteratoren angelegt werden. Am einfachsten definiert man diese am Anfang der `Vector`-Klasse. Sorgen Sie dafür, dass Ihre `Vector`-Klasse nur noch die Datentypen aus den `using`-Deklarationen verwendet.

```
class Vector{
public:
  class ConstIterator;
  class Iterator;
  using value_type = double;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = value_type*;
  using const_pointer = const value_type*;
  using iterator = Vector::Iterator;
  using const_iterator = Vector::ConstIterator;
private:
  //Instanzvariablen
public:
  //Methoden
class Iterator {
    public:
      using value_type = Vector::value_type;
      using reference = Vector::reference;
      using pointer = Vector::pointer;
      using difference_type = Vector::difference_type;
      using iterator_category = std::forward_iterator_tag;
    private:
    //Instanzvariablen
    public:
    //Methoden
  };
  class ConstIterator {
    public:
      using value_type = Vector::value_type;
      using reference = Vector::const_reference;
      using pointer = Vector::const_pointer;
      using difference_type = Vector::difference_type;
      using iterator_category = std::forward_iterator_tag;
    private:
    //Instanzvariablen
    public:
    //Methoden
  };
};
```

2.1 Erweitern von Vector

Erweitern Sie Ihre **Vector**-Klasse um die Methoden **begin()** und **end()**.

iterator begin() Liefert einen Iterator auf das erste Element im **Vector**. Ist der **Vector** leer, entspricht der gelieferte Iterator dem end-Iterator

iterator end() Liefert einen Iterator auf das virtuelle Element nach dem letzten Element im **Vector**.

const_iterator begin() const Liefert einen Iterator auf das erste Element im **Vector**. Ist der **Vector** leer, entspricht der gelieferte Iterator dem end-Iterator

const_iterator end() const Liefert einen Iterator auf das virtuelle Element nach dem letzten Element im **Vector**.

2.2 Iterator

Die Klasse **Iterator** hat folgende **Instanzvariablen**.

pointer ptr: Zeiger auf ein Element im **Vector**.

Die Klasse **Iterator** hat folgende **Konstruktoren**.

Default: Liefert einen Iterator auf **nullptr**.

Parameterliste (pointer ptr): Liefert einen Iterator, der die Instanzvariable auf **ptr** setzt.

Die Klasse **Iterator** hat folgende **Methoden**. Welche Methoden sollten **const** sein?

reference operator*() const?: Retourniert den Wert des von **ptr** referenzierten Wertes.

pointer operator->() const?: Retourniert einen Zeiger auf den referenzierten Wert.

bool operator==(const const_iterator&) const?: Vergleicht die Zeiger auf Gleichheit. (Eine globale Funktion ist eventuell die bessere Wahl.)

bool operator!=(const const_iterator&) const?: Vergleicht die Zeiger auf Ungleichheit. (Eine globale Funktion ist eventuell die bessere Wahl.)

iterator& operator++() const?: (Prefix) Iterator zeigt auf nächstes Element und (eine Referenz darauf) wird retourniert.

iterator operator++(int) const?: (Postfix) Iterator zeigt auf nächstes Element. Kopie des Iterators vor dem Erhöhen wird retourniert.

operator const_iterator() const?: (Typ-Konversion) Erlaubt die Konvertierung von **Iterator** nach **ConstIterator**.

2.3 ConstIterator

Die Klasse **ConstIterator** hat folgende **Instanzvariablen**.

pointer ptr: Zeiger auf ein Element im Vector.

Die Klasse **ConstIterator** hat folgende **Konstruktoren**.

Default: Liefert einen ConstIterator auf **nullptr**.

Parameterliste (pointer ptr) Liefert einen ConstIterator, der die Instanzvariable auf **ptr** setzt.

Die Klasse **ConstIterator** hat folgende **Methoden**. Welche Methoden sollten **const** sein?

reference operator*() const?: Retourniert den Wert des von **ptr** referenzierten Wertes.

pointer operator->() const?: Retourniert einen Zeiger auf den referenzierten Wert.

bool operator==(const const_iterator&) const?: Vergleicht die Zeiger auf Gleichheit. (Eine globale Funktion ist eventuell die bessere Wahl.)

bool operator!=(const const_iterator&) const?: Vergleicht die Zeiger auf Ungleichheit. (Eine globale Funktion ist eventuell die bessere Wahl.)

const_iterator& operator++() const?: (Prefix) Iterator zeigt auf nächstes Element und (eine Referenz darauf) wird retourniert.

const_iterator operator++(int) const?: (Postfix) Iterator zeigt auf nächstes Element. Kopie des Iterators vor dem Erhöhen wird retourniert.

2.4 Methoden `insert` und `erase`

Die Methoden **insert** und **erase** können von hier kopiert werden.

```
iterator insert(const_iterator pos, const_reference val) {
  auto diff = pos-begin();
  if (diff<0 || static_cast<size_type>(diff)>sz)
    throw std::runtime_error("Iterator out of bounds");
  size_type current{static_cast<size_type>(diff)};
  if (sz>=max_sz)
    reserve(max_sz*2); //Achtung Sonderfall, wenn keine Mindestgroesze definiert ist
  for (auto i {sz}; i-->current;)
    values[i+1]=values[i];
  values[current]=val;
  ++sz;
  return iterator{values+current};
}

iterator erase(const_iterator pos) {
  auto diff = pos-begin();
  if (diff<0 || static_cast<size_type>(diff)>=sz)
    throw std::runtime_error("Iterator out of bounds");
  size_type current{static_cast<size_type>(diff)};
  for (auto i{current}; i<sz-1; ++i)
    values[i]=values[i+1];
  --sz;
  return iterator{values+current};
}
```

Damit die Methoden **insert** und **erase** funktionieren muss

```
friend Vector::difference_type operator-(const Vector::ConstIterator& lop,
                                         const Vector::ConstIterator& rop) {
  return lop.ptr-rop.ptr;
}
```

vorhanden sein.

3 Templates

Um Ihre **Vector**-Klasse zu einem Template zu machen, empfiehlt es sich, wie folgt vorzugehen:

1. Klasse **Vector** wird zu einem Template mit einem Typparameter

   ```
   template<typename T>
   class Vector {...};
   ```

2. Ersetzen des Datentyps `double` als Elementtyp durch `T` (weniger Arbeit, wenn man schon fein säuberlich die Typ-Aliase verwendet hat, ansonsten jetzt eine gute Möglichkeit, das nachzuholen).

3. Die Definitionen der Template-Methoden kommen ebenfalls in die Header-Datei (vector.h). Am einfachsten ist es, wenn die Methoden gleich inline (innerhalb der Klassendefinition) definiert werden. Diese Definitionen werden vom Compiler für die Instanziierung benötigt.

4. Eventuelle Fehler beheben und Testen mit unterschiedlichen Datentypen.