Vector – Implementierung

Für den Test mit der moped-Umbegung muss die **Vector** Klasse in einer einzigen Header Datei (**vector.h**) implementiert werden. Das heißt, die sonst übliche Trennung von Interface (.h) und Implementierung (.cpp) entfällt. Durch die Beschränkung auf eine Header-Datei wird der Übergang zur Template Version der **Vector** Klasse vereinfacht.

Die für den Test hochgeladene Version darf nur die hier beschriebene Funktionalität umfassen, erweitert um Iteratoren und Templates, die in den folgenden Einheiten erarbeitet werden. Insbesondere darf Ihr Code keine zusätzlichen Übungsbeispiele oder Beispiele aus Tests der Vorsemester enthalten. Wenn Sie üben wollen, speichern Sie eine Basisversion, auf die Sie für den Upload zurückgreifen können.

Beachten Sie die Deadline für den Uplöad Ihrer Vektorimplementierung am **21.04.2023**

Sie sollten Ihre Implementierung gründlich testen und dazu womöglich auch eigene Testroutinen entwerfen (z. B. `main` Funktionen). Zur Ünterstützung stellen wir wöchentlich Test-Dateien zur Verfügung, eine **Garantie** auf Korrektheit Ihrer Vektor-Klasse können wir aber nicht geben.

1 Basis-Funktionalität

1.1 Instanzvariablen

Die `Vector` Klasse hat folgende **Instanzvariable**:

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

size_t max_sz: Enthält die maximal mögliche Anzahl von Elementen (Kapazität des **Vector**s).

double* values: Zeigt auf ein (dynamisch alloziertes) Array, in dem der **Vector** die Elemente speichert.

Hinweise:

Sie können ein Minimum für die Kapazität Ihres **Vector**s definieren, das in einer zusätzlichen Klassenvariable **static constexpr size_t min_capacity** festgelegt wird. Ist z. B. **min_capacity** = 5, so muss die Kapazität des **Vector**s (**max_sz**) zu jedem Zeitpunkt mindestens 5 sein. Ob Sie eine minimale Kapazität vorgeben oder **Vector**en erlauben, deren Kapazität 0 ist, ist im Kontext dieser Lehrveranstaltung nur eine Frage des persönlichen Geschmacks. Beide Ansätze haben ihre Vor- und Nachteile und führen zu unterschiedlichen Spezialfällen, die im Code entsprechend behandelt werden müssen. Es ist vorteilhaft, ein Typalias **using value_type = double;** zu definieren und dieses statt **double** zu verwenden, wo immer es passt. Dies erleichtert später den Übergang zu Templates.

1.2 Konstruktoren/Destruktor

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

Default Konstruktor: Liefert einen leeren **Vector**.

Copy Konstruktor: Liefert einen **Vector** mit demselben Inhalt wie der Parameter.

Konstruktoren mit folgenden Parameterlisten:

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

(std::initializer_list<double>): Erzeugt einen **Vector** mit dem spezifizierten Inhalt.

Destruktor: Der Destruktor muss allozierten Speicher wieder freigeben.

Vermeiden Sie memory leaks und achten Sie auf die korrekte Behandlung von Spezialfällen wie Vector(0) and Vector{}. Um Probleme mit der Speicherallokation zu finden, ist `valgrind` ein hilfreiches Tool, das sich auf der virtuellen Maschine einfach installieren lässt.

1.3 Member Funktionen

Die `Vector` Klasse hat folgende **member Funktionen**:

Kopierzuweisungsoperator: Das `this` Object übernimmt eine Kopie des Inhalts des Parameters (Wegen dynamischer Speicherallokation notwendig).

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

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

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

void reserve(size_t n): Wenn die Kapazität des **Vector**s nicht schon zumindest n ist, wird dieser entsprechend vergrößert.

void shrink_to_fit(): Kapazität des **Vector**s wird auf das erforderliche Mindestmaß (Anzahl der aktuell gespeicherten Elemente) reduziert.

void push_back(double x): Eine Kopie von `x` wird zum **Vector** als letztes Element (am Ende) hinzugefügt.

void pop_back(): Entfernt das letzte Element aus dem **Vector**. Wirft eine Exception vom Typ `std::runtime_error`, falls der **Vector** leer war.

double& operator[](size_t index): Retourniert eine Referenz auf das Element an der spezifizierten Position (index). Ist der Index nicht im erlaubten Bereich, muss eine Exception vom Typ `std::runtime_error` geworfen werden.

const double& operator[](size_t index) const: Retourniert eine Referenz auf das Element an der spezifizierten Position (index). Ist der Index nicht im erlaubten Bereich, muss eine Exception vom Typ `std::runtime_error` geworfen werden. (Version für const **Vector**en)

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

1.4 Output Format

Für die `Vector` Klasse ist verpflichtend ein Ausgabeoperator zu realisieren.

ostream& operator<<(ostream&, const Vector&): gibt [Element1, Element2, Element3] aus.

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

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

2 Iteratoren

Um den **Vector** mit STL Algorithmen verwenden zu können, werden Iteratoren und einige Typaliase benötigt. Diese definiert man am besten am Beginn der `Vector` Klasse. Stellen Sie sicher, dass Sie in Ihrer **Vector** Klasse nur die passenden Aliase statt der zugrundeliegenden Typen verwenden.

```
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:
  //Instance variables
public:
  //Member Functions
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:
    //Instance variables
    public:
    //Member Functions
  };
  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:
    //Instance variables
    public:
    //Member Functions
  };
};
```

2.1 Erweiterung der Klasse Vector

Erweitern Sie die **Vector** Klasse mit **begin()** und **end()** Memberfunktionen.

iterator begin(): Retourniert einen Iterator, der auf das erste Element im **Vector** verweist, bzw. gleich dem end-Iterator ist, falls der **Vector** leer ist.

iterator end(): Retourniert einen Iterator, der auf das virtuelle Element hinter dem letzten im **Vector** enthaltenen Element verweist.

const_iterator begin() const: Wie begin() für konstante **Vector**en.

const_iterator end() const: Wie end() für konstante **Vector**en.

2.2 Iterator

Die Klasse **Iterator** hat die folgenden **Instanzvariablen**:

pointer ptr: Zeigt auf das entsprechende Element im **Vector**.

Die Klasse **Iterator** hat die folgenden **Konstruktoren**:

Default: Erzeugt einen Iterator mit dem Wert **nullptr** für die Instanzvariable **ptr**.

Parameterliste (pointer ptr): Erzeugt einen Iterator, bei dem die Instanzvariable **ptr** auf den Wert des Parameters gesetzt wird.

Die Klasse **Iterator** hat die folgenden **Memberfunktionen**: Welche Methoden sollten **const**-Methoden sein?

reference operator*() const?: Retourniert eine Referenz auf den Wert, auf den der Iterator verweist (auf den die Instanzvariable **ptr** zeigt).

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

bool operator==(const const_iterator&) const?: Vergleicht, ob die beiden Pointer gleich sind. (Eine globale Funktion könnte eine bessere Wahl sein.)

bool operator!=(const const_iterator&) const?: Vergleicht, ob die beiden Pointer unterschiedlich sind. (Eine glrobale Funktion könnte eine bessere Wahl sein.)

iterator& operator++() const?: (Prefix) Iterator wird auf das nächste Element im **Vector** weitergeschaltet. Die Methode retourniert eine Referenz auf den veränderten Iterator.

iterator operator++(int) const?: (Postfix) Iterator wird auf das nächste Element in **Vector** weitergeschaltet. Eine Kopie des ursprünglichen Iterators wird retourniert.

operator const_iterator() const?: (Typumwandlung) Erlaubt die Konversion von **Iterator** zu **ConstIterator**.

2.3 ConstIterator

Die Klasse **ConstIterator** hat folgende **Instanzvariablen**:

pointer ptr: Zeigt auf das entsprechende Element im **Vector**.

Die Klasse **ConstIterator** hat die folgenden **Konstruktoren**:

Default: Erzeugt einen ConstIterator mit dem Wert **nullptr** für die Instanzvariable **ptr**.

Parameterliste (pointer ptr): Erzeugt einen ConstIterator, bei dem die Instanzvariable **ptr** auf den Wert des Parameters gesetzt wird.

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

reference operator*() const?: Retourniert eine Referenz auf den Wert, auf den der Iterator verweist (auf den die Instanzvariable **ptr** zeigt).

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

bool operator==(const const_iterator&) const?: Vergleicht, ob die beiden Pointer gleich sind. (Eine globale Funktion könnte eine bessere Wahl sein.)

bool operator!=(const const_iterator&) const?: Vergleicht, ob die beiden Pointer unterschiedlich sind. (Eine glrobale Funktion könnte eine bessere Wahl sein.)

const_iterator& operator++() const?: (Prefix) ConstIterator wird auf das nächste Element im **Vector** weitergeschaltet. Die Methode retourniert eine Referenz auf den veränderten ConstIterator.

const_iterator operator++(int) const?: (Postfix) ConstIterator wird auf das nächste Element in **Vector** weitergeschaltet. Eine Kopie des ursprünglichen ConstIterators wird retourniert.

2.4 Memberfunktionen `insert` and `erase`

Die Memberfunktionen **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); //Attention special case, if no minimum size is defined
  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 **insert** and **erase** so funktionieren, muss auch die folgende Methode implementiert werden:

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

3 Templates

Um Ihre **Vector** Klasse in eine Templateklasse überzuführen, empfiehlt sich folgende Vorgangsweise:

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

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

2. Ersetzen des Datentyps `double` an den geeigneten Stellen durch `T` (wesentlich einfacher, wenn schon die Typaliase verwendet wurden, andernfalls ist es eine gute Idee, das jetzt nachzuholen).

3. Die Definitionen der Templatefunktionen müssen auch in vector.h sein, da sie für die Instanziierung durch den Compiler benötigt werden. Am einfachsten definiert man sie gleich inline (innerhalb der Klassendefinitionen).

4. Mit unterschiedlichen Datentypen testen und gegebenenfalls Fehler beheben.