SWE 4x Übung zu Softwareentwicklung mit modernen Plattformen 4 — SS 2015, Übung 1

Abgabetermin: Ostersamstag in der KW 14

☑ **Gr. 1, E. Pitzer** ☐ **Gr. 2, F. Gruber-Leitner**

Name Stefan Kert **Aufwand in h** 8

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

STL Hashtable (12 + 12 Punkte)

Da wir nun bereits viele Containerklassen der STL kennengelernt haben wird es an der Zeit zu versuchen selbst einen generischen Container zu implementieren. Dazu bietet sich eine einfache Hashtabelle an, die einigermaßen flexibel sein soll. Wir wollen, der Einfachheit halber, lineare Verkettung zur Auflösung von Kollisionen verwenden und nur die wichtigsten Operationen anbieten, sowie die Ausgabe auf einen Stream ermöglichen.

Außerdem wollen wir die eigentliche Hashfunktion sowie die Gleichheit von Elementen konfigurierbar machen. Dazu können wir im ersten Schritt gleich die Funktoren `std::hash()` und `std::equal_to()` der STL verwenden. Da es sich bei diesen Funktionen sowohl um einfache C Funktionszeiger als auch um C++ Funktoren handeln kann, müssen die Typen dieser Funktionen als Templateparameter deklariert werden. Es kommen also zusätzlich zum Templateparameter `V` für den Wertetype (value) noch die Templateparameter `H` für die Hashfunktion sowie `C` für die Vergleichsfunktion (compare) dazu.

Als krönenden Abschluss brauchen wir noch die Implementierung eines einfachen Iterators um auch die bereits bestehenden Algorithmen der STL auf unsere Hashtabelle anwenden zu können.

1. Implementieren Sie dir Grundfunktionalität einer STL-kompatiblen Hashtabelle, die mindestens die folgende Schnittstelle erfüllen muss und testen Sie Ihre Implementierung ausführlich.

```
template<typename V, typename H, typename C>
class hashtable {
public:
  void insert(const V &value);
  void erase(const V &value);
  bool contains(const V &value);
  void rehash(size_t new_n_buckets);

  double load_factor() const;
  size_t size() const;
  size_t capacity() const;
  bool empty() const;
};

template<typename V, typename H, typename C>
std::ostream & operator << (std::ostream & os, const hashtable<V, H, C> &ht);
```

Überlegen Sie sich dazu eine geeignete Komposition aus STL Containern um diese Funktionalität in Form einer Hashtabelle mit linearer Verkettung zu erhalten.

2. Erweitern Sie Ihre Implementierung nun auch um einen einfachen Iterator den Sie z.B. von `std::iterator` ableiten können.

```
typedef std::iterator <std::bidirectional_iterator_tag,
                       value_type,
                       difference_type,
                       const_pointer,
                       const_reference> iterator_base;

class const_iterator : public iterator_base {
public:
  bool operator == (const_iterator const & rhs) const;
  bool operator != (const_iterator const & rhs) const;
  reference operator *  () const;
  pointer   operator -> () const;

  const_iterator & operator ++ ();
  const_iterator & operator -- ();

  const_iterator operator ++ (int);
  const_iterator operator -- (int);
};
```

Abschließend können Sie diesen Iterator dann auch noch verwenden um den Vergleichsoperator für Hashtabellen so zu überschreiben, dass zwei Hashtabellen dann gleich sind, wenn sie die gleichen Elemente enthalten.

```
bool hashtable::operator == (const hashtable &other) const;
```

Hinweis: Um unnötige Stolperfallen im Umgang mit der STL zu vermeiden, verwenden Sie am besten die Vorlage `hashtable_template.hpp` die Sie nur mehr erweitern müssen.

1 Hashtable

1.1 Lösungsidee

1.1.1 Hashtable

Bei der Realisierung der *Hashtable* sollte die vorgegeben Header Datei verwendet werden. Dabei werden alle grundlegenden Methoden implementiert, welche für eine *Hashtable* nötig sind. Die zu implementierenden Funktion sind in der Angabe beschrieben. Alle Elemente die neu hinzugefügt werden innerhalb der Hashtable in folgende Datenstruktur abgespeichert:

- *vector<list<value_type>>*

Der *value_type* ist ein Template Parameter der im beim instanziieren der Klasse gewählt wird. Der Vorteil bei dieser Datenstruktur ist, dass der *vector* die Eigenschaft hat, dass es leicht ist ihn zu vergrößern bzw. zu verkleinern und es die Möglichkeit gibt, auf Elemente innerhalb dieses Vektors mittels Index zuzugreifen. Die Elemente dieses Vektors wurden als *list* implementiert, da diese im Prinzip nur durchlaufen werden und nach dem gewünschten Element durchsucht werden. Um einen möglichst schnellen Zugriff auf die einzelnen Elemente zu ermöglichen, wird mittels einer im Konstruktor übergebenen Hash-Funktion ein Hash generiert. Der Einfachheit halber wird bei dieser Hashtable für die Index des aktuellen Elementes der Wert des Elementes der Hash Funktion übergeben, dieser Hash dann modulo der Größe des Vektors gerechnet, dass ein vorhandener Index gefunden wird. Diese Logik für das Berechnen dieses Hash Wertes sollte in eine eigene Funktion namens *getHashedValue*() ausgelagert werden.Beim Hinzufügen eines Elementes wird dazu vorher überprüft ob das Element bereits in der Hashtable vorhanden ist, wenn nein wird der Hash Wert wie oben beschrieben berechnet und danach mit dem zurückgegebenen Index die Liste im Vektor selektiert welcher das Element angefügt werden sollte. Dieser Liste wird das Element hinten angefügt und anschließend die aktuelle Größe erhöht. Wenn diese Größe einen gewissen Wert übersteigt, in unserem Fall sollte das der *MaxLoadFactor* mal der Kapazität sein, wird die Funktion *rehash* mit der verdoppelten aktuellen Größe aufgerufen. In der Funktion *rehash* sollte die Tabelle vergrößert und anschließend neu befüllt werden. Dazu wird zu Erst eine Kopie der aktuellen Tabelle angelegt, anschließend wird die Kapazität mit der gewünschten Größe überschrieben und die aktuelle Größe auf 0 gesetzt. Die Tabelle wird anschließend auf die neue Kapazität geändert. Dies sollte mit Hilfe der *resize* Funktion geschehen. Diese Tabelle wird anschließend geleert und dann neu befüllt. Aufgrund der Tatsache, dass sich die Größe der Tabelle geändert hat, wird jetzt auch ein neuer Hash Wert für die einzelnen Elemente generiert, daher muss die Tabelle neu befüllt werden.
Die erase Funktion sollte gleich aufgebaut sein wie die insert Funktion. Statt der Überprüfung ob das Element nicht vorhanden ist, sollte überprüft werden, ob das Element vorhanden ist, wenn nicht

kann es auch nicht entfernt werden und es muss daher nichts passieren. Falls ein Element mit dem Wert vorhanden ist, wird die Liste mit dem Element mit dem Wert den wir aus der Funktion *getHashedValue* bekommen (wie oben beschrieben) selektiert. Da wir uns sicher sein können, dass der Wert in dieser Liste existiert, brauchen wir anschließend nur noch die *remove* Funktion auf die selektierte Liste aufrufen. Danach wird die aktuelle Größe wieder um eines verringert. Wenn diese Größe unter einen bestimmten Wert sinkt, muss wieder die *rehash* Funktion aufgerufen werden. Der bestimmte Wert ergibt sich dabei aus der Multiplikation von *MinLoadFactor* und der aktuellen Kapazität. Die Funktion *rehash* wird dabei mit der halbierten aktuellen Größe aufgerufen und funktioniert wie oben beschrieben.
Die *contains* Funktion sollte feststellen, ob ein bestimmtes Element in der Tabelle vorhanden ist. Dazu wird über die *getHashedValue* der Hash Wert für das aktuelle Element generiert, über diesen Wert wird schließlich die Liste aus dem Vektor selektiert und diese Liste wird danach durchsucht ob der Wert enthalten ist. Wenn das Ergebnis dieser Suche dem End-Iterator der Liste entspricht existiert es nicht ansonsten gibt es ein Element.
Bei der Überladung des *== Operators* sollte zu Erst überprüft werden ob die übergebene Tabelle die gleiche Adresse hat, wie die aktuelle Klasse. Dann ist es in jedem Fall das gleiche Objekt bzw. hat die gleichen Elemente. Wenn die Größe der übergebenen Tabelle sich von der aktuellen Tabelle unterscheidet kann in jedem Fall gesagt werden, dass die Tabellen unterschiedlich sind. Falls diese beiden Fälle nicht eintreten, sollte die übergebene Liste in einer Schleife durchlaufen werden und für jedes Element dieser Tabelle überprüft werden, ob es im aktuellen Objekt vorhanden ist. Falls nicht kann davon ausgegangen werden, dass es nicht vorhanden ist. Falls die Schleife nicht abgebrochen wurde, sind alle Elemente der übergebenen Tabelle vorhanden und es kann daher davon ausgegangen werden, dass diese Tabellen zumindest die gleichen Elemente besitzen. Für diese Funktion wird es nötig sein einen Iterator zu implementieren. Wie dieser aussehen sollte wird weiter unten beschrieben.
Die Überladung des *<< Operators* geht in einer Schleife alle Elemente der übergebenen Tabelle durch und gibt diese auf dem übergebenen *ostream* aus. Auch diese Funktion benötigt die Implementierung des Iterators.
Die weiteren Funktionen sind trivial und sollten entweder einzelne Werte zurückgeben oder eine Kombination dieser.

1.1.2 Iterator

Bei der Implementierung des Iterators für die Hashtable sollte darauf geachtet werden, dass alle Elemente nacheinander durchlaufen werden. Das heißt, wir müssen nacheinander alle Listen aus dem Vektor selektieren und die einzelnen darin enthaltenen Elemente selektieren. Beim

instanziieren des Iterators sollte hierzu auf das erste gültige Element in der Hashtable verwiesen werden. Das erste gültige Element wird dabei dadurch gefunden, dass man mittels vector.begin() startet und solange die zurückgegebene Liste leer ist den dabei zurückgegeben Iterator erhöht, bis die erste Liste kommt welche mindestens ein Element enthält. Sobald dieses Gefunden wird, werden zwei Membervariablen in der Iterator Klasse gesetzt. Diese Funktionalität sollte in der *begin* Funktion realisiert werden. Falls das Ende der Hashtable gewünscht ist, wird der gleiche Prozess durchgeführt, nur wird vom Ende des Vektors ausgegangen und so lange verringert bis die aktuelle Liste mindestens ein Element enthält. Die Membervariablen werden dann auf diesen Iterator mit der ersten Liste gesetzt und der zweite Iterator wird auf das letzte Element dieser Liste gesetzt. Diese Funktionalität sollte in einer Funktion end() realisiert werden.

Falls auf den Iterator der Hashtable *operator++* augerufen wird, wird zu Erst der Listeniterator erhöht. Wenn dieser am Ende der Liste angekommen ist, dann wird der Vektoriterator erhöht und der Listeniterator danach auf den Anfang der aktuellen Liste gesetzt. Dieser Vorgang wird so lange durchgeführt bis eine Liste gefunden wurde die mindestens ein Element hat.

Wenn *operator--* aufgerufen wird, sollte zu Erst überprüft ob der *listIterator* dem Beginn des Vektoriterators entspricht. Wenn nicht sollte er verringert werden. Falls diese Aussage wahr ist, wird der *vectorIterator* solange verringert, bis er entweder dem Anfang des Vektors entspricht oder die Liste auf die der Vektoriterator verweist eine Größe größer als 0 hat. Falls der Vektoriterator also dem Beginn des Vektors entspricht sollte die oben beschrieben Funktion *begin* aufgerufen werden. Falls nicht, wird der *listIterator* an das Ende des Vektoriterators gesetzt und um eins verringert.

Bei der Überladung des == Operators sollte überprüft werden ob der *vectorIterator* dem übergeben *vectorIterator* entspricht und ob der *listIterator* dem übergebenen entspricht.
Bei dem != Operator sollte überprüft werden ob der *vectorIterator* ungleich dem übergeben *vectorIterator* ist oder der *listIterator* ungleich dem übergebenen entspricht.
Die Funktionen für die *reference* und den *pointer* sind trivial und benötigen daher keine nähere Beschreibung.
Es sollten auch die *begin*() *end*() bzw. die *cbegin*() und *cend*() Funktionen für die Hashtable implementiert werden. Dabei sollte für die *begin* Funktion einfach eine Instanz des zu implementierenden *const_iterators* erstellt weden und bei der *end* Funktion ebenfalls. Der Unterschied ist, dass bei der end Funktion der Iterator noch an das Ende gesetzt wird. Die Funktionen *cbegin* und *cend* sowie deren Überladungen rufen im Prinzip nur *begin* und end auf.

1.2 Tests

1.2.1 Check Int Insert

```
CheckIntInsert
Int List contains element 0: true
Int List contains element 1: true
Int List contains element 2: true
Int List contains element 3: true
Int List contains element 4: true
Expected Size 5 || Actual Size 5
Expected Capacity 10 || Actual Capacity 10
```

1.2.2 Check String Insert

```
CheckStringInsert
String List contains element '0': true
String List contains element '1': true
String List contains element '2': true
String List contains element '3': true
String List contains element '4': true
Expected Size 5 || Actual Size 5
Expected Capacity 10 || Actual Capacity 10
```

1.2.3 Check Int Operator==

```
CheckIntOperatorEquals
Int Lists are the same: true
```

1.2.4 Check String Operator==

```
CheckStringOperatorEquals
String Lists are the same: true
```

1.2.5 Check Erase Int

```
CheckEraseInt
Int List contains Value 5 before erase: true
Int List contains Value 5 after erase: false
```

1.2.6 Check Erase String

```
CheckEraseString
String List contains Value '5' before erase: true
String List contains Value '5' after erase: false
```

1.2.7 Check Print Int List With Operator<<

```
CheckPrintIntListWithOverloadedOperator
0
1
2
3
4
5
6
7
8
9
```

1.2.8 Check Print String List With Operator<<

```
CheckPrintStringListWithOverloadedOperator
7
9
6
1
4
5
3
2
8
0
```

1.2.9 Print List Normal and reversed Order

```
PrintListNormalAndReverseOrder

Normal order:
0
1
2
3
4
5
6
7
8
9
Reverse order:
9
8
7
6
5
4
3
2
1
0
```

1.2.10 Print List With Foreach STL

```
PrintListWithForeachWithSTL
0
1
2
3
4
5
6
7
8
9
```

1.2.11 Find Element With STL

```
FindElementWithSTL
Should find element '5': Found true
Should not find element '15': Found false
```

1.3 Quelltext

1.3.1 Hashtable.hpp

```
#ifndef hashtable_hpp
#define hashtable_hpp

#include <cassert>
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <iomanip>

using namespace std;

template<typename V, typename H, typename C>
class hashtable;

template<typename V, typename H, typename C>
std::ostream & operator << (std::ostream & os, const hashtable<V, H, C> &ht);

template<typename V, typename H, typename C>
class hashtable {
	friend std::ostream & operator << <V, H, C>(std::ostream & os, const
hashtable<V, H, C> &ht);

public:
	typedef V value_type;
	typedef H hash_function_type;
	typedef C key_equal_function_type;
	typedef unsigned int size_t;
	typedef value_type const * const_pointer;
	typedef value_type const & const_reference;
	typedef ptrdiff_t     difference_type;
	typedef const_pointer      pointer;
	typedef const_reference    reference;
	typedef size_t        size_type;

	hashtable(size_t n_buckets = 10,
		hash_function_type hasher = hash<V>(),
		key_equal_function_type equals = equal_to<V>(),
		double max_load_factor = 0.8,
		double min_load_factor = 0.2)
		: _capacity(n_buckets), _hasher(hasher), _equals(equals),
_maxLoadFactor(max_load_factor), _minLoadFactor(min_load_factor)
	{
		rehash(_capacity);
	}

	virtual ~hashtable() {}

	void insert(const V &value);
	void erase(const V &value);
	bool contains(const V &value) const;
	void rehash(size_t new_n_buckets);
	double load_factor() const;
	size_t size() const;
	size_t capacity() const;
	bool empty() const;
	bool operator == (const hashtable &other) const;
```

```
	typedef std::iterator <bidirectional_iterator_tag,
		value_type,
		difference_type,
		const_pointer,
		const_reference> iterator_base;

	class const_iterator : public iterator_base {

	private:
		const hashtable* _table;
		typename vector<list<value_type>>::const_iterator _vectorIterator;
		typename list<value_type>::const_iterator _listIterator;
	public:
		typedef typename iterator_base::difference_type   difference_type;
		typedef typename iterator_base::iterator_category iterator_category;
		typedef typename iterator_base::pointer           pointer;
		typedef typename iterator_base::reference         reference;

		const_iterator(const hashtable *table);

		bool operator == (const_iterator const & rhs) const;
		bool operator != (const_iterator const & rhs) const;

		reference operator *  () const;
		pointer   operator -> () const;

		void begin();
		void end();
		void next();
		void previous();

		const_iterator & operator ++ ();
		const_iterator & operator -- ();

		const_iterator operator ++ (int unused);
		const_iterator operator -- (int unused);
	};

	typedef const_iterator iterator;

	const_iterator begin() const;
	const_iterator end() const;

	iterator cbegin();
	iterator cend();

	const_iterator cbegin() const;
	const_iterator cend() const;

private :
	vector<list<V>> _table;
	size_t _capacity;
	size_t _currentSize;
	double _minLoadFactor;
	double _maxLoadFactor;
	hash_function_type _hasher;
	key_equal_function_type _equals;
	size_t getHashedValue(const V& value) const;
};

/*
```

```
/*----------------------------------------------------------------------------------------
---------------------------------------
*	Hashtable Implementation
*----------------------------------------------------------------------------------------
---------------------------------------
*/

template<typename V, typename H, typename C>
ostream & operator << (ostream & os, const hashtable<V, H, C> &ht){
	for (auto element : ht){
			os << element << endl;
	}
	return os;
}

template<typename V, typename H, typename C>
bool hashtable<V, H, C>::operator == (const hashtable &other) const{
	if (&other == this)
		return true;
	if (other.size() != this->size())
		return false;

	for (auto element : other){
		if (!this->contains(element)){
			return false;
		}
	}
	return true;
}

template <typename V, typename H, typename C>
bool hashtable<V, H, C>::contains(const V &value) const {
	const list<value_type>& whichList = _table[getHashedValue(value)];
	return find(whichList.begin(), whichList.end(), value) != whichList.end();
}

template <typename V, typename H, typename C>
double hashtable<V, H, C>::load_factor() const {
	return _currentSize / _capacity;
}

template <typename V, typename H, typename C>
size_t hashtable<V, H, C>::size() const {
	return _currentSize;
}

template <typename V, typename H, typename C>
size_t hashtable<V, H, C>::capacity() const {
	return _capacity;
}

template <typename V, typename H, typename C>
bool hashtable<V, H, C>::empty() const {
	return _table.empty();
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::insert(const V &value) {
	if (!contains(value)){
		list<value_type>& whichList = _table[getHashedValue(value)];
		whichList.push_back(value);
		if (++_currentSize > (_maxLoadFactor * _capacity))
```

```
                        rehash(static_cast<size_t>(_currentSize * 2));
        }
        else{
                // Element is already available in hashtable so there is no need to add
it
        }
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::erase(const V &value){
        if (contains(value)){
                list<value_type>& whichList = _table[getHashedValue(value)];
                whichList.remove(value);
                if (--_currentSize > _minLoadFactor * _capacity){
                        rehash(static_cast<size_t>(_currentSize * 0.5));
                }
        }
        else{
                // Element is not existent in hashtable so there is no need to remove it
        }
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::rehash(size_t new_n_buckets){
        vector<list<V>> old = _table;
        _capacity = new_n_buckets;
        _currentSize = 0;
        _table.resize(_capacity);
        for (size_t i = 0; i < _table.size(); i++) {
                _table[i].clear();
        }

        for (auto elements : old) {
                for (auto element : elements) {
                        insert(element);
                }
        }
}

template <typename V, typename H, typename C>
size_t hashtable<V, H, C>::getHashedValue(const V& value) const{
        size_t hashValue = _hasher(value) % _capacity;
        if (hashValue < 0){
                hashValue += _table.size();
        }
        return hashValue;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H, C>::begin() const{
        return const_iterator(this);
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H, C>::end() const {
        auto it = const_iterator(this);
        it.end();
        return it;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::iterator hashtable<V, H, C>::cbegin() {
```

```
	return begin();
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::iterator hashtable<V, H, C>::cend(){
	return end();
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H, C>::cbegin() const {
	return begin();
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H, C>::cend() const{
	return end();
}

/*
*-----------------------------------------------------------------------------------------
---------------------------------------
*	Iterator Implementation
*-----------------------------------------------------------------------------------------
---------------------------------------
*/

template <typename V, typename H, typename C>
hashtable<V, H, C>::const_iterator::const_iterator(const hashtable *table) :
_table(table){
	begin();
}

template <typename V, typename H, typename C>
bool hashtable<V, H, C>::const_iterator::operator == (const_iterator const & rhs)
const{
	return _vectorIterator == rhs._vectorIterator && _listIterator ==
rhs._listIterator;
}

template <typename V, typename H, typename C>
bool hashtable<V, H, C>::const_iterator::operator != (const_iterator const & rhs)
const{
	return _vectorIterator != rhs._vectorIterator || _listIterator !=
rhs._listIterator;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator::reference hashtable<V, H,
C>::const_iterator::operator *  () const{
	return *_listIterator;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator::pointer   hashtable<V, H,
C>::const_iterator::operator -> () const{
	return &(*_listIterator);
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::const_iterator::begin(){
	_vectorIterator = _table->_table.begin();
	while (_vectorIterator != _table->_table.end() && _vectorIterator->size() == 0)
```

```
		++_vectorIterator;
	_listIterator = _vectorIterator->begin();
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::const_iterator::end(){
	_vectorIterator = --_table->_table.end();
	while (_vectorIterator != _table->_table.begin() && _vectorIterator->size() ==
0)
		--_vectorIterator;

	_listIterator = _vectorIterator->end();
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::const_iterator::next(){
	if (++_listIterator == _vectorIterator->end()){
		do {
			++_vectorIterator;
		} while (_vectorIterator != _table->_table.end() && _vectorIterator-
>size() == 0);
		if (_vectorIterator == _table->_table.end()){
			end();
		}
		else{
			_listIterator = _vectorIterator->begin();
		}
	}
}

template <typename V, typename H, typename C>
void hashtable<V, H, C>::const_iterator::previous(){
	if (_listIterator != _vectorIterator->begin()){
		--_listIterator;
	}
	else{
		do {
			--_vectorIterator;
		} while (_vectorIterator != _table->_table.begin() && _vectorIterator-
>size() == 0);
		if (_vectorIterator == _table->_table.begin()){
			begin();
		}
		else{
			_listIterator = _vectorIterator->end();
			--_listIterator;
		}
	}
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator & hashtable<V, H,
C>::const_iterator::operator ++ (){
	next();
	return *this;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator & hashtable<V, H,
C>::const_iterator::operator -- (){
	previous();
	return *this;
```

```
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H,
C>::const_iterator::operator ++ (int unused){
	next();
	return *this;
}

template <typename V, typename H, typename C>
typename hashtable<V, H, C>::const_iterator hashtable<V, H,
C>::const_iterator::operator -- (int unused){
	previous();
	return *this;
}

#endif // hashtable_hpp
```

1.3.2 Main.cpp

```
#include <cassert>
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <iomanip>
#include <algorithm>
#include <sstream>
#include "hashtable.hpp"

using namespace std;

void CheckIntInsert(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 5; i++){
		h.insert(i);
		cout << "Int List contains element " << i << ": " << (h.contains(i) ?
"true" : "false") << endl;
	}
	cout << "Expected Size 5 || Actual Size " << h.size() << endl;
	cout << "Expected Capacity 10 || Actual Capacity " << h.capacity() << endl;
}
void CheckStringInsert(){
	hashtable<string, hash<string>, equal_to<string>> h(5, hash<string>(),
equal_to<string>());
	for (int i = 0; i < 5; i++){
		h.insert(to_string(i));
		cout << "String List contains element '" << i << "': " <<
(h.contains(to_string(i)) ? "true" : "false") << endl;
	}
	cout << "Expected Size 5 || Actual Size " << h.size() << endl;
	cout << "Expected Capacity 10 || Actual Capacity " << h.capacity() << endl;
}

void CheckEraseInt(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
	}
	cout << "Int List contains Value 5 before erase: " << (h.contains(5) ? "true" :
"false") << endl;
```

```
	h.erase(5);
	cout << "Int List contains Value 5 after erase: " << (h.contains(5) ? "true" :
"false") << endl;
}
void CheckEraseString(){
	hashtable<string, hash<string>, equal_to<string>> h(5, hash<string>(),
equal_to<string>());
	for (int i = 0; i < 10; i++){
		h.insert(to_string(i));
	}
	cout << "String List contains Value '5' before erase: " <<
(h.contains(to_string(5)) ? "true" : "false") << endl;
	h.erase(to_string(5));
	cout << "String List contains Value '5' after erase: " <<
(h.contains(to_string(5)) ? "true" : "false") << endl;
}

void CheckIntOperatorEquals(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	hashtable<int, hash<int>, equal_to<int>> h2(100, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
		h2.insert(i);
	}
	cout << "Int Lists are the same: " << (h == h2 ? "true" : "false") << endl;
}
void CheckStringOperatorEquals(){
	hashtable<string, hash<string>, equal_to<string>> h(5, hash<string>(),
equal_to<string>());
	hashtable<string, hash<string>, equal_to<string>> h2(100, hash<string>(),
equal_to<string>());
	for (int i = 0; i < 10; i++){
		h.insert(to_string(i));
		h2.insert(to_string(i));
	}
	cout << "String Lists are the same: " << (h == h2 ? "true" : "false") << endl;
}

void CheckPrintIntListWithOverloadedOperator(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
	}
	cout << h << endl;
}
void CheckPrintStringListWithOverloadedOperator(){
	hashtable<string, hash<string>, equal_to<string>> h(5, hash<string>(),
equal_to<string>());
	for (int i = 0; i < 10; i++){
		h.insert(to_string(i));
	}
	cout << h << endl;
}

void PrintListNormalAndReverseOrder(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
	}
	auto it = h.begin();
	cout << "Normal order: " << endl;
	while (it != h.end()){
```

```
		cout << (*it) << endl;
		it++;
	}

	cout << "Reverse order: " << endl;
	auto reverseIt = h.end();
	do{
		--reverseIt;
		cout << (*reverseIt) << endl;
	} while (reverseIt != h.begin());
}

void PrintListWithForeachWithSTL(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
	}
	for_each(h.begin(), h.end(), [](const int& element){
		cout << element << endl;
	});
}
void FindElementWithSTL(){
	hashtable<int, hash<int>, equal_to<int>> h(5, hash<int>(), equal_to<int>());
	for (int i = 0; i < 10; i++){
		h.insert(i);
	}
	auto foundIt = find(h.begin(), h.end(), 5);
	cout << "Should find element '5': Found " << (foundIt != h.end() ? "true" :
"false") << endl;
	auto foundItNot = find(h.begin(), h.end(), 15);
	cout << "Should not find element '15': Found " << (foundItNot != h.end() ?
"true" : "false") << endl;
}

int main(int argc, char** argv) {
	cout << "CheckIntInsert" << endl;
	CheckIntInsert();
	cout << endl;
	cout << "CheckStringInsert" << endl;
	CheckStringInsert();
	cout << endl;
	cout << "CheckIntOperatorEquals" << endl;
	CheckIntOperatorEquals();
	cout << endl;
	cout << "CheckStringOperatorEquals" << endl;
	CheckStringOperatorEquals();
	cout << endl;
	cout << "CheckEraseInt" << endl;
	CheckEraseInt();
	cout << endl;
	cout << "CheckEraseString" << endl;
	CheckEraseString();
	cout << endl;
	cout << "CheckPrintIntListWithOverloadedOperator" << endl;
	CheckPrintIntListWithOverloadedOperator();
	cout << endl;
	cout << "CheckPrintStringListWithOverloadedOperator" << endl;
	CheckPrintStringListWithOverloadedOperator();
	cout << "PrintListNormalAndReverseOrder" << endl;
	cout << endl;
	PrintListNormalAndReverseOrder();
	cout << endl;
```

```
	cout << " PrintListWithForeachWithSTL " << endl;
	PrintListWithForeachWithSTL ();
	cout << endl;
	cout << " FindElementWithSTL " << endl;
	FindElementWithSTL();
	cout << endl;
	return 0;
}
```