Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christoph Stamm, Lucas Braun christoph.stamm@fhnw.ch Tel.: 056 202 78 32

# Übung 2: Set-Klasse

**Lernziele**

- Sie repetieren den Stoff aus der Vorlesung: Struktur eines C++-Programms, einfache und strukturierte Datentypen, Zeiger, Referenzen, Arrays, Klassen, Konstruktoren/Destruktoren, Verschiebesemantik und einfache Vererbung.
- Sie entwickeln eine eigene Set-Klasse für mathematische Mengen von Ganzzahlen ähnlich wie die Set-Klassen in Java.
- Sie schreiben eine kleine Konsolenanwendung und verwenden dabei Ihre Set-Klasse.
- Sie testen Ihre Set-Klasse mit Unit-Tests.
- Sie verstehen das Potential der Move-Semantik und von shared pointers und die Auswirkungen auf die Performance.
- Sie verstehen und üben das Prinzip der Vererbung und wenden dieses an, um Ihre Set-Klasse zu erweitern.

## 1 Tutorial und Aufgaben

In dieser Übung sollen Sie schrittweise eine eigene Set-Klasse für unveränderbare Mengen von Ganzzahlen entwickeln und diese eingehend testen. Diese Klasse wird im Wesentlichen das Set-Interface aus Java implementieren. Dabei lernen Sie verschiedenste (neue) C++-Techniken, statische Bibliotheken, den Debugger und das Unit-Testsystem von Visual Studio[1] kennen.

Zur Erklärung gehen wir von folgender Dateistruktur aus:

\Uebung2

- \Console
- \MySet
- \UnitTest

Der Ordner „Uebung2“ enthält die Solution-Datei „Uebung2.sln“ und die Ordner „Console“, „MySet“ und „UnitTest“ die entsprechenden Projekt- und Quellcode-Dateien. Diese Dateistruktur müssen Sie nicht selber erzeugen, sie wird mithilfe der „Project wizards“ von VS automatisch erstellt werden.

### 1.1 Mengen

Eine mathematische Menge ist definiert durch die Elemente, welche sie enthält.[2] Während es in der Mathematik endliche und unendliche Mengen (wie beispielsweise die Menge der natürlichen Zahlen) gibt, ist es beim Programmieren sinnvoll, sich auf endliche Mengen zu beschränken. In dieser Übung schreiben wir eine Klasse, welche eine endliche Menge von ganzen Zahlen (`int`) repräsentiert. Die Mathematik definiert einige Operationen, um aus bestehenden Mengen neue zu bilden, beispielsweise die Vereinigungsmenge, die Schnittmenge und die Differenzmenge. Eine Bündelung der Mengenoperationen zu einer Klasse Set ist also sinnvoll. Sowohl Java als auch C++ haben Set-Implementierungen in ihren Standard-Bibliotheken, wobei bei Java die klassischen Mengenoperationen implementiert sind, während in C++ ein Set lediglich ein Container ist, bei dem jedes Element höchstens einmal vorkommen darf. In Java und C++ werden veränderbare Mengen repräsentiert, während Sie nun in der vorliegenden eine Klasse für unveränderbare Mengen entwickeln werden.

### 1.2 Set-Klasse

Erstellen Sie ein neues C++ Win32-Projekt mit dem Namen „MySet“ und einem **separaten Verzeichnis für die Solution**. Wählen Sie bei den „Application Settings“ im „Project Wizard“ den „Application type“ „**Static library**“. Auf „Security Development Lifecycle (SDL) checks“ und „precompiled headers“ können Sie verzichten. Der Solution werden wir später noch eine Konsolenanwendung und ein Test-

[1] Die hier gemachten Beschreibungen beziehen sich auf Microsoft Visual Studio Enterprise 2015 (VS). Sie müssen nicht unbedingt mit Visual Studio arbeiten. Sie benötigen jedoch ein Werkzeug, welches Unit-Tests und Code-Coverage-Analysis ermöglicht.

[2] https://de.wikipedia.org/wiki/Menge_(Mathematik)

projekt hinzufügen. Fügen Sie dem aktuellen Projekt in VS dann zwei Dateien hinzu: „MySet.h“ und „MySet.cpp“[3].

Definieren Sie in der Datei „MySet.h“ eine eigene Set-Klasse. Da wir später von der Set-Klasse erben möchten, machen wir die Klasse nicht final und setzen die Sichtbarkeit ihrer Attribute auf protected (und nicht private).

```
class Set {
protected:
   // protected Datenstruktur (erlaubt Vererbung)
public:
   // öffentliche Methoden
};
```

### 1.2.1 Datenstruktururen

Unsere Set-Datenstruktur braucht offensichtlich ein Attribut, welches die Elemente der Menge speichert. Das heisst, sie soll ein nicht veränderbares `int`-Array (`m_values`) verwalten.[4] Wir verwenden `int`, weil wir Mengen von Ganzzahlen speichern möchten, man könnte `int` aber problemlos durch beliebige andere Datentypen ersetzen (solange sie eine Vergleichsfunktion besitzen) und sogar durch einen Template-Parameter. Zur Speicherung des Arrays könnte ein C++-Container verwendet werden. Um den Umgang mit Arrays und Zeigerobjekten zu üben, verwenden wir hier ein herkömmliches C-Array. Bekanntlich speichern C-Arrays die Arraylänge nicht im Array. Daher müssen wir in unserer Datenstruktur ein Grössenfeld `m_size` vom Typ `size_t`[5] vorsehen.

C-Arrays können in C/C++ ganz einfach durch einen rohen Zeiger vom Typ `int*` repräsentiert werden. Ein solcher Rohzeiger wäre hier aber mit grosser Vorsicht zu verwenden, weil mehrere Set-Objekte auf das gleiche C-Array verweisen dürfen (beim Kopieren eines Set-Objektes möchten wir aus Effizienzgründen kein neues C-Array erstellen, sondern nur den Zeiger auf das bereits bestehende Array auf dem Heap zeigen lassen. Hier bietet sich also für `m_values` eines der neuen Zeigerobjekte von C++11 bestens an: `shared_ptr<int>`. Dieses smarte Zeigerobjekt registriert, wenn der Zeiger kopiert wird. Dadurch wird sichergestellt, dass das Array erst dann vom Heap gelöscht wird, wenn kein Set-Objekt mehr darauf verweist.

### 1.2.2 Konstruktoren und Destruktor

Als erstes programmieren wir einen nicht-öffentlichen (protected) Konstruktor, der als Eingabe einfach die maximale Grösse (als `size_t`) bekommt, die unsere Menge haben wird. Die Idee dieses Konstruktors ist, zunächst einmal ein nicht initialisiertes Array der entsprechenden Grösse auf dem Heap zu erstellen und `m_values` darauf zeigen zu lassen (die Grösse des Sets bleibt aber immer noch 0). Das Array wird dann von anderen Konstruktoren gefüllt werden. Der kritische Punkt in diesem Konstruktor ist die korrekte Erzeugung des Zeigerobjekts `shared_ptr<int> m_values`. Dieses Zeigerobjekt verwaltet streng genommen nur einen Zeiger auf einen einzelnen `int` und nicht auf ein ganzes `int`-Array, denn der generische Typ des Zeigerobjekts ist `int`. Leider ist es (noch) nicht möglich, einen `shared_ptr<int[]>` (beachten Sie die eckigen Klammern hinter `int`) zu verwenden. Dieser Umstand wird spätestens beim automatischen Speicherabbau zum Problem, weil dann nur ein einzelner `int` auf dem Heap freigegeben wird und nicht das ganze Array. Um dieses Problem zu lösen, kann dem Konstruktor von `shared_ptr<>` ein zweites Argument, ein sogenannter Deleter, mitgegeben werden, welcher dann für den korrekten Speicherabbau zuständig ist. Ein Deleter ist eine Methode, welche im Destruktor der zugehörigen Klasse (in diesem Fall `shared_ptr<int>`) aufgerufen wird. In unserem Fall können wir den Standard-Deleter für Arrays verwenden und bekommen somit folgenden Konstruktor:

```
Set(size_t size) : m_values(new int[size], default_delete<int[]>()), m_size(0) {
       cout << "private-ctor " << endl;
}
```

---

[3] Ihre Klasse soll schlicht und einfach `Set` heissen. Um Verwechslungen mit vorhandenen Header-Files aus der Standard-Bibliothek zu vermeiden zu vermeiden, verwenden wir für die Dateien den Namen MySet. In C++ muss die Datei nicht den Namen der Klasse tragen.

[4] In der Realität wird für eine Mengen-Implementierung normalerweise kein simples Array, sondern eine Hash-Tabelle oder ein binärer Suchbaum verwendet. Da es in dieser Übung aber nicht um komplexe Datenstrukturen, sondern vielmehr um das Programmieren einer funktionierenden Lösung in C++ geht, ist ein simples Array vorerst in Ordnung.

[5] `size_t` ist ein vorzeichenloser Typ für ganzzahlige Grössenangaben

Zusätzlich zu diesem nicht öffentlichen, wollen wir noch drei weitere, öffentliche Konstruktoren anbieten:

- Der Standardkonstruktor benötigt keine Argumente und repräsentiert eine leere Menge (Grösse = 0). Ein Array muss daher nicht alloziert werden.
- Der Kopierkonstruktor soll eine flache Kopie erstellen, denn es kann problemlos das gleiche Array wiederverwendet werden. Grundsätzlich könnte der vom System automatisch erstelle Kopierkonstruktor verwendet werden. Damit Sie besser verstehen, wann dieser Konstruktor aufgerufen wird und hinein debuggen können, empfehlen wir Ihnen, diesen selber zu programmieren und eventuell mit einer Konsolenausgabe zu versehen.
- Der Typkonvertierungs-Konstruktor nimmt ein herkömmliches C-Array vom Typ `const int*`, sowie die Länge dieses Arrays (als `size_t`) entgegen und erzeugt eine tiefe Kopie, da unser Set-Objekt unabhängig vom Übergabeparameter bleiben soll. Achten Sie darauf, dass eine Menge jedes Element nur einmal enthalten kann. Deswegen sollten Sie nicht einfach eine simple Kopie des C-Arrays erstellen, sondern Elemente, die mehrfach vorkommen, entfernen. Eine Möglichkeit dies zu erreichen, ist als erstes den nicht-öffentlichen Konstruktor aufzurufen, um auf dem Heap ein neues, genügend grosses Array zu erstellen. Danach kopieren Sie die Zahlen einzeln vom Eingabe-Array nach `m_values`, aber nur, wenn eine Zahl noch nicht im neuen Array enthalten ist. Vergessen Sie nicht `m_size` entsprechend anzupassen.

Bleibt schliesslich noch der Destruktor. Was gibt es hier zu tun? Was wäre anders, wenn wir einen rohen Zeiger anstatt dem `shared_ptr<int>` für das Array in unserer Datenstruktur verwendet hätten? Auch hier empfehlen wir Ihnen, diesen Destruktor aus didaktischen Gründen zu implementieren, damit Sie hineindebuggen und überprüfen können, ob wirklich das Array freigegeben wird.

### 1.2.3 Methoden

Bevor Sie die Methoden der Klasse implementieren, empfehlen wir Ihnen Abschnitt **Error! Reference source not found.** durchzuarbeiten. Dort entwickeln Sie eine kleine Konsolenanwendung, welche die Set-Klasse verwendet.

Ihre Set-Klasse soll die nachfolgenden Instanz- und Klassen-Methoden anbieten. Beachten Sie, dass alle Instanzmethoden ein `const` am Ende haben. Damit wird ausgedrückt, dass diese Methoden das this-Objekt nicht verändern, also nur lesenden Zugriff haben. Die Methodendeklarationen entsprechen im Grossen und Ganzen jenen aus der Java-Dokumentation für Mengen.[6]

```
protected:
// geschützte Instanzmethoden
int *begin() const;                          // gibt einen Zeiger auf das erste Element
                                             // der Menge zurueck (nullptr falls leer)

int& operator[](size_t i);                   // gibt das i-te Element des Mengen-Arrays
                                             // zurueck

Set merge(const Set& set) const;             // gibt als neue Menge die Vereinigungs-
                                             // menge dieser Menge mit set zurueck
Set difference(const Set& set) const;        // gibt als neue Menge die Differenzmenge
                                             // zwischen set und dieser Menge
                                             // (set \ this) zurueck
Set intersection(const Set& set) const;      // gibt als neue Menge die Schnittmenge
                                             // dieser Menge mit set zurueck

public:
// öffentliche Methoden


  // Konstruktoren und Destruktor
  ...
  // Instanzmethoden
bool contains(int e) const;                  // testet, ob die Menge e enthaelt
                                             // testet, ob diese Menge alle Elemente der
                                             // anderen Menge enthaelt (und somit
                                             // dessen Supermenge ist)
```

[6] https://docs.oracle.com/javase/8/docs/api/java/util/Set.html. Im Unterschied zum Java-Interface haben wir allerdings die Methodennamen „addAll“, „retainAll“ und „removeAll“ (welche für veränderbare Mengen gedacht sind) durch die Namen der Mengen-Operationen, „merge“, „intersection“, und „difference“ ersetzt, um zu betonen, dass aus zwei bestehenden Mengen eine neue berechnet wird. Der Grund warum die Methode „merge“ nicht „union“ heisst, ist der, dass „union“ in C++ ein Schlüsselwort für eine Union-Datenstruktur ist.

```
bool containsAll(const Set& set) const;
bool isEmpty() const;                    // testet, ob die Menge leer ist
size_t size() const;                     // gibt Anzahl Elemente in der Menge zurueck
                                         // Gleichheitsoperator ('equals' in Java)
                                         // (Inline - Implementation schon gegeben)
bool operator==(const Set& set) const
    { return containsAll(set) && set.containsAll(*this); }

// Ausgabe-Operator für Output-Streams
// (Inline-Implementation schon gegeben)
friend ostream& operator<<(ostream& os, const Set& s) {
    const int* const vptr = s.begin();
    os << "{";
    if (!s.isEmpty()) os << vptr[0];
    for (size_t i = 1; i < s.m_size; ++i) { os << ", " << vptr[i]; }
    os << "}";
    return os;
}


// Klassen-Methoden
// Vereinigungsmenge
static Set merge(const Set& set1, const Set& set2) { return set1.merge(set2); }
// Differenzmenge set1\set2
static Set difference(const Set& set1, const Set& set2)
    { return set2.difference(set1); }
// Schnittmenge
static Set intersection(const Set& set1, const Set& set2)
    { return set1.intersection(set2); }
```

Übernehmen Sie die obenstehenden Methoden-Spezifikationen in Ihre Klassendefinition (MySet.h) und implementieren Sie dann sorgfältig die einzelnen Methoden innerhalb von MySet.cpp.

Der Methode `begin` kommt dabei eine besondere Bedeutung zu. Diese Methode sollte einen Zeiger auf den Beginn des C-Arrays auf dem Heap zurückgeben. Weil wir später beim Erben diese Methode überschreiben werden, ist es wichtig, dass alle anderen Instanz-Methoden immer `begin()` verwenden und nie direkt auf `m_values` zugreifen. Ein Beispiel dafür wäre die Methode `operator[]`, welcher `begin` verwendet, um das i-te Element des Mengen-Arrays zurückzugeben.[7] Die Methode `containsAll(…)` lässt sich am einfachsten durch einen wiederholen Aufruf von `contains(…)` implementieren, während die Methode `operator==`, die bereits implementiert ist, ausnutzt, dass zwei Mengen genau dann gleich sind, wenn sie sämtliche Elemente der jeweils anderen Menge enthalten. Die Methode `operator<<` ist ebenfalls bereits implementiert. Diesen Operator zu überschreiben hat den Vorteil, dass wir Mengen auf der Konsole ausgeben und so gut debuggen können.

Nebst den geschützten Instanzmethoden, gibt es noch (von uns vorgegebene) öffentliche Klassen-Methoden, welche jeweils die entsprechende geschützte Instanzmethode aufrufen. Die Idee hinter dieser Separierung ist die folgende: die angebotenen Mengenoperationen sind binäre Operationen, deren üblichen Operatoren aber nicht als C++-Operatoren zur Verfügung stehen. Daher weichen wir auf Methodennamen wie z.B. intersection aus. Bei einer Schreibweise als Instanzmethode (z.B. set1.intersection(set2)) ist nie ganz klar, ob s1 verändert wird oder nicht. Durch die Schreibweise Set::intersection(set1, set2) kommt deutlicher zum Ausdruck, dass die beiden Mengen gleichberechtigt sind und die Schnittmenge in einem neuen Set abgespeichert wird. Da wir aber später noch Mengen-Operationen (konkret die Vereinigung) in einer abgeleiteten Klasse überschreiben werden, brauchen wir auch noch entsprechende Instanzmethoden, da nur Instanzmethoden vererbt (und überschrieben) werden können. Bitte beachten Sie ferner, dass die Klassen-Methode `difference` die Reihenfolge der Argumente beim internen Aufruf umdreht. Die Instanz-Methode `difference` berechnet also nicht die Differenz `this\set`, sondern `set\this`.[8] Damit Sie eine Idee für die Mengen-Operationen bekommen, geben wir Ihnen die Implementierung von `merge` schon einmal vor:

---

[7] Bitte beachten Sie, dass mathematische Mengen keine Ordnung haben, weswegen wir keinen öffentlichen Indexoperator anbieten möchten. Allerdings speichern wir die Mengenelemente ja intern in einem Array, wo die Elemente in der Reihenfolge geordnet sind, in welcher sie auf dem Heap oder auf dem Stack abgespeichert werden. Den Index-Operator (`operator[]`) intern (protected) zu verwenden, ist also durchaus sinnvoll.

[8] In der Mengenlehre wird für die Differenz nicht das Symbol „-“, sondern das „\“ verwendet)

```
Set Set::merge(const Set& set) const {
    // erstelle eine neue Menge mit allen Elementen von this
    Set result(m_size + set.m_size);
    std::memcpy(result.begin(), begin(), m_size*sizeof(int));
    result.m_size = m_size;

    // fuege nur jene Elemente von set dazu, die in this noch nicht enthalten sind
    for (size_t i = 0; i < set.m_size; ++i) {
       if (!contains(set[i])) result[result.m_size++] = set[i];
    }
    return result;
}
```

## 1.3 Konsolenanwendung

Fügen Sie nun Ihrer Solution eine Konsolenanwendung mit dem Namen „**Console**“ hinzu. Auf SDL-Checks und Precompiled Headers können Sie wieder verzichten. Wenn Sie kein leeres Projekt erstellen, dann wird automatisch auch eine Datei „Console.cpp“ erstellt.

Damit die Konsolenanwendung die statische Bibliothek „MySet.lib“ verwenden kann, fügen Sie der Konsolenanwendung eine Referenz auf das Projekt „MySet“ hinzu: Mit einem Rechtsklick im „Solution Explorer“ auf „Console“ können Sie „Add/Reference…“ auswählen und dann das Projekt „MySet“ auswählen.

Editieren Sie anschliessend die Datei „Console.cpp“ in der nachfolgenden Art. Beachten Sie dabei den relativen Pfad zur Header-Datei „MySet.h“. Führen Sie dann den Build-Prozess aus und starten Sie die Ausführung zuerst ohne und danach mit Debugger. Gehen Sie beim Debuggen mit F11 in die einzelnen Methoden hinein, um zu sehen, wie Ihre Konstruktoren ausgeführt werden. So lange keine Laufzeitfehler auftreten, ist so weit alles OK. Natürlich sagt das noch nichts über die Korrektheit der Implementierung aus. Sobald Sie mehr Methoden der Klasse implementiert haben, sollten Sie dieses Testprogramm um die neuen Methoden bereichern.

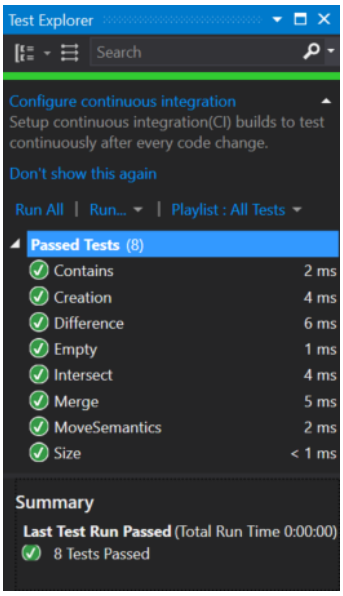
```
#include <iostream>
#include "../MySet/MySet.h"
using namespace std;

int main() {
    Set s1;
    Set s11(s1);

    const int set2[] = { 1,2,3 };
    Set s2(set2, sizeof(set2)/sizeof(int));
    Set s21(s2);
    Set s22 = s2;

    return 0;
}
```

## 1.4 Unit-Tests




In diesem Abschnitt lernen Sie, wie Sie in VS ein natives C++ Unit-Test-Projekt anlegen und ausführen können.

Der Unterordner „**UnitTest**“ soll erst jetzt automatisch durch Erstellung eines neuen Projektes erzeugt werden. Mit einem Rechtsklick im „Solution Explorer“ auf Solution „Uebung2“ können Sie „Add/New Project…“ auswählen. Wählen Sie dann unter „Visual C++/Test“ das „Native Unit Test Project“ aus. Ändern Sie den Projektnamen zu „UnitTest“ und drücken Sie auf OK. Nun wird das Projekt im oben gezeigten Ordner mit der Testdatei „unittest1.cpp“ automatisch angelegt. Bevor Sie die Unit-Tests schreiben, müssen Sie wiederum eine Referenz auf das „MySet“-Projekt hinzufügen: Mit einem Rechtsklick im „Solution Explorer“ auf „UnitTest“ können Sie „Add/Reference…“ auswählen und dann das Projekt „MySet“ auswählen.

Fügen Sie nun vorerst einmal die von uns zur Verfügung gestellten Tests in Ihre Datei ein (alle Test-Funktionen ausser „MoveSemantics“). Wie Sie sehen, gibt es für jede öffentliche Methode der Set-Klasse eine entsprechende Test-Funktion. Sobald Sie die Solution neu kompiliert

haben, können Sie die Tests ausführen lassen.[9] Dazu gehen Sie im Menü auf „Test/Run/All Tests“ oder zum Debuggen der Tests wählen Sie „Test/Debug/All Tests“. Selbstverständlich können Sie auch nur die im „Test Explorer“ selektierten Tests ausführen. Das Fenster „Test Explorer“ sollte beim Ausführen der Tests automatisch geöffnet werden. Falls nicht, so können Sie es über „Test/Windows/Test Explorer“ öffnen. Die rechtsstehende Abbildung zeigt einen Ausschnitt aus dem „Test Explorer“.

## 1.5 Verfeinerung

Sie haben jetzt eine erste Set-Implementierung, die korrekte Resultate liefert. In einem zweiten Schritt werden wir die Set-Klasse noch verfeinern und uns auch ein paar Gedanken zum Thema Effizienz machen.

### 1.5.1 Move Semantik

Die Move-Semantik ist vor allem für Objekte wichtig, bei denen der Kopierkonstruktor eine tiefe Kopie eines grossen Objektes anlegt. Das ist in unserer Set-Klasse nicht der Fall. Dennoch kann die Move-Semantik helfen, überflüssige Kopien von temporären Objekten zu vermeiden. Das wollen wir an folgendem Beispiel ausprobieren:

```
int main() {
    const int set3[] = { 1,2,3 };
    const int set4[] = { 2,3,4 };
    Set s = Set::difference(Set(set3, sizeof(set3)/sizeof(int)),
       Set(set4, sizeof(set4)/sizeof(int)));
    cout << s << endl;
}
```

```
private-ctor
copy values {2, 3, 4}
private-ctor
copy values {1, 2, 3}
Difference
private-ctor
copy-ctor {1}
dtor {1}
dtor {1, 2, 3} delete []
dtor {2, 3, 4} delete []
{1}
dtor {1} delete []
Press any key to continue . . .
```

Überlegen wir uns mal genau, welche Set-Objekte mit welchen Konstruktoren erstellt und wann welche Objekte wieder gelöscht werden. Gehen wir vorerst davon aus, dass für unsere Set-Klasse keine Move-Semantik zur Verfügung steht. `Set(set3, …)` und `Set(set4, …)` rufen offensichtlich jeweils den Typkonvertierungs-Konstruktor (welcher wiederum den nicht-öffentlichen Konstruktor aufruft[10]) auf. Beim Aufruf von `difference` wird zuerst mit dem nicht-öffentlichen Konstruktor ein neues, leeres Set-Objekt erzeugt, welches bereits ein genügend grosses C-Array auf dem Heap alloziert. Dieses wir dann nach und nach mit den entsprechenden Werten (in diesem Fall nur einem, nämlich 1) gefüllt. Das neue Set-Objekt wird by-value zurückgegeben. Das bedeutet konkret, dass auf dem Stack eine mit dem Kopierkonstruktor erzeugte temporäre Kopie des zurückgegebenen Objektes gespeichert wird und die lokalen Set-Objekte der Methode `difference` (`{1}`, `{2,3,4}` und `{2,3,4}`) mit dem Destruktor (dtor) zerstört werden. Ganz zum Schluss, nachdem s auf der Konsole ausgegeben worden ist, wird auch noch s zerstört, weil das Programm zu Ende ist. Schöner wäre natürlich gewesen, wenn das temporäre Set-Objekte (`{1,2,3}` für die resultierende Differenzmenge hätte wiederverwendet werden können statt noch einmal ein neues Set-Objekt zu erstellen. Zum anderen hätten wir in diesem Fall auch das C-Array auf dem Heap wiederverwenden können, denn durch Bilden der Differenz- (oder Vereinigungsmenge) mit einer anderen Menge wird eine Menge ja niemals grösser, sondern höchstens kleiner. Das Array hätte also genügend Platz für alle Elemente der resultierende Menge gehabt.

Implementieren Sie nun zwei zusätzliche Instanzmethoden `difference` und `interesection` mit Move-Semantik (als Parameter wird eine Rechtswert-Referenz erwartet), sodass die folgenden neuen Klassenmethoden unterstützt werden können:

```
// Differenzmenge set1\set2
static Set difference(Set&& set1, const Set& set2)
    { return set2.difference(move(set1)); }
static Set difference(Set&& set1, Set&& set2)
    { return set2.difference(move(set1)); }
```

---

[9] Es kann sein, dass in der x64-Version die Unit-Tests nicht richtig ausgeführt werden. Wechseln Sie in diesem Fall auf die x86-Plattform in VS und führen Sie den Build-Prozess erneut aus.

[10] In der nebenstehenden Konsolenausgabe erscheint „private-ctor“ gefolgt von „copy values“. Dies bedeutet dass der Konvertierungs-Konstruktor den nicht-öffentlichen Konstruktor aufruft, seine Meldung (copy values) aber erst nach dem Aufruf des nicht-öffentlichen Konstruktor auf der Konsole ausgibt.

```
// Schnittmenge
static Set intersection(const Set& set1, Set&& set2)
    { return set1.intersection(move(set2)); }
static Set intersection(Set&& set1, const Set& set2)
    { return set2.intersection(move(set1)); }
static Set intersection(Set&& set1, Set&& set2)
    { return set1.intersection(move(set2)); }
```

**Hinweis**: Da das interne C-Array eines Sets mittels shared-pointer verwaltet wird und der Kopierkonstruktor flach ist (und somit neue Set-Objekte erstellt, die auf dasselbe C-Array auf dem Heap zeigen), dürfen wir ein internes Array nur dann für ein Resultat einer Mengen-Operation wiederverwenden, wenn kein anderes Set-Objekt mehr darauf verweist. Um zu überprüfen, ob ein shared-pointer nicht noch von anderen Objekten verwendet wird, können Sie die Methode `unique` verwenden. Nur wenn diese wahr zurückgibt, dürfen Sie das Array verändern, ansonsten müssen Sie die normale `difference` bzw. `intersection` Methode aufrufen.

Führen Sie dann den zusätzlichen Unit-Test „Move-Semantics“ aus. Falls der Test fehlschlägt, schauen Sie sich den Code der Testfunktion an um zu verstehen, was von den neuen Methoden genau erwartet wird und passen Ihren Code an. Führen Sie anschliessend dasselbe Programm wie oben aus und dokumentieren Sie in der gezeigten Form (Screenshot und Text), was nun genau passiert.[11]

### 1.5.2 Vererbung: Klasse OrderedSet

Einige Mengenoperationen lassen sich effizienter berechnen, wenn die Zahlen im Array geordnet sind. Beispielsweise lässt sich `contains`, welches im Fall eines ungeordneten Arrays lineare Zeit braucht, im geordneten Fall in logarithmischer Zeit berechnen und zwar mit binärer Suche.

Mengen lassen sich auch aus anderen Mengen bilden, indem wir eine Bedingung angeben für die Elemente, die wir aus der anderen Menge übernehmen möchten, wie wir an folgendem Beispiel sehen, wo aus der Menge S zwei neue Mengen $S_1$ und $S_2$ gebildet werden:

$S = \{4,5,6\}, S_1 = \{e | e \in S, e < 6\} = \{4,5\}, S_2 = \{e | e \in S, e > 4\} = \{5,6\}$

Wie wir aus dem Beispiel sehen, liessen sich diese „Behalte-alle-Elemente-kleiner-*x*“ und „Behalte-alle-Elemente-grösser-x“ Methoden zum Erstellen neuer Mengen effizient implementieren, wenn das interne C-Array geordnet ist, da wir in diesem Fall kein neues Array zu erstellen bräuchten, sondern lediglich angeben müssten, an welcher Stelle im bestehenden Array die Menge beginnt und wie gross die neue Menge ist.

Wir implementieren also eine neue Klasse OrderedSet, welche von Set erbt, die Elemente in geordneter Reihenfolge abspeichert und ein zusätzliches Attribut `m_start` beinhaltet, welches aussagt, an welcher Position im Array die Menge *S* beginnt.

Mit einem Rechtsklick im „Solution Explorer“ auf Solution „MySet“ können Sie „Add/New Item…“ auswählen und so zwei neue Dateien „OrderedSet.h“ und „OrderededSet.cpp“ erstellen und dann die beiden Dateien schrittweise erarbeiten. Nebst dem neuen Attribut `m_start` müssen wir für jeden Konstruktor in Set auch einen entsprechenden Konstruktor in OrderedSet haben. Zwar könnten wir mit `using Set::Set;` die Konstruktoren von Set erben, aber diese würden `m_start` nicht initialisieren, weil es dieses Attribut in Set ja nicht gibt. Die Konstruktoren von OrdererSet lassen sich am einfachsten implementieren, wenn man erst einmal den entsprechenden Konstruktor von Set in der Initialisierungsliste aufruft. Der Konvertierungs-Konstruktor muss ausserdem sicherstellen, dass die Elemente im Array auch tatsächlich geordnet sind. Dies erreichen Sie mit dem Einbinden von `<algorithm>` und der folgenden Zeile:

```
int * beg = begin(); std::sort(beg, beg + m_size);
```

Des Weiteren müssen Sie die Methode `begin` überschreiben, damit `m_start` korrekt berücksichtigt wird und somit die Mengen-Operationen, wie Sie sie in Set definiert haben und in OrderedSet erben, weiterhin korrekt funktionieren. Denken Sie daran, beim Überschreiben dieser Methode an den richtigen Stellen im Code `virtual` und `override`[12] hinzuschreiben um sicherzustellen, dass die Polymorphie zum Tragen kommt.

Implementieren Sie schliesslich die beiden nachfolgenden neuen öffentlichen Instanzmethoden zur Erzeugung neuer Mengen, indem Sie für das Resultat keine neuen Arrays allozieren, sondern lediglich den Zeiger, `m_start` und `m_size` entsprechend setzen:

---

[11] Versehen Sie dazu beiden Varianten von `difference` mit einer Konsolenausgabe, die beschreibt, ob die Methode mit oder ohne Verschiebesemantik aufgerufen worden ist.

[12] `override` könnte auch weggelassen werden. Wenn Sie es hinschreiben, dann überprüft der Compiler, ob tatsächlich ein Überschreiben stattfindet.

```
// gibt eine neue Menge zurueck, die alle Elemente von this enthaelt,
// die (strikt) kleiner als x sind
OrderedSet getSmaller(int x) const;
// gibt eine neue Menge zurueck, die alle Elemente von this enthaelt,
// die (strikt) groesser als x sind
OrderedSet getLarger(int x) const;
```

Ergänzen Sie auch die Unit-Tests, um die neue erstellen Konstruktoren und Instanzmethoden zu überprüfen und passen Sie Ihren Code an, sollten diese fehlschlagen. Überprüfen Sie in Ihren Tests insbesondere auch, ob die Resultate immer noch korrekt sind, wenn Sie auf eine Menge, welche Sie mit `getLarger` (oder `getSmaller`) erstellt haben, eine weitere Mengenoperation aufrufen (z. Bsp.) `contains`.

### 1.5.3 Effizientes Berechnen der Vereinigungsmenge

Zum Schluss überlegen wir uns noch, wie wir die bestehenden Mengenoperationen effizienter gestalten könnten und zwar exemplarisch an der Methode `merge`. In dieser Methode können wir die Ordnung ausnützen, indem wir gleichzeitig (mit zwei Zeigern) durch die beiden entsprechenden Arrays durchgehen und die Werte direkt in ein neues OrderedSet kopieren. Die korrekte Merge-Implementierung ist das eine, die richtige Vererbung das andere Problem, das es in dieser letzten Aufgabe zu lösen gilt. Damit wir tatsächlich die `merge`-Methode überschreiben, muss unsere neue Methode genau den gleichen Funktionskopf haben, das heisst das Argument `set` ist eine Referenz zu einem Set- und nicht zu einem OrderedSet-Objekt. Dies bedeutet aber auch, dass beim Aufruf dieser Funktion nicht garantiert ist, dass es sich bei `set` tatsächlich um ein OrderedSet handelt. Konkret heisst das, dass Ihre `merge`-Methode zuerst überprüfen muss, ob es sich beim übergebenen `set` um ein OrdererSet oder nur um „normales“ Set handelt. Dies geht am effizientesten mit einem `dynamic_cast` zu OrderedSet, der ja genau dann fehlschlägt, wenn `set` kein OrderedSet ist. Falls dies tatsächlich der Fall sein sollte, benutzen Sie einfach die `merge`-Methode der Set-Klasse (`Set::merge`), ansonsten Ihre neue Implementierung. Schreiben Sie anschliessend noch einen Unit-Test, welcher die neue Funktion testet und passen Sie Ihren Code gegebenenfalls an.

# 2 Zusammenfassung der Aufgaben

Hier finden Sie nochmals eine Zusammenstellung aller Aufgaben.

1. Set-Klasse vorerst ohne Move-Semantik entwickeln.
2. Set-Klasse mit den vorgegebenen Unit-Tests testen.
3. `difference` und `intersection`-Methoden mit Verschiebesemantik für die Set-Klasse entwickeln, mit dem vorgegebenen Unit-Test testen und dokumentieren, was sich geändert hat.
4. Set-Klasse zu OrderedSet-Klasse mit neuer Funktionalität erweitern und entsprechende Unit-Tests entwickeln und durchführen.
5. Vollständige Set-Klassen, Unit-Tests als Quellcode und Unit-Test-Resultate als Screenshots abgeben.