# 1 Testat-Übung: Set

## 1.1 Einführung

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.

Das entsprechende Projekt ist bereits vorbereitet. Ebenfalls finden Sie die Interface-Definition im `src/set.h` und `src/ordered_set.h` Header. Sie sollen die entsprechend geforderte Funktionalität in `src/set.cpp` und `src/ordered_set.cpp` implementieren. Natürlich dürfen Sie jederzeit weitere Hilfsfunktionen hinzufügen. Die Signatur der definierten *öffentlichen* Methoden dürfen Sie nicht verändern.

Die dazu passenden Unit-Tests sind bereits implementiert und erlauben ihnen, ihre Implementation zu testen. Zudem existiert auch eine kleine Applikation, welche Sie zum Experimentieren verwenden können. Oder Sie schreiben zusätzliche Unit-Tests. Das verwendete Test-Framework Catch2 haben Sie bereits kennengelernt.

Öffnen Sie das `workspace`-Repo in CLion als Root-Projekt und navigieren Sie zu `uebungen/set`. Hier finden sie das vorbereitete Projekt. Aktivieren Sie es indem Sie in `uebungen/CMakeLists.txt` die folgende Zeile aktivieren:

```
1 add_subdirectory(set)
```

Die Aufgaben sind Schritt für Schritt abzuarbeiten. Bis auf den mit *optional* markierten Teil sind alle zwingend korrekt zu implementieren. Aktivieren Sie die einzelnen Tests/Signaturen, indem Sie die vorbereiteten `#defines` in `set.h` und `ordered_set.h` aktivieren.

Genauere Infos zu Mengen und Mengen-Operationen finden Sie im Internet z.B: Wikipedia: Menge (Mathematik)

## 1.2 Testat

Geben Sie die geforderten Implementationen bis zum vereinbarten Datum ab. Senden Sie die Dateien `set.h`, `set.cpp`, `ordered_set.h` und `ordered_set.cpp` per Mail an christian.lang@fhnw.ch. Die Lösung muss einerseits alle Unit-Tests, anderseits auch bestimmte Qualitäts-Merkmale erfüllen. Diese Qualitätsmerkmale werden einerseits automatisch durch cpplint andererseits durch den Dozenten geprüft. Sie erhalten dementsprechend Feedback zu ihrer Abgabe.

### 1.3 Ziele

- Entwickeln einer eigenen `Set`-Klasse für mathematische Mengen von Ganzzahlen ähnlich wie die Set-Klassen in Java.
- Erkennen des Potential der Move-Semantik und von `std::shared_ptr` und die Auswirkungen auf die Performance.
- Verstehen und üben des Prinzip der Vererbung.

## 2 Einführung: Mengen

Eine mathematische Menge ist definiert durch die Elemente, welche sie enthält. 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 Sie eine Klasse, welche eine endliche Menge von ganzen Zahlen (`int`) repräsentiert.

Die Mathematik definiert einige Operationen, um aus bestehenden Mengen neue Mengen 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 `std::set` lediglich ein Container ist, bei dem jedes Element höchstens einmal vorkommen darf. Während in Java und C++ mit diesen Containern *veränderbare* Mengen repräsentiert werden, implementieren Sie nun eine Klasse für *unveränderbare* Mengen.

## 3 Basis-Funktionen der Set-Klasse

In diesem ersten Teil sollen Sie die Grundfunktionen der Klasse implementieren. Achten Sie dabei auf die Inline-Dokumentation im Header.

### 3.1 Interne Datenstruktur

Unsere `Set`-Datenstruktur braucht offensichtlich ein Attribut, welches die Elemente der Menge speichert. Das heisst, sie soll ein nicht veränderbares `int`-Array (`values_`) verwalten. 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 Smart-Pointer 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 `size_` vom Typ `size_t` vorsehen.

C-Arrays können in C/C++ ganz einfach durch einen Raw-Pointer vom Typ `int*` repräsentiert werden. Ein solcher Pointer 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 Pointer auf das bereits bestehende Array auf dem Heap zeigen lassen. Hier bietet sich also für `values_` ein Smart-Pointer an: `shared_ptr<int[]>`. Dieser smarte Pointer registriert, wenn der Pointer kopiert wird. Dadurch wird sichergestellt, dass das Array erst dann vom Heap gelöscht wird, wenn kein `Set`-Objekt mehr darauf verweist.

## 3.2 Konstruktoren und Destruktor

Als erstes programmieren wir einen Konstruktor mit einem einzelnen Parameter (der maximalen Grösse), 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 `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 Allozierung des Speichers. Der korrekte Weg um passende Instanzen für Smart-Pointer (`std::shared_ptr`, `std::unique_ptr` oder `std::weak_ptr`) zu instanziieren ist die Verwendung von `std::make_shared` oder `std::make_unique`. Diese optimieren einerseits die Speicher-Allozierung, andererseits sorgen sie für korrektes Aufräumen, falls der Konstruktor der erzeugten Instanz eine Exception werfen sollte.

Leider gibt es im C++17-Standard noch keine Variante für Arrays in Kombination mit `std::make_shared`. Allerdings in Kombination mit `std::make_unique`. Da `std::unique_ptr` mittels Move-Semantik auch zu `std::shared_ptr` zugewiesen werden können, verwenden wir momentan diesen. Die `shared`-Variante wird im C++20-Standard nachgeliefert.

```
1 Set::Set(size_t size) : values_(std::make_unique<int[]>(size)), ... {
```

Dies soll der einzige Ort im gesamten Code sein, wo dynamisch Memory alloziert wird.

Zusätzlich zu diesem ersten, wollen wir noch weitere 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 erstellte Kopier-

konstruktor verwendet werden. Damit Sie besser verstehen, wann dieser Konstruktor aufgerufen wird und hinein debuggen können, empfehlen wir Ihnen, diesen selbst zu programmieren und mit einer Konsolenausgabe zu versehen.

- Einen Typkonvertierungs-Konstruktor soll eine Initialisierungsliste des Typs `initializer_list<int>` entgegennehmen und die darin enthaltenen Werte in das C-Array abfüllen. Achten Sie darauf, dass eine Menge jedes Element nur einmal enthalten kann. Deswegen sollten Sie nicht einfach alle sondern nur Elemente einfügen, welche nicht bereits eingefügt wurden. Eine Möglichkeit dies zu erreichen, ist als erstes den Konstruktor (welcher die Speicherallozierung durchführt) aufzurufen, und danach die Zahlen einzeln von der Initialisierungsliste nach `values_` zu kopieren. Aber nur, wenn eine Zahl noch nicht im neuen Array enthalten ist. Hierfür sollten Sie bereits die weiter unten erwähnten Hilfs-Methoden `AddUnchecked()` und `Contains()` verwenden.

Bleibt schliesslich noch der Destruktor. Was gibt es hier zu tun? Was wäre anders, wenn wir einen rohen Pointer anstatt dem `std::shared_ptr<int[]>` für das Array in unserer Datenstruktur verwendet hätten? Auch hier wird empfohlen, diesen Destruktor aus didaktischen Gründen mit Konsolen-Ausgabe zu implementieren, damit Sie dessen Aufrufe beobachten können.

### 3.3 Hilfs-Methoden und -Operatoren

Es sind mehrere Hilfs-Methoden deklariert (und teilweise bereits inline implementiert), welche Ihnen für die weiteren Implementierungen nutzen werden.

Die erste ist der Stream-Operator `operator<<`. Dieser sorgt dafür, dass eine Instanz dieser Klasse ganz einfach an einen Output-Stream, wie z.B `std::cout`, übergeben und somit auf die Konsole ausgegeben werden kann.

Des weiteren ist auch bereits `begin()` und `end()` implementiert. Diese Methoden kommt eine besondere Bedeutung zu, denn sie definieren den Start und das Ende des C-Arrays auf dem Heap. Weil wir später beim Erben diese Methoden überschreiben werden, ist es wichtig, dass alle anderen Instanz-Methoden intern immer `begin()` und `end()` verwenden und nie direkt auf `values_` zugreifen. Ein Beispiel dafür wäre der Index-Operator `operator[]`, welcher `AtUnchecked()` verwendet, welcher seinerseits selber `begin()` und `end()` verwenden soll. `AtUnchecked()` muss dabei in zwei Formen implementiert werden. Einerseits als veränderbar und andererseits als **`const`**-Variante.

Um einfach Elemente hinzuzufügen oder vom `Set` zu entfernen sollen, die Helper `AddUnchecked()` und `RemoveUnchecked()` implementiert werden. Das «Unchecked» meint hier, das beide Methoden keine internen Checks brauchen, welche überprüfen würden, ob überhaupt noch genügend Speicher für ein Hinzufügen vorhanden ist oder ob das entsprechende Element zum Entfernen überhaupt im

`Set` existiert. Diese Checks werden von der aufrufenden Methode übernommen. Die Helper selber sollen vorallem das Management von `size_` übernehmen.

### 3.4 Zugriffs-Methoden und -Operatoren

Neben den selbsterklärenden Methoden wie `size()` und `empty()`, sollen auch der Index-Operator `operator[]` implementiert werden, welcher das i-te Element des Mengen-Arrays zurückgibt. Die Methode `ContainsAll()` lässt sich am einfachsten durch einen wiederholten Aufruf von `Contains()` implementieren, während der Vergleichs-Operator `operator==` ausnutzt, dass zwei Mengen genau dann gleich sind, wenn sie sämtliche Elemente der jeweils anderen Menge enthalten.

### 3.5 Klassen-Methoden

Die `static` deklarierten Klassen-Methoden wie z.B: `static Set Merge(const Set& set1, const Set& set2)` kapseln die Member-Methoden, um den Aufruf von zwei `Set`s einfacher zu gestalten. Diese Methoden brauchen Sie nicht anzupassen.

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 `set1` 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 trotzdem entsprechende Instanzmethoden, da nur Instanzmethoden vererbt (und überschrieben) werden können.

## 4 Merge

Die erste Operation welche Sie implementieren sollen, ist die Vereinigungs-Menge. Aktivieren Sie dazu das `#define OPERATION_MERGE` in `set.h`.

Bei der Vereinigungs-Operation werden alle Elemente beider Input-Mengen in eine neue Menge eingefügt. Da die Definition der Menge aber beinhaltet, das kein Element doppelt vorkommen darf, müssen entsprechende Duplikate verhindert werden. Damit Sie ein Gefühl für die Operation und die Implementierung bekommen, hier ein Stück Pseudo-Code, als Hinweis:

```
// Create result set that can hold all elements of both sets.

// Copy all elements of this set with usage of AddUnchecked() and AtUnchecked()

// Conditional copy of the elements into the result set.
// Use Contains() to check if the element is already in the result set.
// If not add it to the result with AddUnchecked().

// return result set
```

## 5 Intersection

Die Schnittmenge ist definiert durch alle Elemente, welche in beiden Input-Mengen vorkommen. Entsprechend können Sie auch hier wieder die oben genannten Helper verwenden.

## 6 Difference

Die Differenz-Menge ist definiert als die Menge, welche übrig bleibt, wenn von der einen Menge alle Elemente der anderen Menge entfernt werden.

Bitte beachten Sie, dass die Klassen-Methode `Set::Difference()` die Reihenfolge der Argumente beim internen Aufruf umdreht. Die Instanz-Methode `Difference()` berechnet also nicht die Differenz $this - other$, sondern $other - this$.

## 7 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() {
  Set s = Set::Difference(Set({1, 2, 3}), Set({2, 3, 4}));
  std::cout << s << std::endl;
}
```

Überlegen Sie sich, welche `Set`-Objekte mit welchen Konstruktoren erstellt und wann welche Objekte wieder gelöscht werden. Gehen Sie vorerst davon aus, dass für unsere `Set`-Klasse keine Move-Semantik zur Verfügung steht. `Set({1,2,3})` und `Set({2,3,4})` rufen offensichtlich jeweils den

`initializer_list`-Konstruktor auf (welcher wiederum den `size_t`-Konstruktor aufruft). Beim Aufruf von `Difference` wird zuerst mit dem `size_t`-Konstruktor ein neues, leeres `Set`-Objekt erzeugt, welches bereits ein genügend grosses C-Array auf dem Heap alloziert. Dieses wird 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` (`{2,3,4}` und `{2,3,4}`) mit dem Destruktor zerstört werden. Ganz zum Schluss, nachdem `s` auf der Konsole ausgegeben worden ist, wird auch noch `s` zerstört, weil das Programm zu Ende ist. Folgendes wird in diesem Fall auf die Konsole ausgegeben:

```
Set: size-ctor: allocate 3 elements
Set: initializer_list-ctor {2, 3, 4}
Set: size-ctor: allocate 3 elements
Set: initializer_list-ctor {1, 2, 3}
Set: copy-difference: {1, 2, 3}
Set: size-ctor: allocate 3 elements
Set: dtor {1, 2, 3}
Set: dtor {2, 3, 4}
{1}
Set: dtor {1}
```

Dies können Sie erzeugen, wenn Sie in allen relevanten Methoden entsprechende Konsolen-Logs einbauen, etwa so:

```
std::cout << "Set: initializer_list-ctor " << *this << std::endl;
```

Schöner wäre natürlich, wenn das temporäre `Set`-Objekt `{1,2,3}` für die resultierende Differenzmenge hätte wiederverwendet werden können, anstatt 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 resultierenden Menge gehabt. In diesem Fall erhalten Sie etwa folgenden Output:

```
Set: size-ctor: allocate 3 elements
Set: initializer_list-ctor {2, 3, 4}
Set: size-ctor: allocate 3 elements
Set: initializer_list-ctor {1, 2, 3}
Set: move-difference: {1, 2, 3}
Set: move-ctor {1}
Set: dtor {1}
Set: dtor {2, 3, 4}
{1}
Set: dtor {1}
```

Implementieren Sie nun die zwei Instanzmethoden `Difference()` und `Interesection()`

mit Move-Semantik, indem Sie `#define OPERATION_INTERSECTION_MOVE` und `#define OPERATION_DIFFERENCE_MOVE` im Header aktivieren.

Hinweis: Da das interne C-Array eines `Set`s mittels `std::shared_ptr` 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 `std::shared_ptr` nicht noch von anderen Objekten verwendet wird, können Sie die Methode `use_count()` verwenden. Nur wenn diese `1` zurückgibt, dürfen Sie das Array verändern, ansonsten müssen Sie die normale `Difference()` bzw. `Interesection()` Methode aufrufen.

# 8 OrderedSet

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

Mengen lassen sich auch aus anderen Mengen bilden, indem wir eine Bedingung für die Elemente angeben, die wir aus der anderen Menge übernehmen möchten. In folgendem Beispiel sehen wir, wie 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 effizienter implementieren, wenn das interne C-Array sortiert wäre. Dies, 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.

Implementieren Sie also eine neue Klasse `OrderedSet`, welche von `Set` erbt, die Elemente in sortierter Reihenfolge abspeichert und ein zusätzliches Attribut `start_` beinhaltet, welches aussagt, an welcher Position im Array die Menge $S$ beginnt. Die entsprechenden Dateien sind bereits vorbereitet. Aktivieren Sie den ersten Teil mittels `#define ORDERED_SET_BASE` und fahren Sie danach mit `ORDERED_SET_GET_SMALLER` und `ORDERED_SET_GET_LARGER` fort.

Nebst dem neuen Attribut `start_` müssen Sie für jeden Konstruktor in `Set` auch einen entsprechenden Konstruktor in `OrderedSet` haben. 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 sortiert sind. Dies erreichen Sie mit Implementieren des Helpers: `SortIntern()`. In diesem können Sie `std::sort` verwenden, welches Sie im Header `<algorithm>` finden.

Nun kommt das Überschreiben der Methode `begin()` zum Zug. Dies ist bereits inline im Header erledigt. Allerdings auch hier wieder die Warnung, dass Sie nur über diese Methode auf den internen Speicher zugreifen sollen, damit auch alle geerbten Methoden von `Set` mit einer `OrderedSet`-Instanz funktionieren.

Implementieren Sie schliesslich die beiden neuen Methoden `GetSmaller()` und `GetLarger()` zur Erzeugung neuer Mengen, indem Sie für das Resultat keine neuen Arrays allozieren, sondern lediglich den `start_` und `size_` entsprechend setzen.

## 9 Optimiertes Merge *(optional)*

Zum Schluss überlegen wir uns noch, wie wir die bestehenden Mengenoperationen, exemplarisch an der Methode `Merge()`, effizienter gestalten könnten. In dieser Methode können wir die Ordnung ausnützen, indem wir gleichzeitig (mit zwei Pointern) durch die beiden entsprechenden Arrays durchgehen und die Werte direkt in ein neues `OrderedSet` kopieren. Die korrekte Implementierung ist das eine, die richtige Vererbung das andere Problem, das es in dieser Aufgabe zu lösen gilt. Damit Sie tatsächlich die `Merge()`-Methode überschreiben, muss die neue Methode die genau gleiche Signatur haben, das heisst das Argument `other` 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 `other` tatsächlich um ein `OrderedSet` handelt. Konkret heisst das, dass Ihre `Merge()`-Methode zuerst überprüfen muss, ob es sich beim übergebenen `other` um ein `OrdererSet` oder nur um „normales“ `Set` handelt. Dies geht am effizientesten mit einem

```
1  dynamic_cast<const OrderedSet*>(&other)
```

der ja genau dann fehlschlägt, wenn `other` 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.