08_stack: Entwicklung eines einfachen Stacks

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

Es geht in diesem Beispiel darum einen einfachen Stack zu entwickeln und jede Menge neuer Konzepte und Software in dieses Projekt einzubauen.

Ein Executable ist in diesem Beispiel nicht von Belang, da hier lediglich ein Stack als Bibliothek implementiert wird und dieser Stack mit Hilfe von Unit-Tests getestet werden wird. D.h. in der `meson.build` wird kein `executable()` benötigt.

3 Anleitung

1. In diesem Beispiel werden wir die Bibliothek `spdlog` verwenden, die die Möglichkeit zur Verfügung stellt, Logginginformationen auf die Konsole, in eine Datei oder über das Netzwerk zu schreiben.

   Diese Bibliothek steht wieder als header-only Version oder als ”compiled”-Bibliothek zur Verfügung. Wir werden wieder eine ”compiled”-Version zum Einsatz bringen, wegen der Übersetzungszeiten (auch wenn dies in unseren Projekten nicht sehr von Belang ist).

   Clone dazu spdlog von github an eine nette Stelle in deinem Dateisystem. Jetzt geht es wieder an das Übersetzen von `spdlog`:

   a) Wechsle in das Verzeichnis von `spdlog`

   b) lege dort ein Verzeichnis `build` an

   c) wiederum hineinwechseln, dann nochmals ein beherztes `cmake ..`

   d) und nicht auf das finale `make` vergessen.

   Voilà!

   Letzter Schritt: `meson.build` und `meson_options.txt` anpassen, um die Bibliothek einzubinden.

   Teste dies in deinem Projekt gleich mit einer Testausgabe!

BTW: Wenn du `spdlog` einbindest, denke daran, dass `spdlog` auch `fmt` verwendet und deshalb die entsprechenden `#pragma`-Direktiven davor stehen müssen.

2. Jetzt kannst du die Testausgabe wieder löschen. Aber vorher hast du committed, nicht wahr?

3. Als nächstes wollen wir ein Modul `stack` (`.h` und `.cpp`) anlegen, das einen üblichen Stack implementieren soll. Die soll jedoch als eigene *statische* Bibliothek in einem eigenem Verzeichnis `stack` passieren, das in einem eigenen Unterverzeichnis unseres Projektverzeichnisses angelegt werden soll.

   Der Stack selber ist *noch nicht* zu implementieren (siehe nächster Punkt).In diesem Schritt geht es nur darum, die Struktur und die `meson.build` Dateien anzulegen,

   Dieser Teil des Projektes soll in der `meson.build` des Projektes einfach mittels `subdir()` eingebunden werden.

   Lege daher in deinem Projektverzeichnis ein neues Verzeichnis `stack` an, das wiederum eine Datei `meson.build` und die Verzeichnisse `src`, `include` und `tests` enthalten soll (*kein* `build`!). Was in `meson.build` in solch einem Fall enthalten sein soll und was auf keinem Fall enthalten sein darf, steht in `meson_tutorial` (wo sonst?).

4. Jetzt geht es darum das Modul `stack` zu implementieren, das eigentlich nur einen Stack beinhalten soll. Implementiere die folgende Version in einem Namespace `stacks`:

   ```
   struct Node {
       int value{};
       Node* next{};
   };

   class Stack final {
     public:
       Stack();
       ~Stack();
       int& top();
       int pop();
       void push(int);
       void clear();
       bool empty();
     private:
       Node* top_{};
   };
   ```

   Die Semantik der Methoden als auch deren Implementierung sollten soweit klar sein. Es ist offensichtlich wieder eine Implementierung auf Basis von rohen Zeigern gefragt (nicht, dass dies eine gute Lösung ist...).

   Der Konstruktor ist eigentlich nicht notwendig (der compilergenerierte Defaultkonstruktor wäre ausreichend, siehe member initializer von `top_`), aber wir werden diesen implemetieren, um später Logginginformationen ausgeben zu können. Der Rest ist sowieso klar und trivial.

   Was soll bei `top()` bzw. `pop()` passieren, wenn der Stack leer ist? Prinzipiell würde es ja die Möglichkeit geben, einen Fehlerwert oder einen ungültigen Wert zurückzuliefern, aber beides ist bei der Rückgabe eines lvalue oder einer lvalue reference eines fundamentalen Typs ausgeschlossen. Prinzipiell wäre das Setzen eines Fehlerzustandes (z.B. in einer globalen Variable oder in der Instanz) möglich, aber auch das ist nicht sonderlich elegant...

Also werfen wir eine Exception. Aber welche? Schaue dir die Hierarchie der Exceptions, z.B. auf cppreference, an und werfe voller Inbrunst einen `domain_error`!

5. D.h. schreibe jetzt Unit-Tests für die bisherige Funktionalität. Tipps:

   - Konsultiere den Foliensatz `data_types`! D.h. jetzt werden wir unseren Ansatz zum Schreiben der Unit-Tests formalisiert betrachten!!!

     Aber Achtung: Beim Pushen eines Wertes gibt es eigentlich 2 Fälle: Pushen auf leeren Stack und Pushen auf nicht leerem Stack!

     Weiters beachte bitte, dass unser Interface ein bisschen anders aussieht als bei axiomatischen Beschreibung des abstrakten Datentyp Stack aus den Folien.

   - `doctest` hat auch die folgenden Macros `CHECK_THROWS` und `CHECK_THROWS_AS`!

6. Überlege dir was bei folgendem Codefragment passiert:

   ```
   Stack s1;
   s1.push(1);
   Stack s2{s1};
   ```

   Was also ist das Problem?

   Ok, wenn es nicht klar sein sollte und auch wenn es klar sein sollte, dann füge sowohl im Konstruktor als auch im Destruktor eine Logging-Anweisung hinzu und weiters füge auch Anweisungen der folgenden Art in `main` hinter den obig angeführten Anweisungen hinzu:

   ```
   spdlog::info(fmt::format("s1.top_ == {}", static_cast<void*>(s1.top_)));
   spdlog::info(fmt::format("s2.top_ == {}", static_cast<void*>(s2.top_)));
   ```

   Damit dies funktioniert musst du *kurzfristig* `private:` in der Klasse `Stack` auskommentieren. Beachte, dass die "Implementierung" von spdlog es will, dass man einen Pointer zu `void*` casten muss, damit der Wert ausgegeben wird...

   Bei mir kommt es danach beispielsweise zu folgender Ausgabe:

   ```
   [2019-07-12 13:57:36.848] [info] constructor
   [2019-07-12 13:57:36.848] [info] s1.top_ == 0x565474f69200
   [2019-07-12 13:57:36.848] [info] s2.top_ == 0x565474f69200
   [2019-07-12 13:57:36.848] [info] destructor
   [2019-07-12 13:57:36.848] [info] destructor
   ```

   Ok, jetzt sollte es aber klar sein. Allerdings ist es trotzdem keine sinnvolle Option sein Projekt mittels logging zu testen!

7. Was haben wir bis jetzt gesehen bzw. wie sieht es mit den vom Compiler generierten Konstruktoren aus:

   - Der Compiler generiert automatisch einen Copy-Konstruktor, wenn wir keinen definieren
     - außer wir definieren einen Move-Konstruktor oder Copy-Assignment-Operator, dann ist dieser sogar "deleted" (aber das ist eine andere Geschichte).
   - Wenn wir keinen Konstruktor definieren, dann generiert der Compiler automatisch einen Default-Konstruktor.
   - Wenn wir irgendeinen Konstruktor definieren, dann wird kein Copy-Konstruktor vom Compiler generiert.
   - Ein Default-Destruktor wird vom Compiler immer generiert, außer wir definieren selber einen Destruktor.

8. So, jetzt ist es soweit, du musst einen Copy-Konstruktor implementieren!

   - Tipp: Fange einfach einmal simpel an und gib einfach ”copy constructor” auf stdout aus.

     Damit du das übersetzen kannst, musst du auch etwas mit dem übergebenen Argument machen, sonst übersetzt es nicht. Du erinnerst dich, dass wir unsere Projekte so konfigurieren, dass alle Compilerwarnungen als Compilerfehler gewertet werden. Das war bis jetzt kein Problem, aber der Parameter ist `const`, damit geht ja auch ein `other = other;` nicht mehr. Ok, man kann auch noch auf solche wahnwitzigen Sachen verfallen, um den Compiler zufrieden zu stellen:

     ```
     Stack* tmp{const_cast<Stack*>(&other)};
     tmp = tmp;
     ```

     Aber ehrlich... Das kann ja nicht wahr sein!

     Ok, es gibt in C++ Attribute. Eines haben wir schon kennengelernt: `[[fallthrough]]`, jetzt kommt `[[maybe_unused]]`. Dieses kann einfach vor den Parameter platziert werden, der nicht verwendet wird und alles ist gut. Probiere es aus! Ist doch viel besser, nicht wahr?

   - Jetzt das Programm starten?! Bei mir kommt es jetzt zu folgender *erwarteter* Ausgabe:

     ```
     [2019-07-12 14:26:25.328] [info] constructor
     [2019-07-12 14:26:25.328] [info] copy constructor
     [2019-07-12 14:26:25.328] [info] s1.top_ == 0x5625baa74200
     [2019-07-12 14:26:25.328] [info] s2.top_ == 0x0
     [2019-07-12 14:26:25.328] [info] destructor
     [2019-07-12 14:26:25.328] [info] destructor
     ```

     Jetzt ist es an der Zeit einen richtigen Copy-Constructor zu implementieren. Go!

   - Wenn dieser funktioniert, ist es wieder an der Zeit, die unnötigen Logging-Anweisungen zu löschen (und auch `private:` wieder zu aktivieren). Belasse aber den Default-Konstruktor.

9. Implementiere als nächstes einen hübschen überladenen Operator `<<`, man weiß ja nie genau wann man diesen benötigt.

10. Copy-Konstruktor ist die eine Sache, aber wie sieht es mit dem folgendem Konstrukt aus?

    ```
    Stack s1;
    Stack s2;
    s1 = s2;
    ```

    Es fehlt noch ganz Entscheidendes in unserem Programm, damit auch so etwas funktioniert!

    Denken!

    Ok, es ist eh klar, nicht wahr? Wenn nicht, dann teste einmal mit folgendem Codesnippet:

    ```
    Stack s1;
    s1.push(3);
    s1.push(2);
    Stack s2;
    s1 = s2;
    cout << s1 << endl;
    cout << s2 << endl;
    s1.push(1);
    cout << s1 << endl;
    ```

    Denken, erst dann zum nächsten Punkt!

11. Genau, der (copy) assignment operator fehlt noch!

    Was ist hier zu beachten?

    a) Eine Zuweisung an sich selber sollte geprüft werden (Performance und u.U. auch Fehlerquelle)

    b) Der alte Speicher ist zu freizugeben

    c) Der neue benötigte Speicher ist anzufordern (entsprechend der Größe des anderen Objektes)

    d) Vom anderen Objekt in den neuen Speicher kopieren

    Das ist der prinzipielle Ablauf, den wir jedoch in unserem konkreten Fall *nicht* implementieren werden. Daher zuerst diesen Ansatz verstehen und dann weiter zum nächsten Punkt.

12. Der Ansatz ist prinzipiell gut, hat aber ein zwei prinzipielle Nachteile:

    - Es ist das Kopieren zwei Mal zu implementieren: Einmal im Copy-Konstruktor und einmal im Copy-Assignment-Operator. Das ist doppelte Arbeit und eine zusätzliche Fehlerquelle.
    - Es gibt eine kleine Schwachstelle: Was ist, wenn eine Exception geworfen wird, wenn der neue Speicher angefordert wird (z.B. wenn kein Speicher mehr vorhanden) oder ein Konstruktor von einem Objekt, das in dem neu angeforderten Speicher liegt, eine Exception wirft? Verstanden?!

      Dann ist das neue Objekt nicht vorhanden und das alte Objekt (Speicher schon freigegeben) auch schon weg!

    Deshalb werden wir einen anderen Ansatz wählen, nämlich wir auf das *copy-and-swap* idiom zurückgreifen, das den schon implementierten Copy-Konstruktor verwendet:

    ```
    X& X::operator=(const X& rhs) {
        if (this == &rhs)
            return *this;
        X tmp{rhs};
        swap(*this, tmp);
        return *this;
    }
    ```

    Und man benötigt noch eine geeignete Funktion `swap`, die natürlich ein "Freund" unserer Klasse sein muss:

    ```
    void swap(X& first, X& second) noexcept {
        using std::swap;

        swap(first.a, second.a);
        swap(first.b, second.b);
    }
    ```

    Wir gehen hier davon aus, dass `X` zwei Instanzvariablen `a` und `b` besitzt, die natürlich in der Regel auch Zeiger sein können.

    Was ist hier zu beachten?

    - Beachte im speziellen `noexcept` und lese nach was dies bedeutet! Ein vom Compiler generierte Destruktor ist automatisch `noexcept`! Hmm, jetzt haben wir selber einen Destruktor definiert, der im Moment auch nicht `noexcept` ist. Können wir unseren Destruktor `noexcept` deklarieren?

Ja, warum nicht, denn wir rufen nur `clear()` auf und wenn wir `clear()` richtig implementiert haben, dann wird diese auch keine Exception werfen und kann daher auch den `noexcept` Spezifizierer erhalten. Go!

Wenn wir gerade so richtig in Fahrt gekommen sind: Gibt es noch andere Methoden, die als `noexcept` spezifiziert werden können?

- Weiters siehst du eine nette Verwendung einer `using`-Deklaration.

Nachteile? Die Nachteile dieses Ansatzes sind:

- Es ist extra eine Funktion `swap` ist zu implementieren.
- Die Performance ist etwas geringer als bei der ersten Variante.

Unit-Tests nicht vergessen!

13. Wann generiert der Compiler einen Copy-Assignment-Operator? Eigentlich eh immer, außer wir definieren selber einen.

    Was heißt hier "eh immer"? Tja, wenn wir einen Move-Konstruktor oder einen Move-Assignment-Operator definieren, dann wird keiner definiert und diese sind sogar "deleted", aber wie wir schon wissen: Das ist eine andere Geschichte!

14. Abschließend noch die Faustregel "rule of three": Wenn eine der folgenden Elementfunktionen (member function)

    - Destruktor
    - Copy-Konstruktor
    - Copy-Assignment-Operator

    implementiert ist, dann sind alle zu implementieren!

    Was wir hiermit auch getan haben.

15. Schauen wir uns jetzt einmal die Konstruktoren etwas genauer an. Was ist, wenn wir den Stack schon initialisiert anlegen wollen?

    Wir wollen einen Stack mit genau einem Element anlegen. Implementiere daher einen entsprechenden Konstruktor und teste diesen ebenfalls.

16. Fein, jetzt erkennen wir allerdings, dass unser Default-Konstruktor eigentlich ziemlich unnötig ist (die Logging-Anweisung haben wir ja schon entfernt). Also, weg mit der Implementierung des Default-Konstruktors. Go!

    Und wie geht es? Übersetzungsfehler? Warum? Denken!

17. Ok, der Default-Konstruktor wird vom Compiler nicht mehr generiert, da...

    So, füge folgendes zu deiner Klassendefinition hinzu:

    ```
    Stack()=default;
    ```

    Nicht schlecht, oder?

18. Wir wollen aber auch einen Stack mit einer beliebigen Anzahl an Elementen initialisieren, wie wir das von einem `std::vector` gewohnt sind. Dazu benötigen wir einen Konstruktor, der einen Parameter vom Typ `std::initializer_list` bekommt. Schaue dir das entsprechende Beispiel in der cppreference an und implementiere solch einen Konstruktor!

    Ach ja, die Unit-Tests dürfen nicht vergessen werden.

19. Wenn du alles richtig gemacht hast und überall die ”uniform initialization” verwendet hast, dann hast du jetzt in deinen Tests vermutlich einen Fehler, den du aber nicht bemerkst!

    Ändere die Implementierung von `Stack(int)` so ab, dass diese einen falschen Wert auf den Stack gibt und starte deine Tests erneut. Was passiert?

    Bitte wirklich erst weiterlesen, wenn du das ausprobiert hast!!!

20. Nichts? Genau, aber warum? Weil der Konstruktor mit dem Parameter vom Typ `initializer_list` auch verwendet wird, wenn `Stack s{1}` geschrieben wird. Ist halt so und kennst du auch: Denke an `vector{1}` vs. `vector(1)`! Wenn du jetzt nicht weißt was ich damit meine, dann konsultiere die cppreference zum Theme `vector` und dessen Konstruktoren!

    Und fixe deinen Test (und stelle die Implementierung von `Stack(int)` wieder richtig!

21. Hmm, was ist aber, wenn wir folgenden Code schreiben:

    ```
    Stack tmp={Stack{1, 2, 3}};
    ```

    - Funktioniert dies? Ja, warum nicht?
    - Was passiert? Es wird ein temporäres Objekt vom Typ `Stack` angelegt, von diesem eine Kopie erstellt und danach das temporäre Objekt wieder gelöscht. Damit einhergehend wird natürlich Speicher angelegt und auch wieder freigegeben (bis `C++14`). Ziemlich sinnlos!

      Überprüfen wir das, indem wir wieder `spdlog::info("copy ctor")` in den Copy-Konstruktor einfügen!

      Und? Nichts zu sehen?! Warum? Weil der Compiler den Aufruf des Copy-Konstruktors wegoptimiert (wird *copy elision* genannt, ab `C++17` Pflicht und so definiert!). Optimierungen in dieser Art nimmt der Compiler viele vor! Dies hängt vom Compiler und den Einstellungen (im speziellen, ob als Debug oder Release-Version übersetzt wird) ab.

      Besser wäre es natürlich von Haus aus gewesen die Definition der Variable `tmp` mittels *direct initialization* anzuschreiben (anstatt mittels *copy initialization*):

      ```
      Stack tmp{Stack{1, 2, 3}};
      ```

      Aber selbst das ist an sich nicht sonderlich schlau, denn eigentlich wäre die folgende Initialisierung die richtige gewesen:

      ```
      Stack tmp{1, 2, 3};
      ```

      So werden wir das klarerweise auch in Zukunft verwenden, auch wenn der Compiler die unnötigen Aufrufe der Copy-Konstruktoren wegoptimiert.

      Aber wie sieht es mit folgendem Code aus:

      ```
      Stack test() {
          spdlog::info("inside test");
          Stack r;
          return r;
      }
      ```

      Teste indem du `test()` aufrufst!

      Und? Wieder nichts zu sehen?! Wieder eine pflichtmäßige Optimierung (wird speziell NRVO – Named Return Value Optimization genannt). Das funktioniert klarerweise auch, wenn man anstatt `return r;` einfach ein temporäres Objekt zurückliefert (z.B. mittels `return Stack{};`), dann wird diese Optimierung einfach RVO genannt. Aber schaue dir einmal folgenden Code an (abtippen und ausprobieren):

```
Stack test([[maybe_unused]]bool mark) {
    spdlog::info("inside test");
    if (mark) {
        Stack r1;
        return r1;
    } else {
        Stack r2;
        return r2;
    }
}
```

Und wieder aufrufen!

Und? Hier wirst du sehen, dass der Copy-Konstruktor aufgerufen wird, da der Compiler hier nicht in vorhinein wissen kann, welcher Zweig durchlaufen wird und daher auch nicht dementsprechend Code generieren kann, sodass das richtige Objekt gleich beim Aufrufer "eingefügt" wird.

Solche Situationen gibt es viele. Was ist hier zu tun, um sich trotzem das Kopieren zu ersparen? Weiter zum nächsten Schritt!

22. Wie kann man sich das Kopieren ersparen, wenn es ohne Kopieren nicht geht? Der Punkt ist, dass man das ursprüngliche Objekt nicht mehr benötigt. Dann einfach stehlen!

    Implementiere jetzt den Move-Konstruktor, sodass nur eine Logging-Ausgabe der Art "move ctor" ausgegeben wird. Der Prototyp wäre dann:

    ```
    Stack(Stack&&) noexcept;
    ```

    Jetzt solltest du sehen, dass der Aufruf des Move-Konstruktor vom Compiler generiert wird!

    Wie aber ist der Move-Konstruktor zu implementieren? Man könnte diesen manuell implementieren analog zu unserer ersten Version des Copy-Konstruktors oder aber wir verwenden wieder unsere schon implementierte Funktion `swap` (keine weitere Fehlerquelle!):

    ```
    Stack::Stack(Stack&& other) noexcept {
        swap(*this, other);
    }
    ```

    Beachte, dass `noexcept` wieder verwendet werden kann!

    Damit ist auch der Move-Konstruktor auch implementiert.

23. Betrachten wir in weiterer Folge Konstrukte der folgenden Art:

    ```
    Stack s;
    s = Stack{1, 2, 3};
    ```

    Auf der rhs der Zuweisung steht wieder ein temporäres Objekt, für das zuerst der entsprechende Konstruktor aufgerufen wird und danach in weiterer Folge der Copy-Konstruktor aufgerufen wird. Das ist natürlich auch nicht sonderlich schlau. Implementieren wir deshalb analog zum Copy-Assignment-Operator einen Move-Assignment-Operator mit folgendem Prototypen

    ```
    Stack& operator=(Stack&& rhs) noexcept;
    ```

    und folgendem Rumpf

    ```
    Stack tmp{std::move(rhs)};
    swap(*this, tmp);
    return *this;
    ```

Wir sehen hier die Verwendung der ”Funktion” `move`, die als Argument einen eine lvalue-Referenz erhält und daraus (zur Übersetzungszeit) diese eine rvalue-Referenz konvertiert, womit dann der Move-Konstruktor zum Zug kommt, um `tmp` zu initialisieren.

24. So, nachdem dies alles richtig implementiert wurde, können wir wieder den Compiler anweisen, selber einen Default-Konstruktor zu definieren und unseren Konstruktor weglöschen, den wir nur zum manuellen Testen verwendet hatten.
25. Kommen wir jetzt zu einer weiteren Regel, nämlich die ”rule of five”, die als Erweiterung der ”rule of three” aussagt, dass wenn eine dieser speziellen Elementfunktionen implementiert wird, alle fünf zu implementieren sind.

    Klarerweise wird diese Regel auch nur schlagend, wenn Move-Semantik erwünscht ist!
26. Kommen wir jetzt abschließend zu diesem Thema zur Regel ”rule of zero”, die besagt, dass wenn möglich keine der 5 speziellen Methoden implementiert werden sollten. Damit werden diese automatisch vom Compiler generiert, wenn alle Instanzvariablen (d.h. deren Typen) diese Methoden ebenfalls anbieten.
27. So, jetzt kommen wir schon langsam zum Schluss und werden noch einen Blick auf unsere Klassendeklaration werfen. Gibt es Methoden, die wir auch auf konstanten Objekten verwenden könnten?

    Ja, dann wäre es wieder an der Zeit solche Methoden als `const` zu kennzeichnen und zwecks Überprüfung einen kurzen Test zu schreiben, der diese Methoden verwendet.
28. Überlege einmal was bei der Methode `top()` alles schief gehen kann.
29. Prinzipiell könnten wir auch einzelne vom Compiler generierte Elementfunktionen explizit löschen. Dies funktioniert analog zu `= default` nämlich, dass `= delete` hinten angefügt wird.
30. Falls du es bis jetzt nicht erledigt hast, ist jetzt noch einmal Zeit das Programm auf Memory-Leaks zu untersuchen!
31. Lösche noch alle Testausgaben aus `main()` heraus, den wir werden diese nicht mehr benötigen.
32. Lösche weiters auch die `spdlog`-Ausgaben, denn ich denke, dass wir diese nicht mehr weiters benötigen. Ok, als ”debug”-Variante wären diese vielleicht noch interessant, aber so wirklich notwendig sind diese nicht, da ein Stack eben ein Stack ist und wir eh Unit-Tests geschrieben haben.

Ok, vom eigentlichen Rechner haben wir im Moment noch gar nichts implementiert, aber das werden wir auf ein anderes Mal verschieben und daher sind wir für den Moment

Fertig!!!

4 Übungszweck dieses Beispiels

- `spdlog` übersetzen, einbinden und verwenden (`cmake` und `make`, einbinden als statische Bibliothek)
- Einbinden eines Verzeichnisses in Meson mittels `subdir`
- Erstellen und verwenden einer statischen Bibliothek
- einfache Speicherverwaltung mit rohen Zeigern, Erkennen der Probleme!
- Implementierung eines Copy-Konstruktors
- `C++` Attribut `[[maybe_unused]]`

- Wiederholung dynamische Datenstrukturen, `data_structures_simple`!
- Referenzen als Rückgabewerte
- Wiederholung Datentypen `data_types`!
- Werfen von Exceptions und Kennenlernen der Standardexceptions
- Vertiefen des Schreibens von Unit-Tests
- Implementierung eines copy assignment operators
- Kennenlernen von `std::swap`
- Verwenden einer `using`-Deklaration, z.B. `using std::swap;`
- explizite Definition einer vom Compiler nicht-generierten Elementfunktion mittels `= default`
- *Rule of three, rule of five* und *rule of zero* kennenlernen
- Konstruktor mit `std::initializer_list` implementieren
- Konstruktoren mit `std::initializer_list` vs. Konstruktoren mit einem Argument bzgl. Initialisierung verstehen
- `noexcept` kennenlernen und verstehen
- Implementierung eines Move-Konstruktors
- Kennenlernen von `std::move`
- Implementierung eines move assignment operators
- explizites Löschen einer vom Compiler generierten Elementfunktion mittels `= delete`