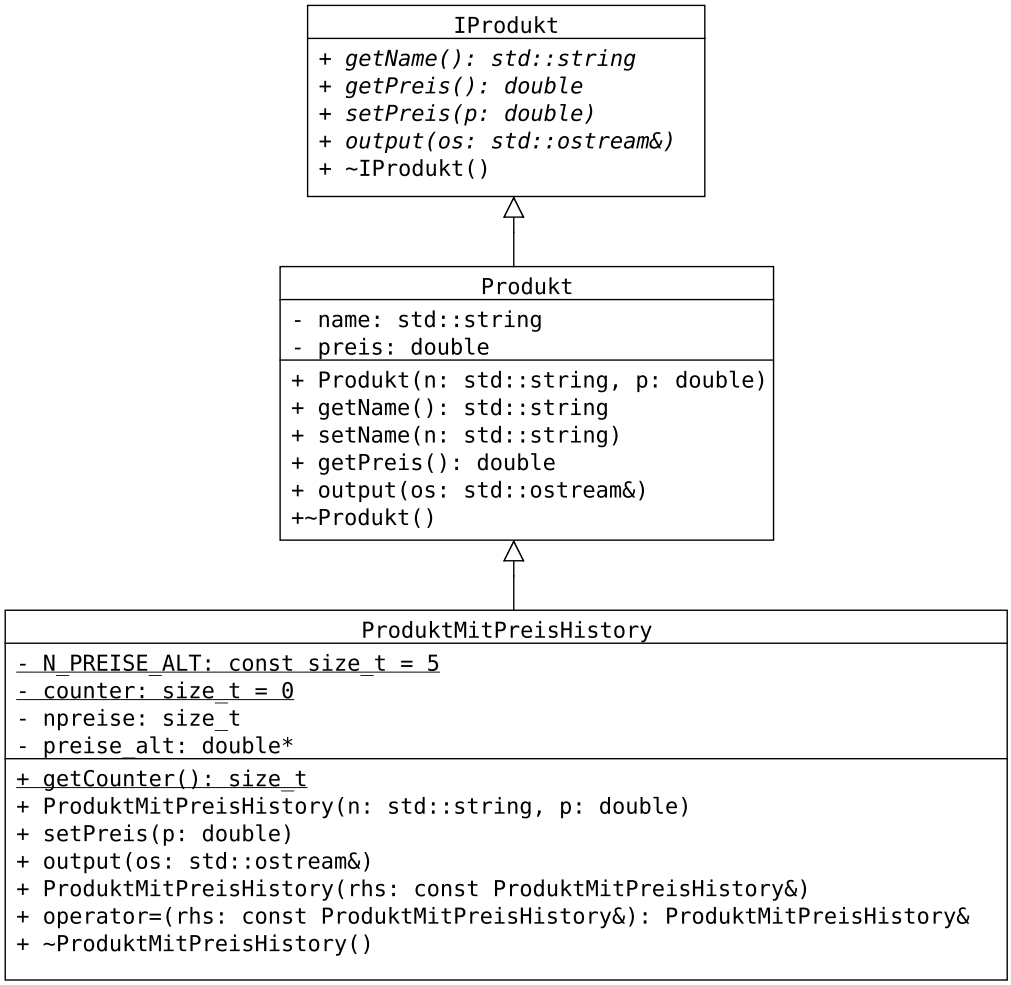
Zusammenfassende Aufgaben VPP

Vorbemerkung:

- Eine genaue Spezifikation des zu erstellenden Codes ist durch den bereitgestellten Google-Test gegeben (**TDD**).
- Deshalb können die folgenden Aufgaben **nur am Rechner zusammen mit dem bereitgestellten Google-Test Projekt** gelöst werden.
- In der Datei `AufgabenTest.pro` des Google-Test Projektes müssen Sie ggf. noch den Pfad für das Google-Test Verzeichnis für Ihr System anpassen.
- **Überprüfen Sie immer** mit einem Tool, das für Ihr Betriebssystem geeignet ist (Valgrind, Heob, oder Leaks), **Ihre implementierte Lösung auf Memory-Leaks**.

1) Aufgabe (Objektorientierung, einfache Klassen)

Vorgegeben sei das folgende UML-Klassendiagramm:



a) Fügen Sie dem in Moodle bereitgestellten Test-Projekt die neuen Dateien `iprodukt.h` und `iprodukt.cpp` hinzu und implementieren Sie darin die Schnittstelle `IProdukt`.

   In der Header-Datei überladen Sie weiterhin den Ausgabe-Operator `<<`.

b) Fügen Sie dem in Moodle bereitgestellten Test-Projekt die neuen Dateien `produkt.h` und `produkt.cpp` hinzu und implementieren Sie darin die Klasse `Produkt`. Die genaue Spezifikation der Methode `output()` ist durch den bereitgestellten Google-Test vorgegeben (s. u.).

c) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` erstellen Sie eine Funktion `void a01()` in die Sie Ihre eigenen Tests zur Schnittstelle `IProdukt` und der Klasse `Produkt` einfügen, z. B.:

   – Erstellen Sie auf dem Stack ein Objekt von Typ `Produkt`, lassen Sie dann eine Referenz vom Typ `IProdukt&` auf dieses Objekt verweisen. Verwenden Sie nun diese Referenz um alle erstellten Methoden aufzurufen.

   – Erstellen Sie auf dem Heap ein Objekt von Typ `Produkt`, lassen Sie dann einen Pointer vom Typ `IProdukt*` auf dieses Objekt verweisen. Verwenden sie nun diesen Pointer um alle erstellten Methoden aufzurufen.

     Vergessen Sie nicht danach den in Anspruch genommenen Heap-Speicher wieder frei zu geben.

d) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` erstellen Sie eine Funktion `void myMain()` in der Sie die Funktion `a01()` aufrufen.

e) Kommentieren Sie mit einem Block-Kommentar in der bereitgestellten Datei `tst_testaufgaben.h` alle Tests bis auf die ersten beiden Tests aus.

   Weiterhin passen Sie die vorgegebenen Include-Anweisungen durch zeilenweises Auskommentieren und Hinzufügen eigener Include-Anweisungen so an, dass sie zu dem bereits erstellten Code passen.

f) Rufen Sie dann `myMain()` in `main()` vor dem Aufruf

```
::testing::InitGoogleTest(&argc, argv);
```

   auf und führen Sie das Projekt aus.

g) Passen Sie insbesondere die Überschreibung der Methode `output()` so an, dass Ihr Code alle aktiven Tests besteht.

   Tipp:
   Hierzu müssen insbesondere Elemente, die in `<iomanip>` deklariert sind, verwendet werden.

2) Aufgabe (Vererbung, Big Three)

a) Fügen Sie dem in Moodle bereitgestellten Test-Projekt die neuen Dateien `produktmitpreishistory.h` und `produktmitpreishistory.cpp` hinzu und implementieren Sie darin die Klasse `ProduktMitPreisHistory` gemäß dem in der 1) Aufgabe dargestellten UML-Diagramm und den folgenden Vorgaben:

- Die Klasse `ProduktMitPreisHistory` speichert neben dem aktuellen Preis (geerbtes Attribut) noch zusätzlich die History der letzten `N_PREISE_ALT` alten Preise in einem **Array auf dem Heap**, auf den das Attribut `preise_alt` verweist.
- Die Klassen-Variable `counter` dient dazu, die Anzahl Objekte vom Typ `ProduktMitPreisHistory`, die zur jeweiligen Laufzeit vorhanden sind, zu zählen. Überlegen Sie sich, in welchen Methoden dieser Zähler angepasst werden muss.
- Die statische Methode `getCounter()` ermöglicht es, den Objekt-Zähler `counter` abzufragen.
- Im Attribut `npreise` wird gezählt, wie viele alte Preise bereits gespeichert sind. Dies geschieht aber nur so lange, bis `npreise` den Wert `N_PREISE_ALT` erreicht hat. In der überschriebene Methode `setPreis()` muss `npreise` ggf. inkrementiert werden, nachdem der alte Vorgänger-Preis an der nächsten freien Position im Array, auf das der Pointer `preise_alt` verweist, gespeichert wurde.

  Sind bereits schon alle `N_PREISE_ALT` Positionen belegt, muss durch Verschieben (nach Links) Platz geschaffen werden. Das an Index-Position 0 gespeicherte Element geht verloren und wird ersetzt durch das an Index-Position 1 gespeicherte Element, ..., das an Index-Position `N_PREISE_ALT-2` gespeicherte Element wird ersetzt durch das an Index-Position `N_PREISE_ALT-1` gespeicherte Element. Dies wird man natürlich mit einer Schleife umsetzen. Der alte Vorgänger-Preis kann dann an der Index-Position `N_PREISE_ALT-1` gespeichert werden.

  Vergessen Sie nicht danach das geerbte Attribut `preis` mit dem aktuellen Preis (Übergabe-Parameter `p`) zu aktualisieren.
- Die genaue Spezifikation für die Ausgabe durch die überschriebene Methode `output()` ist durch den bereitgestellten Google-Test festgelegt (s. u.).
- Weiterhin sind die Big-Three geeignet zu implementieren.

  Warum muss im Copy-Zuweisungs-Operator kein neuer Heap-Speicher angelegt werden?

b) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` erstellen Sie eine Funktion `void a02()` in die Sie Ihre eigenen Tests zur Klasse `ProduktMitPreisHistory` einfügen, z. B.:

- Erstellen Sie auf dem Stack ein Objekt von Typ `ProduktMitPreisHistory`, lassen Sie dann eine Referenz vom Typ `IProdukt&` auf dieses Objekt verweisen. Verwenden Sie nun diese Referenz um alle erstellten Methoden aufzurufen.
- Erstellen Sie auf dem Heap ein Objekt von Typ `ProduktMitPreisHistory`, lassen Sie dann einen Pointer vom Typ `IProdukt*` auf dieses Objekt verweisen. Verwenden sie nun diesen Pointer um alle erstellten Methoden aufzurufen.

  Vergessen Sie nicht danach den in Anspruch genommenen Heap-Speicher wieder frei zu geben.

c) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` rufen Sie in der Funktion `void myMain()` die Funktion `a02()` auf, den Aufruf von `a01()` können Sie auskommentieren.

d) Verschieben Sie den Block-Kommentar in der bereitgestellten Datei `tst_testaufgaben.h` so weit nach unten, so dass alle Tests für die Klasse `ProduktMitPreisHistory`, diese haben als ersten Makro-Parameter `ProduktMitPreisHistory`, aktiv werden.

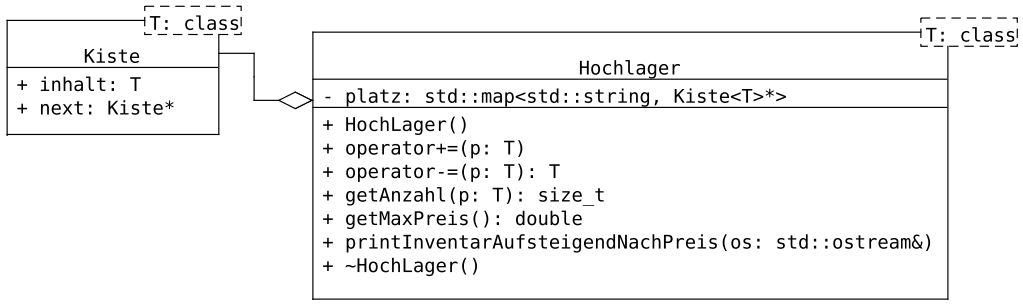
   Weiterhin passen Sie die vorgegebenen Include-Anweisungen durch zeilenweises Auskommentieren und Hinzufügen eigener Include-Anweisungen so an, dass sie zu dem bereits erstellten Code passen.

e) Passen Sie insbesondere die Überschreibung der Methode `output()` so an, dass Ihr Code alle aktiven Tests besteht.

   Tipp:
   Hierzu müssen insbesondere Elemente, die in `<iomanip>` deklariert sind, verwendet werden.

3) Aufgabe (Templates, Verkettete Listen)

Vorgegeben sei das folgende UML-Klassendiagramm:



a) Fügen Sie dem in Moodle bereitgestellten Test-Projekt die neue Datei `hochlager.h` hinzu und implementieren Sie darin die Template-Klassen `Kiste` und `Hochlager` gemäß dem oben dargestellten UML-Diagramm und den folgenden Vorgaben:

- Objekte vom Typ `Hochlager` dienen dazu, die Lagerung von Produkten (Typ-Parameter `T`) in einem Hochlager nachzubilden.

  Für jede Produkt-Art (eindeutig festgelegt durch ihren Namen) ist ein eigener Lagerplatz (ausziehbares Schubfach) vorgesehen, der über das Attribut `platz` gespeichert wird.

  Die im Hochlager vorrätigen Artikel der gleichen Produkt-Art werden in Kisten gelagert, die im vorgesehenen ausziehbaren Lagerplatz (Schubfach) hintereinander (modelliert als **einfach verkettete Liste von Kisten-Objekten**) gelagert werden.

  Mit der Hilfe des Attributes `platz` erhält man über den Namen des Produktes (der im Map als Schlüssel verwendet wird) den Anfangs-Zeiger (der im Map als Wert verwendet wird) auf die verkettete Liste von Kisten die die Artikel der gleichen Produkt-Art enthalten.

  Wird ein Schubfach mit neuen Artikeln der gleichen Produkt-Art aufgefüllt, werden sie zunächst in Kisten gestellt. Die Kisten werden dann am Ende des Schubfaches eingefügt.

  Wird ein Artikel aus einem Schubfach entnommen, geschieht das dadurch, dass die vorderste Kiste aus dem Schubfach entfernt wird und aus ihr der gewünschte Artikel entnommen wird.
- Wie man sich leicht überlegt, ist für den Standard-Konstruktor eine leere Implementierung ausreichend.
- Mit der Hilfe des überladenen `+=` Operators wird ein Artikel `p` einer bestimmten Produkt-Art dem Hochlager hinzugefügt.

  Hierzu muss zunächst der Artikel in eine Kiste abgelegt werden. Dann ist über den Namen des Artikels der Lagerplatz für seine Produkt-Art zu finden. Hierzu kann die Methode `find()` der Klasse `std::map` verwendet werden.

War die Artikel-Art noch nicht im Hochlager vorhanden, wird dem Map `platz` ein neues Schlüssel/Werte Paar hinzugefügt. Dabei ist der Name der Produkt-Art als Schlüssel und der Pointer auf die Kiste, in die der neue Artikel abgelegt wurde, als Wert zu verwenden.

Ist die Artikel-Art schon im Hochlager vorhanden erhält man über den Namen der Produktart und das Attribut `platz` den Anfangs-Pointer für die verkettete Liste, in die Kiste mit dem neuen Artikel am Ende eingefügt werden muss.

Beachten Sie dass aufgrund der hier speziellen Verwendung des überladenen `+=` Operators dieser den Rückgabe-Typ `void` hat (im Fall einer arithmetischen Semantik hingegen besitzt er üblicherweise den Typ Referenz auf die eigene Klasse als Rückgabe-Typ).

- Mit der Hilfe des überladenen `-=` Operators wird ein Artikel `p` einer bestimmten Produkt-Art aus dem Hochlager entnommen.

  Ist die Artikel-Art im Hochlager nicht vorhanden, ist eine Ausnahme, die im Unit-Test genau spezifiziert ist, auszuwerfen.

  Ansonsten wird die erste vorderste Kiste (Listenanfang) aus dem zugeordneten Schubfach entnommen und der Artikel, der in ihr abgelegt war, als Ergebnis zurück gegeben. Bei der Modellierung im Programm muss vor der Rückgabe das Kisten-Objekt vom Heap gelöscht werden.

  Falls es die letzte Kiste war, die im Schubfach für die Produkt-Art abgelegt war, muss die Produkt-Art ganz aus dem Hochlager entfernt werden. Im Programm kann dies durch den Aufruf der Methode `erase()` über das Attribut `platz` erfolgen (siehe Dokumentation für `std::map`).

- Die Methode `getAnzahl()` liefert die Anzahl der im Hochlager vorhandenen Artikel, die die Produkt-Art `p` besitzen, als Ergebnis zurück.

  Um diese Anzahl zu ermitteln, muss gezählt werden, wie viel Kisten im zugeordneten Schubfach vorhanden sind. Dazu muss die dem Schubfach entsprechende verkettete Liste vollständig durchlaufen werden.

- Die Methode `getMaxPreis()` liefert den maximalen Preis aller Produkt-Arten, die im Hochlager vorhanden sind, als Ergebnis zurück.

- Die Methode `printInventarAufsteigendNachPreis()` druckt in der durch den Unit-Test genau spezifizierten Form das Inventar der Produkte, die im Hochlager gespeichert sind, auf den übergebenen Stream `os` aus.

  Zum Sortieren kann man ein temporär gespeichertes Objekt von Typ `std::vector` und die Methode `std::sort()` verwenden, wobei ein geeignet formulierter Lambda-Ausdruck als First-Class-Funktion übergeben wird.

- Im Destruktor sind zunächst für alle im Hochlager vorhandenen Produkt-Arten die Schubfächer zu leeren (d. h. die Verketteten Listen vom Heap zu löschen).

  Mit Hilfe der Methode `clear()` kann man danach alle Elemente aus dem Map `platz` entfernen.

- **Was besagt die Rule of Three?** Ist diese für die Klasse `Hochlager` erfüllt?
- In dem hier betrachteten Fall soll das Kopieren von Objekten vom Typ `Hochlager` unterbunden werden.

  Welche Anweisungen sind hierfür in der Definition der Klasse `Hochlager` erforderlich?

  Fügen sie diese Anweisungen Ihrem Code hinzu.

b) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` erstellen Sie eine Funktion `void a03()` in die Sie Ihre eigenen Tests zur Klasse `Hochlager` einfügen.

c) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` rufen Sie in der Funktion `void myMain()` die Funktion `a03()` auf, den Aufruf von `a02()` können Sie auskommentieren.

d) Verschieben Sie den Block-Kommentar in der bereitgestellten Datei `tst_testaufgaben.h` so weit nach unten, so dass alle Tests für die Klasse `Hochlager`, diese haben als ersten Makro-Parameter `Hochlager`, aktiv werden.

Weiterhin passen Sie die vorgegebenen Include-Anweisungen durch zeilenweises Auskommentieren und Hinzufügen eigener Include-Anweisungen so an, dass sie zu dem bereits erstellten Code passen.

e) Passen Sie insbesondere die Überschreibung der Methode `printInventarAufsteigendNachPreis()` so an, dass Ihr Code alle aktiven Tests besteht.

f) **Unter Windows** verursacht der Test `TEST(HochLager, Exceptions)` bei der Analyse mit Heob aus mir unerklärlichen Gründen einen **Memory-Leak** mit der Meldung

```
32 bytes in 1 blocks are lost in loss record 1 of (#464).
```

Unter Linux mit valgrind tritt dieser Fehler nicht auf!

Unter MacOS kann ich das nicht überprüfen und bitte um Nachricht.

Nach längerer Untersuchung habe ich festgestellt, dass dieser Fehler im Zusammenhang mit Google-Test unter Windows dann auftritt, wenn Exceptions ausgeworfen werden, auch wenn der eigene Code keinen Heap-Speicher in Anspruch nimmt. Dies muss ein Fehler in Google-Test oder im MinGW-Compiler oder in Heob sein.

Nachdem Ihr Programm den Test `TEST(HochLager, Exceptions)` bestanden hat, deaktivieren Sie deshalb unter Windows diesen Test mit einem Block-Kommentar.

4) Aufgabe (Funktionale Programmierung, STL)

a) Fügen Sie dem in Moodle bereitgestellten Test-Projekt die neuen Dateien `a04.h` und `a04.cpp` hinzu.

b) Erstellen Sie in der Datei `a04.h` einen Generator (Funktor, dessen überladener Aufruf-Operator keine Übergabe-Parameter besitzt) `Collatz`, der für einen im Konstruktor übergebenen Anfangswert `start` die Collatz-Folge generiert.

Bekanntlich werden die Glieder der Collatz-Folge $(c_i)_{i\in\mathbb{N}}$ wie folgt gebildet:

$$\begin{aligned} c_1 &= \text{start} \\ c_{i+1} &= \frac{c_i}{2} \qquad \text{für } i \geq 1 \text{ und } c_i \text{ ist gerade} \\ c_{i+1} &= 3 \cdot c_i + 1 \quad \text{für } i \geq 1 \text{ und } c_i \text{ ist ungerade} \end{aligned}$$

c) Erstellen Sie in der Datei `a04.cpp` eine Funktion `void a04()`, in der Sie sich ein Generator Objekt `c10` vom Typ `Collatz` mit Startwert `start = 10` erstellen.

Rufen Sie in einer Schleife das Generator-Objekt `c10` 15 mal auf und geben Sie jeweils das Ergebnis des Aufrufes nach `cout` aus.

Wenn Sie den Code ausführen (s. Teil d)), sollten Sie erkennen, dass die Folge in den Zyklus $4, 2, 1, 4, 2, 1, \ldots$ herein läuft.

Im Jahr 1937 hat Lothar Collatz die sog. **Collatz-Vermutung** aufgestellt, dass für alle Startwerte start $\in \mathbb{N}$ immer die Folge mit dem Zyklus $4, 2, 1$ endet.

Trotz zahlreicher Anstrengungen gehört diese Vermutung heute **noch immer zu den ungelösten Problemen der Mathematik**. Mehrfach wurden Preise für eine Lösung ausgelobt.

d) In der bereitgestellten Datei `testZusatzAufgabenMain.cpp` rufen Sie in der Funktion `void myMain()` die Funktion `a04()` auf, den Aufruf von `a03()` können Sie auskommentieren. Führen Sie nun das Projekt aus.

e) Fügen Sie in der Datei `a04.cpp` in der Funktion `a04()` Code hinzu, der folgendes bewirkt:

- Erstellen Sie einen STL-Container `vec` vom Typ `std::vector` in dem 30 `int`-Werte gespeichert werden können und alle Elemente mit 0 initialisiert sind.

  Warum muss man hierfür den Konstruktor mit den runden Klammern `()` aufrufen und kann nicht die Braces `{}` verwenden?
- Verwenden Sie den erstellten Generator `Collatz` und den STL-Algorithmus `std::generate()` um den Container `vec` mit den Gliedern der Collatz-Folge zu füllen, die den Startwert 100 hat.

- Geben Sie danach **sowohl mit einer bereichsbasierten Schleife als auch mit einem Iterator** alle in `vec` gespeicherten Elemente in einer einzigen Ausgabe-Zeile, in der die einzelnen Elemente durch ein Leerzeichen getrennt sind, nach `cout` aus.

f) Erstellen Sie in der Datei `a04.h` das Funktions-Template

```
template <class G, class P>
int laenge(G& g, P p)
```

Der erste Übergabe-Parameter `g` des Funktions-Templates ist ein Generator, mit dem die Glieder einer Folge erzeugt werden können.

Mit dem Funktions-Template soll es möglich sein die Länge (Anzahl Anfangs-Folgen-Glieder) zu bestimmen für die das im zweiten Übergabe-Parameter `p` übergebene Prädikat (s. Teil g)) erfüllt ist (den Wert `true` für ein übergebenes Folgen-Glied zurück liefert) bis das Prädikat zum ersten mal nicht erfüllt ist (den Wert `false` für ein übergebenes Folgen-Glied zurück liefert).

g) Erstellen Sie in der Datei `a04.h` das Prädikat (Funktor, der einen boolschen Wert als Ergebnis zurück liefert) `PCollatz`, mit dem unter Verwendung des in Aufgaben-Teil f) erstellten Funktions-Templates `laenge()` bestimmt werden kann, ab welchem Glied die Collatz-Folge zum ersten mal den Wert 1 annimmt.

h) Bekanntlich kann man aufrufbare Elemente (normale Funktionen, Funktoren (also auch Prädikate), Lambda-Ausdrücke) in Variablen speichern, die mit der Hilfe des Templates `std::function` (erfordert das Inkludieren von `<functional>`) definiert wurden.

Definieren Sie in der Datei `a04.h` unter Verwendung von `std::function` den neuen Typ `TFC` (`using` Anweisung in der neuen Form oder `typedef` Anweisung in der alten Form), so dass Variablen von diesem Typ auch Objekte vom Typ `PCollatz` speichern können.

Fügen Sie in der Funktion `a04()` in der Datei `a04.cpp` folgenden Code hinzu:

- Erstellen Sie neues Generator-Objekt `cfunk` vom Typ `Collatz`, mit dem man die Glieder der Collatz-Folge mit Startwert 100 erzeugen kann.
- Erstellen Sie ein Prädikat `pc` vom Typ `PCollatz`.
- Rufen Sie nun das Funktions-Template `laenge()` mit den Parametern `cfunk` und `pc` auf und geben Sie das Ergebnis nach `cout` aus.
- Definieren Sie nun eine Variable `fc` vom Typ `TFC` und initialisieren Sie sie mit dem Prädikat `pc`.
- Erstellen Sie neues Generator-Objekt `cfunk2`, mit dem man die Glieder der Collatz-Folge mit Startwert 100 erzeugen kann (**Warum muss ein neues Generator-Objekt erstellt werden?**).

  Rufen Sie dann nochmals das Funktions-Template `laenge()` aber mit den Parametern `cfunk2` und `fc` auf und geben Sie das Ergebnis nach `cout` aus.

- Anstelle eines Prädikates kann man dem Funktions-Template `laenge()` als 2-ten Parameter auch einen geeigneten Lambda-Ausdruck direkt übergeben.
  Erstellen Sie neues Generator-Objekt `clam`, mit dem man die Glieder der Collatz-Folge mit Startwert 100 erzeugen kann.
  Rufen Sie dann nochmals das Funktions-Template `laenge()` aber mit den Parametern `clam` und einem geeigneten Lambda-Ausdruck auf und geben Sie das Ergebnis nach `cout` aus.
- Den zuvor formulierten Lambda-Ausdruck kann man auch zuerst in der Variablen `fc` speichern, und dann wiederum `fc` als zweiten Parameter für den Aufruf vom Funktions-Template `laenge()` verwenden.
  Probieren Sie dies ebenfalls aus.

i) Als zweiten Parameter für den Aufruf vom Funktions-Template `laenge()` kann man auch eine (normale) Funktion verwenden.

Fügen Sie der Datei `a04.h` den Prototypen `bool pcollatz(int a);` hinzu und implementieren Sie diese Funktion in der Datei `a04.cpp`.

Definieren Sie in der Datei `a04.h` den Funktions-Pointer Typ `TFP` (`using` Anweisung in der neuen Form oder `typedef` Anweisung in der alten Form), so dass Variablen von diesem Typ z.B. die Funktion `pcollatz()` zugewiesen werden kann.

In der Funktion `a04()` in der Datei `a04.cpp` deklarieren Sie eine Variable `fp` vom Typ `TFP` und initialisieren Sie sie mit der Funktion `pcollatz()`.

Erstellen Sie neues Generator-Objekt `cfp`, mit dem man die Glieder der Collatz-Folge mit Startwert 100 erzeugen kann.

Rufen Sie dann nochmals das Funktions-Template `laenge()` aber mit den Parametern `cfp` und und `fp` auf und geben Sie das Ergebnis nach `cout` aus.

Kann man der Variablen `fc` den Wert der Variablen `fp` (Funktions-Pointer) zuweisen und dann wiederum `fc` anstelle von `fp` als Parameter für den Aufruf des Funktions-Template `laenge()` verwenden?

j) Erstellen Sie in der Datei `a04.h` das Funktions-Template

```
template <int N, class G, class P>
int maxLaengeCollatz(P p)
```

mit dem der Startwert einer Folge, deren Generator den Typ `G` hat, bestimmt werden kann, die von allen Startwerten im Bereich $5 \leq \text{start} \leq N$ die maximale Länge hinsichtlich des Prädikates `p` hat.

In der Funktion `a04()` in der Datei `a04.cpp` fügen sie Code zum Testen dieses Funktions-Templates hinzu wobei der nicht-Typ Parameter `N` den Wert 1000 hat.

Verwenden Sie für den Übergabe-Parameter `p` alle Möglichkeiten (Objekt vom Typ `PCollatz`, Funktion `pcollatz()`, Lambda-Ausdruck, Variable vom Typ `TFP`, Variable vom Typ `TFC` initialisiert sowohl mit Objekt vom Typ `PCollatz`, oder mit einem Lambda-Ausdruck, oder mit einem Funktions-Pointer.

i) Löschen Sie den Block-Kommentar in der bereitgestellten Datei `tst_testaufgaben.h` so dass alle 15 Tests aktiv werden.

   Ihre Lösung sollte alle 15 Tests (unter Windows 14 Tests siehe 3) Aufgabe Teil f)) bestehen und keine Memory-Leaks enthalten.