

# Übung zum C/C++-Praktikum

## Grundlagen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

### Übungsblatt 1 – Sommersemester 2025

Die Aufgaben für das C/C++-Praktikum sind thematisch sortiert. Zu Beginn jedes Themengebiets der Vortragsfolien ist vermerkt, welche Aufgaben zu diesem Themengebiet gehören ([G]: Grundlagen; [S]: Speicherverwaltung; [O]: Objektorientierung; [F]: Fortgeschrittene Themen; [C]: (Embedded) C; [Z]: Optionale Zusatzaufgaben (C++)).

## Einführung

Für alle Übungen des C/C++-Praktikums wird CLion als IDE verwendet. Als Compiler kommt MinGW zum Einsatz.

Alle Materialien zur Vorlesung und Übung sind im Moodle-Kurs der Veranstaltung zu finden. Dieses beinhaltet zum einen die Vorlesungsfolien, zum anderen diese Übungsblätter und alle Lösungen zu den Übungsaufgaben.

## Hinweise

- Bei Fragen und Problemen bitte aktiv um Hilfe!
- Alle Lösungen zum C++-Teil enthalten ein CMakefile und können entweder über die Kommandozeile mit Hilfe von `cmake -B build` gebuildet werden, oder direkt aus CLion heraus, indem du sie als Projekt importierst. Dann kannst du, wie im Tutorial gezeigt, einfach über entsprechende Schaltflächen in CLion builden und ausführen.
- Folgende Tastenkürzel könnten sich dir im Verlauf des Praktikums als nützlich erweisen:

Tastenkürzel	Befehl	Beschreibung
Ctrl+Space F2	Autocomplete Navigate between errors	Anzeige von Vervollständigungshinweisen (z. B. nach <code>std::</code> oder <code>main</code> ) Schneller Wechsel von der Anzeige von markierten Fehlermeldungen am Code
Ctrl+Alt+S	Settings	Öffnen der generellen Einstellungen von CLion
Ctrl+N	New	Anlegen neuer Datei
Double Shift	Search everywhere	Gesamtsuche in allen Ordnern/Klassen nach z. B. Hilfsaktionen
Ctrl + F9	Build	Startet den Buildprozess (Aufruf von Compiler und Linker)

Es können auch eigene Tastenkürzel erstellt werden. Hierzu einfach Ctrl + Alt + S tippen und unter Keymap dann für die ausgewählte Aktion ein Tastenkürzel wählen.

## Projekte in CLion importieren



Die angebotenen Musterlösungs-/Microcontroller-Projekte basieren auf CMake. Daher ist es wichtig, dass du die zu jedem Projekt gehörende `CMakeLists.txt` entsprechend importierst: *File* → *Open* und dann zum entsprechenden Projektordner navigieren. Dann den Projektordner importieren, indem du auf OK klickst.

---

## Aufgabe 1: [G] Hello World

---

Lege ein neues C++-Projekt an, indem du *New* → *New project* im CLion-Menü (die drei Balken oben links) wählst. Bei der Übersicht der ganzen Projektarten wählst du dann C++ Executable aus. Falls dann der gewünschte Projektpath und der passende Projektname z. B. `Tag1_Aufgabe1` ausgewählt wurde, kannst du auf `create` klicken.

Das Projekt enthält bereits eine Datei `src/main.cpp`. Ergänze die `main`-Funktion, sodass sie der gezeigten entspricht und kompiliere das Projekt mit einem Klick auf das Build-Symbol . Führe dann das Programm mit einem Klick auf das Run-Symbol  aus.

```
#include <iostream>
int main() {
    std::cout << "Hello World" << std::endl; // prints "Hello World"

    return 0;
}
```

Jedes vollständige C++-Programm muss **genau eine** Funktion mit Namen `main` und Rückgabetypen `int` außerhalb von Klassen im globalen Namensraum besitzen. Andernfalls wird der Linker mit der Fehlermeldung *undefined reference to 'main'* abrechnen. Der Rückgabotyp wird verwendet, um dem Aufrufer (Betriebssystem, Shell, ...) den Erfolg oder Misserfolg der Ausführung zu signalisieren. Typischerweise wird im Erfolgsfall 0 zurückgegeben.

Die erste Zeile des obigen Programms bindet den Header der `iostream` Bibliothek ein, welche unter anderem Klassen und Funktionen zur Ein- und Ausgabe mit Hilfe von `<<` (*insertion operator*) und `>>` (*extraction operator*) anbietet. Diese Bibliothek ist Teil der C++-Standardbibliothek, welche eine Sammlung an generischen Containern, Algorithmen und vielen häufig genutzten Funktionen ist. Um auf die Elemente dieser Bibliothek zuzugreifen, muss man ihren `namespace` (in diesem Fall `std`) voranstellen, gefolgt von zwei Doppelpunkten und dem gewünschten Element (in diesem Fall `cout` und `endl`). Um Überschneidungen mit eigenen Definitionen zu vermeiden ist es üblich, Bibliotheken in einem `namespace` zu kapseln, welcher analog zu `package` in Java funktioniert, jedoch nicht an Ordnerstrukturen gebunden ist.

In der dritten Zeile wird der String `"Hello World"` in `std::cout` eingefügt, gefolgt von `std::endl`, das einen Zeilenumbruch erzeugt und die Ausgabepuffer leert. Für weitere Informationen zur Kommandozeilenausgabe siehe [http://www.cplusplus.com/doc/tutorial/basic\\_io/](http://www.cplusplus.com/doc/tutorial/basic_io/) und <http://www.cplusplus.com/reference/iomanip/>.

### Hinweise

- Einzeilige Kommentare können durch `//`, mehrzeilige durch `/* ... */` eingeschlossen werden.
- Anders als in Java können Funktionen auch außerhalb von Klassen definiert und verwendet werden.
- Die `return` Anweisung darf in der `main` Funktion weggelassen werden.

---

### Häufige (Compiler-)Fehlermeldungen

---

Im Folgenden sind einige Fehlermeldungen von `clang` zusammen mit möglichen Lösungsstrategien aufgelistet. Die generelle Faustregel lautet: **Kompilierfehler sollten immer von oben nach unten abgearbeitet werden, so wie sie in der Konsole erscheinen.** Der Grund hierfür ist, dass es durch einen Fehler zu weiteren Folgefehlern kommen kann.

```
main.exe: not found
```

Dieser Fehler wird von CLion ausgegeben, wenn es nach dem Kompilieren das lauffähige Programm nicht findet. Das kann zwei Gründe haben:

- Der Kompiliervorgang ist gescheitert. Prüfe die Konsole auf Fehler.
- Der Kompiliervorgang wurde noch nicht ausgeführt. Kompiliere das Programm mit einem Klick auf das Build-Symbol.

```
error: expected ';' before ...
```

Dies bedeutet, dass in der Zeile davor ein ; vergessen wurde. Allgemein beziehen sich Fehlermeldungen **expected ... before ...** häufig auf die Zeile **vor** dem markierten Statement. Beachte, dass *die Zeile davor* auch die letzte Zeile einer eingebundenen Header-Datei sein kann. Beispiel:

```
#include "main.h"
int main() {
    ...
}
```

Falls im Header `main.h` in der letzten Zeile ein Semikolon fehlt, wird der Compiler die Fehlermeldung trotzdem auf die Zeile beziehen, in der `int main() {` steht.

```
error: invalid conversion from <A> to <B>.
```

Dies bedeutet, dass der Compiler an der entsprechenden Stelle einen Ausdruck vom Typ *B* erwartet, im Code jedoch ein Ausdruck vom Typ *A* angegeben wurde. Insbesondere bei verschachtelten Typen sowie (später vorgestellten) Zeigern und Templates kann die Fehlermeldung sehr lang werden. In so einem Fall lohnt es sich, den Ausdruck in mehrere Teilausdrücke aufzubrechen und die Teilergebnisse durch temporäre Variablen weiterzureichen.

```
undefined reference to ...
```

Dies bedeutet, dass das Programm zwar korrekt kompiliert wurde, der Linker aber die Definition des entsprechenden Bezeichners nicht finden kann. Das kann passieren, wenn man dem Compiler durch einen Prototypen mitteilt, dass eine bestimmte Funktion existiert (**deklariert**), diese aber nirgendwo tatsächlich **definiert**. Überprüfe in diesem Fall, ob der Bezeichner tatsächlich definiert wurde und ob die Signatur der Definition mit dem Prototypen übereinstimmt.

---

## Aufgabe 2: [G] C++: Grundlagen, Funktionen und Strukturierung

---

Für diese Aufgabe findest du im Moodle-Kurs einen Lösungsvorschlag in Form eines Zip-Archivs. Die Lösung ist dort im Unterordner `/basics` hinterlegt. Für diese Aufgabe kannst du entweder das vorherige Programm weiterentwickeln oder genauso wie vorher ein neues Projekt anlegen.

---

### Primitive Datentypen

---

Die primitiven Datentypen in C++ sind ähnlich zu denen in Java. Allerdings sind alle Ganzzahl-Typen in C++ sowohl mit als auch ohne Vorzeichen verfügbar. Standardmäßig sind Zahlen vorzeichenbehaftet. Mittels `unsigned` kann man vorzeichenlose Variablen deklarieren. Durch das freie Vorzeichenbit kann ein größerer positiver Wertebereich dargestellt werden.

```
int i;           // signed int, -2147483648 to +2147483647 on a 32-bit machine
unsigned int ui; // unsigned int, 0 to 4294967295 on a 32-bit machine
// unsigned double d;           // not possible
```

Eine andere Besonderheit von C++ ist, dass Ganzzahlwerte implizit in Boolesche Werte (Typ: `bool`) umgewandelt werden. Werte, die ungleich 0 sind, werden als `true` gewertet, 0 als `false`. Somit können Ganzzahlen direkt in Bedingungen ausgewertet werden.

---

### 2a) Größe von Datentypen

---

Oft ist es notwendig, die tatsächliche Größe der Datentypen im Speicher zu kennen. Diese kann mit Hilfe des `sizeof`-Operators ermittelt werden. Deshalb sollst du dir in dieser Aufgabe die Größe der folgenden Datentypen in Bytes, wie auch deren minimalen und maximalen Wert ausgeben lassen.

```
int
unsigned int
double
unsigned short
bool
```

### Hinweise

- Die C++ Klasse `std::numeric_limits`<sup>1</sup> bietet Funktionen sich minimale und maximale Werte von Datentypen ausgeben zu lassen. Einbinden lässt sich diese über den Header `limits`.

---

### 2b) Sternenmuster mit Funktionen malen

---

Schreibe eine Funktion `printStars(int n)`, die `n`-mal einen Stern (\*) auf die Konsole ausgibt und mit einem Zeilenumbruch abschließt. Ein Aufruf von `printStars(5)` sollte folgende Ausgabe generieren:

```
*****
```

Platziere die Funktion **vor** `main`, da sie ansonsten von dort aus nicht aufgerufen werden kann. Erstelle nun eine zweite Funktion `printFigure(int n)`, die `printStars(int n)` nutzt, um eine Figur wie unten dargestellt auszugeben. Verwende hierzu eine Schleife.

---

<sup>1</sup>[http://en.cppreference.com/w/cpp/types/numeric\\_limits](http://en.cppreference.com/w/cpp/types/numeric_limits)

```
*****
*****
***
**
*
**
***
****
*****
```

## Hinweise

- Was die Benennung von Funktionen, Variablen und Klassen angeht, bist du frei. Für Klassen ist „CamelCase“ wie in Java üblich. Bei Funktionen und Variablen wird zumeist entweder auch CamelCase oder Kleinschreibung mit Unterstrichen verwendet.
- Um Strings auszugeben, stellt dir C++ den Stream `std::cout` zur Verfügung, welches den String auf die Standardausgabe ausgibt. Einen Zeilenumbruch erreicht man, indem man das Escape-Zeichen `\n` in einen String oder als einzelnen `char`-Wert eingibt. Die STL stellt auch den „Manipulator“ `std::endl` zur Verfügung, der neben den Zeilenumbruch auch ein „Flushen“ des Ausgabepuffers erzwingt. (Je nachdem, wieviele Ausgaben erfolgen, kann dies den Ablauf deutlich verlangsamen.)

---

## 2c) Erweiterung durch einen enum-Typen

---

Ein Aufzählungstyp (engl. enumerated type) ist ein Datentyp, dessen Wert auf eine definierte Menge begrenzt ist. Alle möglichen Werte werden bereits bei der Deklaration des Typs mit einem eindeutigen Namen angegeben. Um einen solchen Datentyp zu deklarieren, benutzt man das Schlüsselwort `enum`:

```
enum Fruit {
    APPLE,
    BANANA,
    CHERRY
};
```

Die Werte in den geschweiften Klammern sind Symbole, die intern als Zahlen (Integer) gespeichert sind. Die Werte beginnen normalerweise bei 0, man kann sie aber auch selbst festlegen:

```
enum Fruit1 {
    APPLE = 4,
    BANANA,
    CHERRY
};
// APPLE == 4
// BANANA == 5
// CHERRY == 6

enum Fruit2 {
    APPLE = 4,
    BANANA = 10,
    CHERRY
};
// APPLE == 4
// BANANA == 10
// CHERRY == 11
```

Die Namen der Werte einer „klassischen“ Aufzählung sind global in dem Scope gültig, in dem die Aufzählung deklariert ist. Der Compiler verbietet, dass mehrere gleiche Namen für Aufzählungswerte in einem Scope vorliegen, da diese dann ja nicht unterscheidbar wären. D. h. die Beispiele mit den Aufzählungen `Fruit`, `Fruit1` und `Fruit2` könnten nicht zusammen verwendet werden.

Ab C++11 steht mit `enum class` eine weitere Möglichkeit zur Verfügung, Aufzählungen zu deklarieren. Hier gehören die definierten Werte zum Namensraum (siehe unten) des Aufzählungstyps, und der Zugriff muss mit einem doppelten Doppelpunkt erfolgen, wie folgendes Beispiel zeigt:

```
enum class Fruit {
    Apple,
    Banana,
    Cherry
};

enum class Fruit2 {
    Apple = 4,
    Banana = 10,
    Cherry
};

// Fruit::Apple == 0
// Fruit::Banana == 1
// Fruit::Cherry == 2

// Fruit2::Apple == 4
// Fruit2::Banana == 10
// Fruit2::Cherry == 11
```

Nutzen kann man den Enum-Typ nun, indem man eine Variable von diesem Typ deklariert und diese wie gewohnt verwendet:

```
Fruit myFruit = Fruit::Apple;

if (myFruit == Fruit::Banana) {
    // Do something
}
```

Deine Aufgabe ist es jetzt, das Programm um einen Enum-Typen `Direction` zu erweitern, der die Richtung (`Left`, `Right`) angibt, in die das Pattern ausgegeben werden soll. Erweitere dazu, falls nötig, die Funktionen `printStars` bzw. `printFigure`. Es könnte außerdem nützlich sein, eine Funktion `printSpaces` einzuführen. Das Resultat soll am Ende jeweils folgendermaßen aussehen:

```
//Left-aligned      or      Right-aligned
*****              *****
****                ****
***                 ***
**                  **
*                   *
**                  **
***                 ***
****                ****
*****              *****
```

---

## 2d) Auslagern der Datei

---

Erstelle eine neue Header-Datei **functions.hpp** und eine neue Sourcdatei **functions.cpp**. Klicke hierzu mit der **rechten Maustaste** auf den Ordner **src** und wähle **New C++ Class** aus und gebe dann der Datei den Namen **functions**. Bestätige den Dialog mit **OK**. Wiederhole diesen Schritt entsprechend für C++ Source File und ebenfalls dem Namen **functions**. Füge in der Headerdatei die folgenden Include-Guards hinzu.

```
#ifndef FUNCTIONS_HPP_
#define FUNCTIONS_HPP_
// your header ...
#endif /* FUNCTIONS_HPP_ */
```

Binde danach **functions.hpp** in beide Sourcdateien (**functions.cpp** und **main.cpp**) ein, indem du

```
#include "functions.hpp"
```

verwendest. **Verschiebe** die beiden zuvor erstellten Funktionen nach `functions.cpp`.

Schreibe nun in `functions.hpp` **Funktionsprototypen** für die Funktionen, die sich nun in `functions.cpp` befinden. Funktionsprototypen dienen dazu, dem Compiler mitzuteilen, dass eine Funktion mit bestimmtem Namen, Parametern und Rückgabetyp existiert. Ein Prototyp ist eine mit ; abgeschlossene Signatur der Funktion ohne Funktionsrumpf. Der Prototyp von `printStars(int n)` lautet

```
void printStars(int n);
```

Auch der in der vorherigen Aufgabe erstellte Enum-Typ sollte in der Header-Datei platziert werden.

Fertig – die Ausgabe des Programms sollte sich nicht verändert haben.

### Hinweise

- Sourcdateien tragen in der Regel die Endung `.cpp`, Headerdateien `.h` oder `.hpp`.
- Denke daran, auch in `functions.cpp` den Header `iostream` einzubinden, falls du dort Ein- und Ausgaben verwenden willst (`#include<iostream>`).
- Beachte, dass es zwei verschiedene Möglichkeiten gibt, eine Header-Datei einzubinden – per `#include <Bibliothekensname>` sowie per `#include "Dateiname"`. Bei der ersten Variante sucht der Compiler nur in den Standard-Include-Verzeichnissen, während bei der zweiten Variante auch (zunächst) die Projektordner durchsucht werden. Somit eignet sich die erste Schreibweise für System-Header (wie die der Standardbibliothek) und die zweite für projektspezifische Header.
- Anstelle der Include-Guards kannst du auch die Präprozessor Direktive `#pragma once` verwenden. Diese ist zwar nicht standardisiert, wird aber von den meisten Compilern unterstützt.

---

## 2e) Dokumentation (optional)

---

Diese Aufgabe dient der Vertiefung deines Wissens in C++ und ist nicht notwendig, um die Klausur zu bestehen.

Für die Lesbarkeit eines Programms ist eine ausführliche Dokumentation des Programmcodes essentiell. Dazu wirst du in dieser Aufgabe den Header `functions.hpp` mit Kommentaren versehen, die das Tool *Doxygen*<sup>2</sup> interpretieren kann.

Damit Doxygen deine Kommentare erkennt, muss ein spezielles Format eingehalten werden.

- Kommentare müssen vor den jeweiligen zu kommentierenden Elementen (z. B. Funktionen, Klassen) stehen.
- Mehrzeilige Kommentare müssen den folgenden Stil einhalten und mit einem Doppelstern beginnen (`/**`):

```
/**
 * Comment content
 */
```

Außerdem müssen bestimmte Befehle in den Kommentaren verwendet werden, die Doxygen bei der Dokumentationsgenerierung verwenden kann. Eine Liste aller Doxygen-Befehle findest du bei Interesse unter <https://www.doxygen.nl/manual/commands.html>.<sup>3</sup> Diese Kommandos sind die folgenden:

---

<sup>2</sup>Doxygen-Projektseite: <http://www.doxygen.nl/>

<sup>3</sup>Man kann seinen Kommentar auch speziell formatieren und so diese Kommandos teilweise weglassen. Beispiele unter <https://www.doxygen.nl/manual/docblocks.html#docexamples>

@file Dateiname	Damit Doxygen das komplette File parst.
@brief KurzeBeschreibung	Einzeilige Beschreibung des zu dokumentierenden Elements.
@author AutorenName	Name des Autors des zu dokumentierenden Elements.
@param Parametername Beschreibung	Je Funktionsparameter eine Zeile, die seinen Zweck erläutert.
@return Beschreibung	Kurze Beschreibung der Rückgabe.

Da das Dokumentieren der Datei nicht vor der Datei passieren kann (wo sollte das sein?), geschieht es deshalb direkt nach den Präprozessor-Direktiven.

Deine Aufgabe ist es nun, den von dir erstellten Code sorgfältig zu dokumentieren. Die Dokumentation geschieht dabei in der `.h` oder `.hpp` Datei. Hier ein kleines Beispiel dazu, damit du eine Vorstellung davon bekommst, wie das ganze am Ende auszusehen hat.

```
#ifndef TESTING_HPP_
#define TESTING_HPP_

/**
 * @file testing.hpp
 * @author Your Name
 * @brief Just a showcase hpp file to demonstrate doxygen comments
 * This is a very long description of the given file holding all the information one needs to get an
 * overview of the importance of this file.
 */

/**
 * @author Your Name
 * @brief A short description of this enum
 */
enum class Fruit {
    /**A delicious apple*/
    Apple,
    /**A delicious banana*/
    Banana,
    /**A delicious cherry*/
    Cherry
};

/**
 * @author Your Name
 * @brief A showcase function.
 * @param a Used for important stuff.
 * @return void
 */
void first_func(int a);

/**
 * @author Your Name
 * @brief A showcase function.
 * @param a Used for important stuff.
 * @param b Used for important stuff.
 * @param d Used for important stuff.
 * @return void
 */
void second_func(int a, char b, double d);

#endif /* TESTING_HPP_ */
```

**Erstellen der HTML-Dokumentation** Erstellen kannst du die Dokumentation am Ende über die Kommandozeile. Dafür öffnest du das Terminal mit `Shift + F12` und wechselst in das Verzeichnis, in dem dein Projekt liegt. Dort gibst du `doxygen -g` ein, was dir eine vorgefertigte Konfigurationsdatei für Doxygen generiert. Mit dem Befehl `doxygen` kannst du dir jetzt die fertige Dokumentation generieren lassen. Diese befindet sich nun im Verzeichnis `html` in der Datei `index.html`. Öffnest du diese Datei per Doppelklick, findest du unter dem Menüpunkt `Files` die von dir dokumentierten Dateien.



---

## 2f) Eingabe

---

**Eingabe der Breite** Erweitere das Programm um eine Eingabeaufforderung zur Bestimmung der Breite des auszugebenden Musters. Die Breite soll dabei eine im Programmcode vorgegebene maximale Breite (z. B. 80 Zeichen) nicht überschreiten dürfen. Gib gegebenenfalls eine Fehlermeldung aus und frage den Benutzer erneut nach der Breite des Musters. Verwende zum Einlesen `std::cin` und `operator>>` wie in folgendem Beispiel.

```
int x;
std::cin >> x; // Type, e.g., 174 and press ENTER.
// Now, x contains the entered number.
std::cout << x << std::endl.
```

**Eingabe der Richtung** In c) hast du einen enum-Typen definiert, der die Richtungen definiert, in die das Muster ausgegeben wird. Füge nun eine weitere Eingabeaufforderung hinzu, die vom Nutzer die gewünschte Richtung erfragt (beispielsweise 0 für `Left` und 1 für `Right`). Gibt der Nutzer eine ungültige Richtung ein, kannst du eine Fehlermeldung ausgeben oder erneut die Eingabe abfragen.

Erstelle auch für diesen Aufgabenteil eine eigene Funktion in `functions.cpp`.

---

## 2g) Fortlaufendes Alphabet ausgeben

---

Statt eines einzelnen Zeichens soll nun das fortlaufende Alphabet ausgegeben werden. Sobald das Ende des Alphabets erreicht wurde, beginnt die Ausgabe erneut bei a. Beispiel:

```
abc
de
f
gh
ijk
```

Implementiere dazu eine Funktion `char nextChar()`. Diese soll bei jedem Aufruf das nächste auszugebende Zeichen vom Typ `char` zurückgeben, beginnend bei 'a'. Dazu muss sich `nextChar()` intern das aktuelle Zeichen merken. Dies kann durch die Verwendung von statischen Variablen erreicht werden. Diese behalten ihren aktuellen Wert auch nach Verlassen der Funktion. Das heißt, wenn `nextChar()` das nächste Mal aufgerufen wird, steht der Wert des vorherigen Zeichens noch zur Verfügung. Eine statische Variable `c` wird wie folgt deklariert:

```
static unsigned char c = 'a';
```

In diesem Fall wird die Variable `c` **einmalig vor dem ersten Aufruf** mit 'a' initialisiert und kann später beliebig verändert werden.

### Hinweise

- Der Datentyp `char` kann wie eine Zahl verwendet werden, d. h. man kann die Modulooperation `%` verwenden um am Ende des Alphabets wieder zu 'a' umzubrechen.

---

## 2h) Namensräume

---

Ein Namensraum (engl. namespace) dient dazu, Namenskonflikte zu vermeiden. Erweitere dazu das Programm, indem du im Header die Funktionsprototypen wie folgt in einen namespace setzt.

```
namespace fun {
    // function prototypes ...
}
```

Denke daran, dass du die Namen der Funktionen in der Sourcedatei noch anpassen musst, indem du vor jede Funktion den gewählten `namespace`-Namen gefolgt von zwei Doppelpunkten setzt. Genauso muss der Namensraum auch vor jeden Aufruf der Funktion gesetzt werden.

```
void fun::print_star(int n) {  
    // ...  
}
```

Vergisst man, den Namensraum in der Sourcedatei vor den Funktionsdefinitionen anzugeben, findet der Linker keine Implementation zu der im Header deklarierten Funktion. Weiterhin stünde diese Funktion nicht mehr im Bezug zum Header und könnte nur noch lokal verwendet werden (`print_star(int n)` und `fun::print_star(int n)` sind zwei unterschiedliche Funktionen!).

Für mehrere Funktionsdefinitionen in der Sourcedatei ist es außerdem üblich, genauso wie im Header einen `namespace`-Block anstatt des vorangestellten `namespace` zu verwenden:

```
namespace fun {  
    void print_star(int n) {  
        // ...  
    }  
    void some_other_function(int n) {  
        // ...  
    }  
}
```

Falls man seine Funktionen noch weiter unterteilen möchte, kann man Namensräume auch schachteln. Hierzu definiert man wie oben einen weiteren Namensraum mit `namespace` in einer bereits vorhandenen Namensraum-Instanz.

```
namespace fun {  
    namespace ny {  
        // function prototypes ...  
    }  
}
```

In der Sourcedatei folgt dann nach dem ersten Namensraum (hier `fun`) der geschachtelte Namensraum `ny`. Die verschiedenen Ebenen der Namensräume werden durch zwei Doppelpunkte (den sogenannten Scope-Resolution-Operator) voneinander getrennt. Danach können die Funktionen in dem Namensraum `ny` verwendet werden, indem man diese ebenfalls mit dem Scope-Resolution-Operator an die Namensraum-Hierarchie anhängt.

```
void fun::ny::print_star(int n) {  
    // ...  
}
```

In diesem Projekt wird dies nicht notwendig sein, da die Anzahl der definierten Funktionen überschaubar ist, aber trotzdem empfehlen wir dir, es auszuprobieren.

## Hinweise

- Du kannst `using namespace fun;` verwenden, um diesen Namensraum zu importieren (vergleichbar mit `static import` in Java). Allerdings kann es dabei leichter zu Namenskollisionen zwischen Elementen der verschiedenen Namensräume kommen. Deshalb sollte der Befehl `using namespace` mit Bedacht verwendet werden.
- Bei dem geschachtelten Namensraum ist entsprechend `using namespace fun::ny;` zu verwenden um den Namensraum zu importieren.

---

## 2i) Structs

---

Structs sind Gruppen von Elementen, die unter einem Namen zusammengefasst sind. Diese Datenelemente, die als members bezeichnet werden, können unterschiedliche Typen und unterschiedliche Längen haben. Datenstrukturen können in C++ mit der folgenden Syntax deklariert werden:

```
struct struct_name {
    member_typ1 member_name1;
    member_typ2 member_name2;
    member_typ3 member_name3;
    ...
};

struct_name struct1;

struct1.member_name1 = ...;
struct1.member_name2 = ...;
```

Innerhalb der geschweiften Klammern befindet sich eine Liste mit den Datenmitgliedern, die jeweils mit einem Typ und einem gültigen Namen angegeben werden.

Ein Struct umfasst also mehrere zusammengehörige Elemente, die in einem bestimmten Kontext zusammengebunden werden müssen. Ein solcher Kontext kann die Übergabe von Argumenten an eine Funktion sein:

```
struct Point {
    double x;
    double y;
};

void distance(Point p1, Point p2);
```

Obwohl sich es bei einer `struct` im Grunde nur um ein „Bündel“ handelt, kann damit die Abstraktionsebene effektiv erhöht werden, um den Code zu verbessern: Im obigen Beispiel erwartet die Funktion `distance` Punkte statt Doubles. Obendrein hat das Struct auch noch den Vorteil der logischen Gruppierung.

Erstelle eine Struktur zum Speichern von Produkten eines Markts. Die Produkte werden durch das Gewicht als `int` und den Preis als `float` definiert.

Deklariere drei Produkte: Banane, Apfel und Wassermelone mit den Gewichten 200, 150 bzw. 1000 g und den Preisen 1,5, 0,75 bzw. 4,0 Euro. Zeige mit `std::cout` die Werte für die Banane:

```
Gewicht einer Banane: 200g
Preis einer Banane: 1.5 EUR
```

### Hinweise

- Die Codebeispiele oben zeigen die Verwendung einer `struct` in C++. In C ist darauf zu achten, dass bei der Deklaration einer Variable ebenfalls das `struct`-Keyword vorangestellt werden muss. Zum Beispiel wie folgt:

```
struct Point {
    double x;
    double y;
};

void distance(struct Point p1, struct Point p2) {
    struct Point p = p1;
    // ...
}
```

Gleiches gilt auch für `enums`.

---

### Aufgabe 3: [G] Klassen

---

Für diese Aufgabe findest du im Moodle-Kurs einen Lösungsvorschlag in Form eines Zip-Archivs. Die Lösung ist dort im Unterordner `/classes` hinterlegt. Ziel dieser Aufgabe ist es, die schon bearbeitete Aufgabe 2g) objektorientiert zu lösen. Schreibe hierfür manuell eine Klasse, die das aktuelle Zeichen als Attribut enthält und durch Methoden ausgelesen und inkrementiert werden kann.

#### Hinweise

- Verwende in dieser Aufgabe noch **nicht** den Klassengenerator (**Rechtsklick auf den Ordner `src/` → New C++ class...**) von CLion! Es ist besser, wenn man zumindest einmal eine Klasse und ihren Header selbst angelegt hat, da in der Klausur kein Klassengenerator zur Verfügung steht. :-)

---

#### 3a) Definition

---

Eine Klasse wird üblicherweise analog zu der vorherigen Aufgabe in Schnittstelle (Headerdatei) und Implementation (Sourcedatei) aufgeteilt. Die Struktur der Klasse mit allen Attributen und Funktionsprototypen wird im Header beschrieben, während die Sourcedatei nur die Implementation der Funktionen und Initialisierungen statischer Variablen enthält.<sup>4</sup> Standardmäßig sind alle Elemente einer Klasse `privat`.<sup>5</sup> Im Gegensatz zu Java werden in C++ die Access-Modifier `public/private/protected` nicht bei jedem Element einzeln, sondern blockweise angegeben.

```
class ClassName {  
    // private members ...  
public:  
    // public members ...  
private:  
    // private members ...  
}; // semicolon!
```

Erzeuge einen Header `CharGenerator.hpp` und erstelle den Klassenrumpf der Klasse `CharGenerator`. Füge der Klasse das `private` Attribut `char nextChar` hinzu, in dem das als nächstes auszugebende Zeichen gespeichert wird und einen `public` Konstruktorprototypen `CharGenerator()`, der `nextChar` auf `'a'` initialisieren soll. Füge noch einen `public` Funktionsprototypen `char generateNextChar()` hinzu, welcher das nächste auszugebende Zeichen zurückgeben soll.

#### Hinweise

- Ein Konstruktor wird als eine Funktion ohne Rückgabetyt deklariert, die den gleichen Namen wie die Klasse hat, und beliebige Parameter beinhalten kann.
- Es bietet sich an, auch hier den Programmcode zu dokumentieren. Das funktioniert wieder sehr ähnlich wie in Aufgabe e), nur tauschst du den Tag `@file` gegen `@class` aus und platzierst die Dokumentation vor der Definition der Klasse. Während der Bearbeitung der weiteren Aufgaben kannst du die Dokumentation anpassen und entdecken, wie `doxygen` deine Kommentare in eine fertige Dokumentation umsetzt.

---

#### 3b) Implementation

---

Wie bei der Verwendung von `namespace` muss der Scope der Klasse (der Klassenname) in der Sourcedatei vor jeder Elementbezeichnung (Konstruktor, Funktion, ...) durch zwei Doppelpunkte (den Scope-Resolution-Operator) getrennt angegeben werden.

---

<sup>4</sup>Es dürfen die Member-Funktionen auch direkt im Header definiert werden. Dies ist für sehr kurze Funktionen, wie z. B. Getter und Setter sinnvoll, um dem Compiler das Inlining zu ermöglichen.

<sup>5</sup>Dies ist tatsächlich auch der einzige Unterschied bei C++ zwischen einer `class` und einer `struct`. Bei letzterer sind die Elemente standardmäßig `public`.

```
void ClassName::functionName() {  
    // function implementation ...  
}
```

Um Attribute zu initialisieren, wird üblicherweise eine sogenannte Initialisierungsliste im Konstruktor verwendet, da diese vor dem Eintritt in den Konstruktorrumpf aufgerufen wird. Die Initialisierungsliste wird durch einen Doppelpunkt zwischen der schließenden Klammer der Parameterliste und der öffnenden geschweiften Klammer des Rumpfes eingeleitet, und bildet eine mit Komma separierte Liste von Attributnamen und ihren Initialisierungsargumenten in geschweiften Klammern.

```
ClassName::ClassName() : // initializer list:  
    attributeOne(initialValueOne),  
    attributeTwo(initialValueTwo) {  
    // constructor body  
}
```

Erzeuge eine Sourcedatei `CharGenerator.cpp` für die Implementation der Klasse und binde die `CharGenerator.hpp` ein. Implementiere den Konstruktor, indem du `nextChar` mit `'a'` in der Initialisierungsliste initialisierst. Implementiere zudem `generateNextChar()`, indem du `nextChar` zurückgibst.

### Hinweise

- Die Reihenfolge der Initialisierungsliste sollte der Deklarationsreihenfolge der Attribute entsprechen. (Unabhängig davon, welche Reihenfolge hier verwendet wird, erfolgt die tatsächliche Initialisierung in der Deklarationsreihenfolge der Attribute.)
- Konstanten **müssen** in der Initialisierungsliste zugewiesen werden, damit diese zur Laufzeit bekannt sind.

---

### 3c) Instantiierung

---

Erzeuge wie aus den vorherigen Aufgaben bekannt eine `main.cpp` mit einer `main()`-Funktion in der du ein `CharGenerator`-Objekt erzeugst und `generateNextChar()` mehrfach aufrufst und ausgibst.

```
CharGenerator charGen;  
char next = charGen.generateNextChar();  
std::cout << next << '\n';
```

Überprüfe das Ergebnis über die Konsole oder den Debugger.

### Hinweise

- Um ein Objekt zu erzeugen, muss hier, anders als in Java, kein `new` verwendet werden. Im Themenbereich „Speicherverwaltung“ des Praktikums gehen wir näher darauf ein.

---

### 3d) Default-Parameter

---

Damit man nicht immer das Startzeichen angeben muss, kann man einen Default-Wert für einen Parameter angeben. Beim Aufruf kann dieser Parameter dann weggelassen werden. Hierzu wird dem Parameter im Prototypen (im Header) ein Wert zugewiesen, ohne die Implementation zu ändern.

```
class CharGenerator {  
public:  
    CharGenerator(char initialChar = 'a');  
    //...  
};
```

Erweitere den Konstruktor um einen Parameter `char initialChar`, welcher defaultmäßig 'a' ist und ändere die Initialisierung von `nextChar`, damit dieser mit dem übergebenen Parameter gestartet wird.

Teste deine Implementierung sowohl mit als auch ohne Angabe des Startzeichens. Um ein Startzeichen anzugeben, lege das Objekt wie folgt an:

```
CharGenerator charGen('x');
```

### Hinweise

- Bei der Definition eines Default-Parameters müssen für alle nachfolgenden Parameter ebenfalls mit Default-Werten angegeben werden, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

---

### 3e) PatternPrinter

---

Implementiere folgende Klasse gemäß der Kommentare neben den Methodenprototypen.

```
class PatternPrinter {
    CharGenerator charGen; // (private) member variable ("attribute")

public:
    PatternPrinter();
    void printPattern(); // read width and print chars in a pattern

private:
    void printNChars(int n); // print n characters to the console
    int readWidth();        // read width (user input)
};
```

Teste deine Implementation, indem du ein `PatternPrinter`-Objekt anlegst und `printPattern()` darauf aufrufst.

### Hinweise

- Ohne eine Initialisierungsliste wird `charGenerator` mit dem Default-Parameter initialisiert. Um ein eigenes Startzeichen anzugeben, muss eine Initialisierungsliste erstellt und `charGenerator` mit dem entsprechenden Argument initialisiert werden.

---

## Aufgabe 4: [G] Überladen von Operatoren

---

Für diese Aufgabe findest du im Moodle-Kurs einen Lösungsvorschlag in Form eines Zip-Archivs. Die Lösung ist dort im Unterordner `/operator_overloading` hinterlegt. In C++ besteht die Möglichkeit, Operatoren wie `+` (`operator+`), `*` (`operator*`), ... zu überladen. Man kann selber spezifizieren, was beim Verknüpfen von Objekten mit einem Operator geschehen soll, um zum Beispiel den Quellcode übersichtlicher zu gestalten. Du hast bereits das Objekt `std::cout` der Klasse `std::ostream` kennengelernt, welche den `<<`-Operator überlädt, um Ausgaben von `std::string`, `int`, ... komfortabel zu tätigen. In dieser Aufgabe sollst du eine eigene Vektor-Klasse schreiben und einige Operatoren überladen.

### Hinweise

- Ausführliche Hinweise zum Überladen von Operatoren findest du hier: <http://en.cppreference.com/w/cpp/language/operators>.

---

### 4a) Konstruktor und Destruktor

---

Implementiere die folgende Klasse. Füge jedem Konstruktor und Destruktor eine Ausgabe auf der Konsole hinzu, um beim Programmlauf den Lebenszyklus der Objekte nachvollziehen zu können.

```
class Vector3 {
    double a_, b_, c_; // vector components

public:
    Vector3(); // initialize vector with zero
    Vector3(double a, double b, double c); // initialize vector with a, b, c
    Vector3(const Vector3& other); // copy constructor: copy a vector
    ~Vector3(); // destructor: destroy the vector
};
```

Der Copy-Konstruktor wird aufgerufen, wenn das Objekt kopiert werden soll, z. B. für eine Call-by-Value Parameterübergabe. Jeder Copy-Konstruktor benötigt eine Referenz auf ein Objekt vom gleichen Typ wie die Klasse selbst als Parameter. Sinnvollerweise wird noch `const` vor oder nach der Typbezeichnung eingefügt (aber vor `&`), da typischerweise das Ursprungsobjekt nicht verändert wird.

Der Destruktor wird aufgerufen, sobald die Lebenszeit eines Objekts endet. Er wird verwendet, um Ressourcen, die das Objekt besitzt, freizugeben. Die Syntax des Prototypen lautet

```
~ClassName();
```

und die Implementation entsprechend

```
ClassName::~~ClassName() {
    /* destructor implementation ... */
}
```

### Hinweise

- Es darf eine beliebige Anzahl an Konstruktoren mit verschiedenen Parametersätzen existieren.
- Der Compiler wird automatisch einen `public` Destruktor und `public` Copy-Konstruktor erzeugen, falls sie nicht *deklariert* wurden. Ebenso wird ein `public` Defaultkonstruktor (keine Argumente) automatisch vom Compiler generiert, falls überhaupt keine Konstruktoren deklariert wurden.

Falls du sie jedoch *deklarierst*, musst du auch eine Implementierung angeben.

- Würden beim Copy-Konstruktor `other` by-Value übergeben werden, müsste eine Kopie von `other` angelegt werden. Dazu würde der Copy-Konstruktor aufgerufen, was zu einer unendlichen Rekursion führt, bis der Stack seine maximale Größe überschreitet und das Programm abstürzt.

---

## 4b) Vektoraddition, Vektorsubtraktion und Skalarprodukt

---

Erweitere die Klasse um folgende `public` Funktionen, um Vektoren durch  $v1 + v2$ ,  $v1 - v2$  und  $v1 * v2$  addieren/subtrahieren und das Skalarprodukt bilden zu können, indem die Operatoren `+`, `-` und `*` überladen werden.

```
Vector3 operator+(Vector3 rhs) const; // add two vectors component-by-component
Vector3 operator-(Vector3 rhs) const; // subtract two vectors component-by-component
double operator*(Vector3 rhs) const; // determine the dot product of two vectors
```

Innerhalb der Methode kannst du durch `a_`, `b_` und `c_` (oder auch durch `this->a_`, `this->b_` und `this->c_`) auf eigene Attribute und über `rhs.a_`, `rhs.b_` und `rhs.c_` auf Attribute der rechten Seite zugreifen. Das `const` nach der Parameterliste der Funktionen bedeutet, dass diese Funktion keine Änderungen an den eigenen Attributen vornehmen darf.

Denke daran, bei der Implementation der Klassen den Scope der Klasse in der Sourcedatei vor jeder Elementbezeichnung durch zwei Doppelpunkte getrennt (den bereits bekannten Scope-Resolution-Operator) anzugeben.

```
Vector3 Vector3::operator+(Vector3 rhs) const {
    /* function implementation ...*/
}
Vector3 Vector3::operator-(Vector3 rhs) const {
    /* function implementation ...*/
}
double Vector3::operator*(Vector3 rhs) const {
    /* function implementation ...*/
}
```

### Hinweise

- Der Parameter `rhs` steht für die rechte Seite („right-hand-side“) des jeweiligen Operators. Dadurch, dass der Operator als Member der Klasse deklariert wurde, nimmt die aktuelle Instanz hierbei automatisch die linke Seite der Operation („left-hand side“) an.
- Der Rückgabetypp eines Skalarprodukts (dot product) ist kein `Vector3` sondern ein Skalar (`double`)!

---

## 4c) Ausgabe

---

Überlade den `operator<<` zur Ausgabe eines Vektors mit der gewohnten `std::cout << ...` Syntax, indem du den folgenden Funktionsprototypen **außerhalb** der Klassendefinition setzt

```
std::ostream& operator<<(std::ostream& out, Vector3 rhs);
```

und innerhalb der Sourcedatei wie folgt implementierst.

```
std::ostream& operator<<(std::ostream& out, Vector3 rhs) {
    out << ...;
    return out;
}
```

Da der `operator<<` außerhalb der Klasse `Vector3` liegt, hat dieser keinen Zugriff auf die privaten Member der Klasse. Du hast zwei Möglichkeiten, Zugriff auf diese zu erlangen: per Getter und per `friend`-Deklaration.

**Getter** Definiere die folgenden Gettermethoden, die die Werte für die `private` Attribute `a_`, `b_` und `c_` zurückgeben:

```
double getA() const; // get the first component
double getB() const; // get the second component
double getC() const; // get the third component
```



---

**friend** Füge die folgende Zeile am Ende der Klasse `Vector` hinzu:

```
friend std::ostream& operator<<(std::ostream&, Vector3);
```

Von nun an kann die entsprechende Funktion auf alle privaten Member der Klasse `Vector` zugreifen, was insbesondere praktisch ist, falls die Klasse verändert werden soll.

## Hinweise

- Denke daran, den Header `iostream` einzubinden.
- Diesmal musste die Überladung **außerhalb** der `Vektor3`-Klasse definiert werden, weil das `Vektor3`-Objekt auf der rechten Seite der Operation steht. Als linke Seite wird hierbei ein `std::ostream`-Objekt (wie z. B. `std::cout`) erwartet, um Ausgabeketten `std::cout << ... << ...` zu ermöglichen. Hierzu muss das Ausgabeobjekt auch zurückgegeben werden. Damit das `std::ostream`-Objekt aber nicht jedes Mal kopiert wird, wird es als Referenz & durchgereicht.
- Anstatt Getter und Setter für `private` Attribute zu schreiben, kann man auch einer Klasse oder Funktion vollen Zugriff mit Hilfe des Schlüsselworts `friend` erlauben. In der nächsten Übung wird hierauf noch einmal eingegangen.

---

## 4d) Testen

---

Teste deine bisher definierten Methoden und Funktionen mithilfe der bereitgestellten Tests. Die Catch2-Tests setzen hierbei voraus, dass alle Operatorenüberladungen implementiert sind. Probiere auch Kombinationen von verschiedenen Operatoren aus und beobachte das Ergebnis. Schreibe auch eine einfache Funktion, die Vektoren als Parameter nimmt. Wie du siehst, werden sehr viele `Vector3`-Objekte erstellt, kopiert und gelöscht. Dies liegt daran, dass die Objekte immer per Call-by-Value übergeben und dabei kopiert werden. Wie dies vermieden werden kann, siehst du im Themenbereich „Speicherverwaltung“.



Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie <http://creativecommons.org/licenses/by-nc-nd/4.0/> oder wenden Sie sich brieflich an Creative Commons, Postfach 1866, Mountain View, California, 94042 USA.