Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 1. Tag

Einführung

Für alle Übungen des C/C++ Praktikums wird CodeLite als IDE verwendet. Als Compiler kommt Clang zum Einsatz. Eine Einführung in den Umgang mit CodeLite findet am ersten Praktikumstag statt.
Die Materialien zur Vorlesung und Übung sind in zwei Git-Repositories zu finden. Diese beinhalten zum einen die Folien und die Programmbeispiele aus der Vorlesung und zum anderen diese Übungsblätter, Vorlagen für die letzten beiden Tage des Praktikums und alle Lösungen zu den Übungsaufgaben.

- **Vorlesung:** `https://github.com/Echtzeitsysteme/tud-cpp-lecture`
- **Übungen/Git- und CodeLite-Einführung:** `https://github.com/Echtzeitsysteme/tud-cpp-exercises`

Falls du mit dem Übungsblatt für den dafür vorgesehenen Tag innerhalb der Präsenzzeit fertig werden solltest kannst du außerdem gerne mit dem nächsten Übungsblatt anfangen.

Hinweise

- Bei Fragen und Problemen aktiv um Hilfe bitten!
- Alle Lösungen enthalten ein Makefile und können entweder über die Kommandozeile mit Hilfe von `make` kompiliert werden, oder in CodeLite, indem du sie als Projekt importierst.
- Folgende Shortcuts könnten sich dir im Verlauf des Praktikums als nützlich erweisen:

Tastenkürzel	Befehl	Beschreibung
Ctrl+Space	Autocomplete	Anzeige von Vervollständigungshinweisen (z.B. nach `std::` oder `main`)
Alt+Shift+L	Rename local	Umbennenen von lokalen Variablen
Alt+Shift+H	Rename symbol	Umbennenen von Funktionen und Klassen
Ctrl+N	New	Anlegen neuer Datei
F12	Switch tab	Wechsel zwischen der Header- und der Implementierungsdatei
F7	Build	Startet den Buildprozess (Aufruf von Compiler und Linker)

Musterlösungs-/Microcontroller-Projekte in CodeLite importieren

Die angebotenen Musterlösungs-/Microcontroller-Projekte basieren auf Makefiles. Daher ist es wichtig, dass du sie entsprechend importierst: **Workspace -> Add an existing project** und dann zur entsprechenden `.project`-Datei navigieren.

Aufgabe 0 Hello World

Lege ein neues C++ Projekt an, indem du **Workspace → New project** im CodeLite Menü wählst und als Projekttyp **CPPP/C++ Projekt** auswählst. Wähle einen passenden Projektnamen, z.b. `Tag1_Aufgabe1`. Als Projektpfad sollte `/home/cppp/CPPP/Workspace` ausgewählt sein. Achte darauf, dass der Haken bei `Create the project under a separate directory` gesetzt ist.
Das Projekt enthält bereits eine Datei `src/main.cpp`. Erweitere die `main`-Funktion wie gezeigt und kompiliere das Projekt mit einem Klick auf das Build-Symbol (grünes Rechteck mit weißem Pfeil). Führe dann das Programm mit einem Klick auf das Zahnrad-Symbol aus.

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

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'* abbrechen. Der Rückgabetyp 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/` 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 CodeLite geworfen, wenn es nach dem Kompilieren das lauffähige Programm nicht findet. Das kann zwei Gründe haben:

- Der Kompiliervorgang ist gescheitert. Prüfe die `Console` 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 „`int main() {`“ beziehen!!

```
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.

Übungen für den 1. Tag

Aufgabe 1 C++ Grundlagen, Funktionen und Strukturierung

Für diese Aufgabe kannst du entweder das vorherige Programm weiter entwickeln oder genauso wie vorher ein neues Projekt anlegen.

Primitive Datentypen

Die primitiven Datentypen in C++ sind ähnlich 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. Alles ungleich 0 wird als `true` gewertet, 0 als `false`. Somit können Ganzzahlen direkt in Bedingungen ausgewertet werden.

Aufgabe 1.1 Größe von Datentypen

Die Größe von den verschiedenen Datentypen ist essentiell zu wissen, wenn man mit ihnen arbeiten möchte. Deshalb sollst du dir in dieser Aufgabe die Größe der folgenden Datentypen in Bits, wie auch deren minimalen und maximalen Wert ausgeben lassen.

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

Hinweise

- Zum Überprüfen der Größe von Datentypen kann man den `sizeof()` Operator[1] verwenden.
- Die C++ Klasse `std::numeric_limits`[2] bietet Funktionen sich minimale und maximale Werte von Datentypen ausgeben zu lassen. Einbinden lässt sich diese über den Header `limits`.

Aufgabe 1.2 Sternenmuster mit Funktionen malen

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

```
*****
```

Platziere die Funktion **vor** der `main`, da sie sonst von dort aus nicht aufgerufen werden kann. Benutze die erstellte Funktion `printStars(int n)`, um eine weitere Funktion zu schreiben, die eine Figur wie unten dargestellt ausgibt. Verwende hierzu Schleifen.

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

[1] http://en.cppreference.com/w/c/language/sizeof

[2] http://en.cppreference.com/w/cpp/types/numeric_limits

Übungen für den 1. Tag

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++ `std::cout` zur Verfügung, welches den String zu dem Standard Output Stream weitergibt. Diesen Output Stream kann man mit dem Manipulator `std::endl` zu einem Zeilenumbruch zwingen.

Aufgabe 1.3 Auslagern der Datei

Erstelle eine neue Header-Datei **`functions.h`** und eine neue Sourcedatei **`functions.cpp`**. Klicke hierzu mit der **rechten Maustaste** auf den Ordner **`src`** und wähle **Add a New File**, wähle den Dateitypen `Header File` und gebe der Datei den Namen `functions`. Bestätige den Dialog mit **OK**. Das ganze wiederholst du mit dem Dateitypen `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.h` in beide Sourcedateien (`functions.cpp` und `main.cpp`) ein, indem du

```
#include "functions.h"
```

verwendest. **Verschiebe** deine beiden Funktionen nach `functions.cpp`.
Schreibe nun in `functions.h` **Funktionsprototypen** für die beiden Funktionen aus der vorherigen Aufgabe. Funktionsprototypen dienen dazu, dem Compiler mitzuteilen, dass eine Funktion mit bestimmtem Namen, Parametern und Rückgabewert existiert. Ein Prototyp ist im Wesentlichen eine mit `;` abgeschlossene Signatur der Funktion ohne Funktionsrumpf. Der Prototyp von `printStars(int n)` lautet

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

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

Hinweise

- Sourcedateien 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 <Bibliotheksname>` sowie per `#include "Dateiname"`. Bei der ersten Variante sucht der Compiler nur in den Include-Verzeichnissen der Compiler-Toolchain, während bei der zweiten Variante auch die Projektordner durchsucht werden. Somit eignet sich die erste Schreibweise für System-Header und die zweite für eigene, projektspezifische Header.
- Anstelle der Include Guards kannst du auch die Präprozessordirektive `#pragma once` verwenden. Diese ist zwar nicht standardisiert, wird aber von den meisten Compilern unterstützt.

Aufgabe 1.4 Dokumentation

Für die Lesbarkeit eines Programms ist eine ausführliche Dokumentation des Programmcodes essentiell. Damit du einen Einblick darin bekommt, wird es deine Aufgabe sein, deinen geschriebenen Code durchgehend zu kommentieren. Zum Erstellen der Dokumentation werden wir das Tool *Doxygen*[3] verwenden.
Damit Doxygen deine Kommentare erkennt, muss ein spezielles Format eingehalten werden.

- Kommentare müssen vor den jeweiligen zu kommentierenden Elementen (z.B. Funktionen) stehen.

[3] Doxygen-Link als Referenz: `http://www.doxygen.nl/`

- Mehrzeilige Kommentare müssen den folgenden Stil einhalten (beachte hierbei das zusätzliche * in der ersten Zeile)[4]

```
/**
 * Commment content
 */
```

Außerdem müssen bestimmte Kommandos[5] in den Kommentaren verwendet werden, die Doxygen bei der Dokumentationsgenerierung verwenden kann[6]. Diese Kommandos sind die folgenden

- `@file Dateiname` damit Doxygen den kompletten File parst.
- `@name Name` Name des zu dokumentierenden Elements.
- `@brief kurzeBeschreibung` einzeilige Beschreibung des Elements.
- `@author AuthorenNamen` für den Namen des Autoren.
- `@param Parametername Beschreibung` um Parameterübergaben bei Funktionen zu erklären.
- `@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.
Eure Aufgabe ist es nun, den von euch erstellten Code sorgfältig zu dokumentieren. Die Dokumentation geschieht dabei in der `.h` oder `.hpp` Datei.
Hier ein kleines Beispiel dazu, damit ihr eine Vorstellung davon bekommt, 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 need to
    get an overview of the importance of this file.
 */

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

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

[4] Es gibt noch andere Formate, aber wir werden hier in dem Praktikum den sogenannten JavaDoc-Style verwenden.
[5] Liste aller Doxygen Kommandos `http://www.stack.nl/~dimitri/doxygen/manual/commands.html`
[6] Man kann sein Kommentar auch speziell formatieren und so diese Kommandos teilweise weglassen. Um aber so ausführlich wie möglich zu sein, werden wir diese Kommandos verwenden. Beispiele unter `http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html#docexamples`

```
 * @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_ */
```

Letzendliches Erstellen der Dokumentation
Erstellen könnt ihr die Dokumentation am Ende über die Kommandozeile. Dafür öffnet ihr euer Terminal mit `STRG + ALT + t` und wechselt in das Verzeichnis in dem euer Projekt liegt (üblicherweise `cd CPPP/Workspace/NameEuresProjektes`). Dort gebt ihr `doxygen -g` ein, welches euch eine vorgefertigte Konfigurationsdatei `Doxygen` von `doxygen` erstellt, gefolgt von einem `doxygen`, was letztendlich die Dokumentation erstellt. Diese findet ihr in dem Ordner `html` unter der Datei `index.html`. Ihr erreicht die Datei ganz einfach, wenn ihr in der Terminal das Kommando `pcmanfm` eingebt, was euren Dateimanager öffnet. Auf der Webseite, die sich dann öffnet, findet ihr unter Files die von euch dokumentierte `functions.h,hpp`.

Aufgabe 1.5 Eingabe

Erweitere das Programm um eine Eingabeaufforderung zur Bestimmung der Breite der auszugebenden Figur. Die Breite soll dabei eine im Programmcode vorgegebene Grenze nicht überschreiten dürfen. Gib gegebenenfalls eine Fehlermeldung aus. 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.
```

Erstelle auch für diesen Aufgabenteil eine eigene Funktion und lagere diese nach `functions.cpp` aus.

Aufgabe 1.6 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 von 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 alten Wert beim Wiedereintritt in die Funktion. Ein statische Variable `c` wird mittels

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

deklariert. In diesem Fall wird die Variable `c` **einmalig zu Beginn des Programms** mit `'a'` initialisiert und kann später beliebig verändert werden.

Hinweise

- Der Datentyp `char` kann wie eine Zahl verwendet werden, d.h. man kann z.B. die Modulooperation `%` verwenden.

Aufgabe 1.7 Namensräume

Bibliotheken werden in einen eigenen Namensraum gekapselt, damit ihre Funktionen nicht mit gleichnamigen Funktionen in anderen Bibliotheken kollidieren. Erweitere dazu das Programm, indem du im Header die Funktionsprototypen wie folgt in einen `namespace` setzt.

```
namespace fun {
    // function prototypes ...
};  // semicolon!
```

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 vor jedem Aufruf der Funktion gesetzt werden.

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

Vergisst man, den Namensraum in der Sourcedatei anzugeben, findet der Linker keine Implementation zu der im Header definierten 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 unterschiedliche Funktionen!).

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

```
namespace fun {
    namespace ny{
    // function prototypes ...
    };  // semicolon!
};  // semicolon!
```

In der Sourcedatei folgt dann nach dem initialen Namensraum, hier `fun` gefolgt von dem Doppelpunktpaar, der geschachtelte Namensraum `ny` wieder gefolgt von zwei Doppelpunkten. Erst dann können die Funktionen in dem Namensraum `ny` verwendet werden.

```
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). Genauso wie in Java kann es hierdurch leichter zu Namenskollisionen kommen und sollte daher eher nicht verwendet werden.
- Bei dem geschachtelten Namensraum ist entsprechend `using namespace fun::ny;` zu verwenden um den Namensraum zu importieren.

Übungen für den 1. Tag

Aufgabe 2 Klassen

Ziel dieser Aufgabe ist es die vorherige Aufgabe 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 class...**) von CodeLite!

Aufgabe 2.1 Definition

Eine Klasse wird üblicherweise analog zu der vorherigen Aufgabe in Deklaration (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. Standardmäßig sind alle Elemente einer Klasse privat. Im Gegensatz zu Java werden in C++ die Access-Modifier `public`/`private`/`protected` nicht bei jedem Element einzeln sondern blockweise angegeben.

```
class ClassName {
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 eine `public` Funktionsprototypen `char generateNextChar()` hinzu, welche das nächste auszugebende Zeichen zurückgeben soll.

Hinweise

- Ein Konstruktor wird als eine Funktion ohne Rückgabetyp deklariert, die den gleichen Namen wie die Klasse hat, und beliebige Parameter beinhalten kann.

Aufgabe 2.2 Dokumentation von Klassen

Auch in dieser Aufgabe geht es wieder darum, euren Programmcode zu dokumentieren. Das funktioniert wieder sehr ähnlich wie in Aufgabe Aufgabe 1.4, nur tauscht ihr den Tag `@file` gegen `@class` aus und platziert eine dementsprechende Dokumentation vor der Definition der Klasse. Dies könnt ihr fortlaufend in der Aufgabe erfüllen und muss nicht direkt jetzt geschehen. Am Ende erstellt ihr euch wieder eine Dokumentation eurer Klassen und könnt so entdecken, wie `doxygen` eure Kommentare in Dokumentation umsetzt.

Aufgabe 2.3 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 getrennt angegeben werden.

```
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 geschweiften Klammer des Rumpfes eingeleitet, und bildet eine mit Komma separierte Liste von Attributnamen und ihren Initialisierungsargumenten in 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 entsprechen.
- Konstanten **müssen** in der Initialisierungsliste zugewiesen werden, damit diese zur Laufzeit bekannt sind.

Aufgabe 2.4 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 << std::endl;
```

Überprüfe das Ergebnis über die Konsole oder den Debugger.

Hinweise

- Um ein Objekt zu erzeugen, muss in C++ kein `new` verwendet werden (siehe dazu nächste Vorlesung).

Aufgabe 2.5 Default-Parameter

Damit man nicht immer das Startzeichen angeben muss, kann man sogenannte Default-Parameter angeben, der beim Aufruf weggelassen werden kann. 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 Implementation 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 Default-Werte angegeben werden, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

Aufgabe 2.6 PatternPrinter

Implementiere folgende Klasse.

```
class PatternPrinter {
public:
   PatternPrinter();
   void printPattern();       // read width and print chars in a pattern
private:
   CharGenerator charGen;
   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.

Übungen für den 1. Tag

Aufgabe 3 Operatorenüberladung

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

- Am Tag 4 soll dieser Vektor um weitere Funktionen erweitern werden. Falls du mit dieser Aufgabe bis dahin nicht fertig sein solltest, kannst du natürlich auf die Musterlösung zurückgreifen.
- Ausführliche Hinweise zum Überladen von Operatoren findest du hier: `http://en.cppreference.com/w/cpp/language/operators`.

Aufgabe 3.1 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 {
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
private:
   double a, b, c;                             // vector components
};
```

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 dürfen 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 sie jedoch *deklariert* wurden, 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.

Aufgabe 3.2 Vektoraddition, -subtraktion 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);     // add two vectors component-by-component
Vector3 operator-(Vector3 rhs);     // subtract two vectors component-by-component
double operator*(Vector3 rhs);      // determine the dot product of two vectors
```

Innerhalb der Methode kannst du durch `a`, `b` und `c` auf eigene Attribute und über `rhs.a`, `rhs.b` und `rhs.c` auf Attribute der rechten Seite zugreifen. Denke daran, bei der Implementation den Klassen den Scope der Klasse in der Sourcedatei vor jeder Elementbezeichnung durch zwei Doppelpunkte getrennt anzugeben.

```
Vector3 Vector3::operator+(Vector3 rhs) {/* function implementation ...*/}
Vector3 Vector3::operator-(Vector3 rhs) {/* function implementation ...*/}
double Vector3::operator*(Vector3 rhs)  {/* 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 Rückgabetyp eines Skalarprodukts (dot product) ist kein `Vector3` sondern ein Skalar (`double`)!

Aufgabe 3.3 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(); // get the first component
double getB(); // get the second component
double getC(); // 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.

Aufgabe 3.4 Testen

Teste deine bisher definierten Methoden und Funktionen. 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 nächsten Teil des Praktikums.