

Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Hinweis: Die Präfixe der Aufgaben beziehen sich auf die Themengebiete der Vorlesung. ([G] : Grundlagen; [S] : Speicherverwaltung/Memory; [O] : Objektorientierung; [F] : Fortgeschrittene Themen; [C] : (Embedded) C; [A] : Zusatzaufgaben zum Aufzugsimulator)

Inhaltsverzeichnis

Aufgabe 1	[G] Hello World	2
Aufgabe 2	[G] C++ Grundlagen, Funktionen und Strukturierung	5
Aufgabe 3	[G] Klassen	12
Aufgabe 4	[G] Operatorenüberladung	15
Aufgabe 5	[S] Zeiger und Referenzen Grundlagen	18
Aufgabe 6	[S] Arrays und Zeigerarithmetik	22
Aufgabe 7	[S] Verkettete Listen	26
Aufgabe 8	[S] Smart Pointers	31
Aufgabe 9	[O] Vererbung und Polymorphie	34
Aufgabe 10	[O] Pure-Virtual-Methoden	36
Aufgabe 11	[O] Mehrfachvererbung	37
Aufgabe 12	[O] Exceptions	39
Aufgabe 13	[F] Template Funktionen	41
Aufgabe 14	[F] Generische Vektor-Implementation	42
Aufgabe 15	[F] Generische Verkettete Liste	43
Aufgabe 16	[F] Standard-Container	44
Aufgabe 17	[F] Funktionales Programmieren	45
Aufgabe 18	[F] Callbacks	49
Aufgabe 19	[F] Eigene Arrays (optional)	54
Aufgabe 20	[F] Makefiles	55
Aufgabe 21	[C] Die Programmiersprache C im Vergleich zu C++	63
Aufgabe 22	[C] Testprogramm auf den Microcontroller laden (optional)	68
Aufgabe 23	[C] Taster abfragen (optional)	72
Aufgabe 24	[C] Display ansteuern (optional)	72
Aufgabe 25	[C] Joysticks abfragen (optional)	76
Aufgabe 26	[C] Touchscreen ansteuern (optional)	78
Aufgabe 27	[C] Eigenes Microcontroller-Projekt umsetzen (optional)	79
Aufgabe 28	[A] Aufzugsimulator – Teil 1 (optional)	82
Aufgabe 29	[A] Aufzugsimulator – Teil 2 (optional)	85
Aufgabe 30	[A] Aufzugsimulator – Teil 3 (optional)	89



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.

Aufgaben zu C/C++-Grundlagen

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 einem GitHub-Repository zu finden: <https://github.com/Echtzeitsysteme/tud-cpp>. Dieses beinhaltet 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.

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 zum C++-Teil 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	Umbenennen von lokalen Variablen
Alt+Shift+H	Rename symbol	Umbenennen 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 1 [G] 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*

Aufgaben zu C/C++-Grundlagen

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

- Einzelige 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 Kompilervorgang ist gescheitert. Prüfe die Console auf Fehler.
- Der Kompilervorgang 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!!

Aufgaben zu C/C++-Grundlagen

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

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.

Aufgaben zu C/C++-Grundlagen

Aufgabe 2 [G] 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 2.1 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()` Operator¹ 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 2.2 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 `printFigure(int n)` 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

Aufgaben zu C/C++-Grundlagen

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

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.
- Klassen beginnen mit einem Großbuchstaben. Funktionen und Variablen hingegen sollten mit Kleinbuchstaben beginnen.
- 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 2.3 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 definieren, 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
```

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

```
Fruit myFruit = apple;  
  
if(myFruit == banana) {  
    // Do something  
}
```

Aufgaben zu C/C++-Grundlagen

Deine Aufgabe ist es jetzt, das Programm um einen enum-Typen Direction zu erweitern, der die Richtung (left oder 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 ganze soll am Ende jeweils folgendermaßen aussehen:

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

Aufgabe 2.4 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_ */
```

Bindet 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 Funktionen aus den beiden vorherigen Aufgaben. 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);
```

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

- 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

Aufgaben zu C/C++-Grundlagen

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 2.5 Dokumentation

Für die Lesbarkeit eines Programms ist eine ausführliche Dokumentation des Programmcodes essentiell. Damit du einen Einblick darin bekommst, 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.
- Mehrzeilige Kommentare müssen den folgenden Stil einhalten (beachte hierbei das zusätzliche * in der ersten Zeile)⁴

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

Außerdem müssen bestimmte Kommandos⁵ in den Kommentaren verwendet werden, die Doxygen bei der Dokumentationsgenerierung verwenden kann⁶. Diese Kommandos sind die folgenden:

<code>@file Dateiname</code>	Damit Doxygen das komplette File parst.
<code>@name Name</code>	Name des zu dokumentierenden Elements.
<code>@brief KurzeBeschreibung</code>	Einzelige Beschreibung des zu dokumentierenden Elements.
<code>@author AutorenName</code>	Name des Autors des zu dokumentierenden Elements.
<code>@param Parametername Beschreibung</code>	Je Funktionsparameter eine Zeile, die seinen Zweck erläutert.
<code>@return Beschreibung</code>	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.
```

³ Doxygen-Link als Referenz: <http://www.doxygen.nl/>

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 seinen 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>

Aufgaben zu C/C++-Grundlagen

```
/*
 */
/***
 * @name enum Fruit
 * @author Your Name
 * @brief A short description of this enum
 */
enum Fruit {
    /**A delicious apple*/
    apple,
    /**A delicious banana*/
    banana,
    /**A delicious cherry*/
    cherry
};

/***
 * @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.
 * @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_ */
```

Letztdliches Erstellen der Dokumentation

Erstellen kannst du die Dokumentation am Ende über die Kommandozeile. Dafür öffnest du das Terminal mit STRG + ALT + t und wechselst in das Verzeichnis, in dem dein Projekt liegt (üblicherweise cd ~/CPPP-Workspace/NameDeinesProjektes). 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 dokumentierte functions.h.hpp.

Aufgabe 2.6 Eingabe

Eingabe der Breite

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.

Aufgaben zu C/C++-Grundlagen

```
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 Aufgabe 2.3 hast du einen enum-Typen definiert, der die Richtungen definiert, in die das Pattern ausgegeben werden kann. Füge nun eine weitere Eingabeaufforderung hinzu, die die vom Nutzer gewünschte Richtung abfragt (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 und lagere diese nach `functions.cpp` aus.

Aufgabe 2.7 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 mittels

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

deklariert. In diesem Fall wird die Variable `c` **einmalig beim 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 z.B. die Modulooperation `%` verwenden.

Aufgabe 2.8 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. Achte auch auf das Semikolon nach der schließenden Klammer.

```
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 auch vor jeden Aufruf der Funktion gesetzt werden.

Aufgaben zu C/C++-Grundlagen

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

Vergisst man, den Namensraum in der Sourcedatei vor den Funktionsaufrufen anzugeben, findet der Linker keine Implementation zu der im Header definierten Funktion. Weiterhin würde 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!).

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. Wichtig ist auch hier wieder das Semikolon nach jeder schließenden Klammer der Namespace-Definition.

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

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.

Aufgaben zu C/C++-Grundlagen

Aufgabe 3 [G] 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 3.1 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 einen **public** Funktionsprototypen **char** generateNextChar() hinzu, welcher 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 3.2 Dokumentation von Klassen

Auch in dieser Aufgabe geht es wieder darum, deinen Programmcode zu dokumentieren. Das funktioniert wieder sehr ähnlich wie in Aufgabe Aufgabe 2.5, nur tauschst du den Tag **@file** gegen **@class** aus und platziert die Dokumentation vor der Definition der Klasse. Dies kannst du fortlaufend in der Aufgabe erfüllen, es muss nicht direkt jetzt geschehen. Am Ende erstellst du dir wieder eine Dokumentation der Klassen und kannst so entdecken, wie doxygen deine Kommentare in eine fertige Dokumentation umsetzt.

Aufgabe 3.3 Implementation

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

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

Aufgaben zu C/C++-Grundlagen

Um Attribute zu initialisieren, wird üblicherweise eine sogenannte Initialisierungsliste im Konstruktor verwendet, da diese vor dem Eintritt in den Konstruktorkrumpf 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 Initialisierungswerten 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 3.4 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 3.5 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 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.

Aufgaben zu C/C++-Grundlagen

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 mit Default-Werten angegeben werden, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

Aufgabe 3.6 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.

Aufgaben zu C/C++-Grundlagen

Aufgabe 4 [G] 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 4.1 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();
    Vector3(double a, double b, double c);           // initialize vector with zero
    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 Typebezeichnung 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.

Aufgaben zu C/C++-Grundlagen

- 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 4.2 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 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) {/* 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 („left-hand side“) an.
- Der Rückgabetyp eines Skalarprodukts (dot product) ist kein `Vector3` sondern ein Skalar (**double**)!

Aufgabe 4.3 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.

Aufgaben zu C/C++-Grundlagen

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 `Vector3`-Klasse definiert werden, weil das `Vector3`-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 4.4 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.

Aufgaben zur Speicherverwaltung in C++

Aufgabe 5 [S] Zeiger und Referenzen Grundlagen

In dieser Aufgabe sollst du den Umgang mit Zeigern (*Pointer*) und Referenzen erlernen. Diese erlauben es zum Beispiel Werte zwischen Funktionen auszutauschen, ohne eine Kopie der zu übermittelnden Daten zu erzeugen. Anstelle dessen kann ein (vergleichsweise kleiner) Zeiger auf einen Speicherbereich übergeben werden. Alternativ kann auch eine Referenz auf eine Variable übergeben werden, welche intern ähnlich wie ein Zeiger gehandhabt wird.

Aufgabe 5.1 Experimente

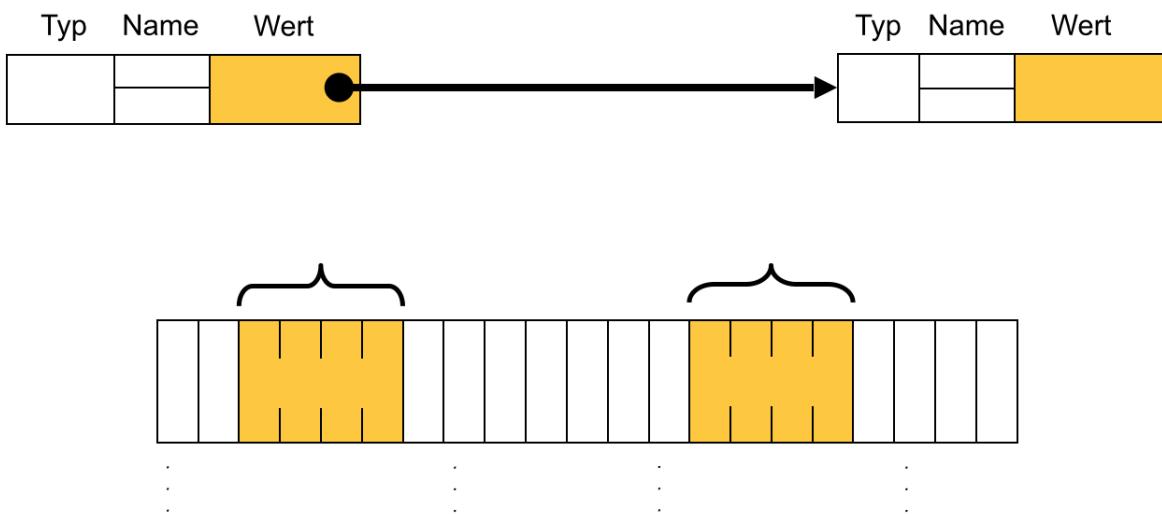
Experimentiere mit Zeigern, Adressen und Referenzen. Als Ausgangsbasis kann folgendes Programmfragment dienen. Fülle danach die untenstehende Skizze aus, um dir klarzumachen, wie Variablen und ihre Speicherabbilder zusammenhängen.

```
int intValue = 42;
int *pIntPtr = &intValue;
cout << "Wert von IntVal " << intValue << endl;
cout << "Wert von &IntPtr " << &intValue << endl;
cout << "Wert von pIntPtr " << pIntPtr << endl;
cout << "Wert von *pIntPtr " << *pIntPtr << endl;
cout << "Wert von &pIntPtr " << &pIntPtr << endl;
```

Speicherabbild

Wir nehmen an, dass Speicheradressen immer 4 Byte (= 32 Bit) breit sind. Trage nun die auftretenden Variablen intValue und pIntPtr in das folgende Speicherabbild ein. Orientiere dich dabei an der Vorlesung (Folien 69 und 70) und trage folgende Dinge ein:

- Typ, Name und Wert jeder Variablen
- Speicheradressen in Bytes
- Der Wert, der an der jeweiligen Speicheradresse steht.



Hinweise

- Die Speicheradressen kannst du frei wählen, der Pointer sollte aber natürlich auf die entsprechend gewählte Adresse zeigen.

Aufgaben zur Speicherverwaltung in C++

Aufgabe 5.2 Bedeutung verstehen

Versuche die Bedeutung folgender Ausdrücke zu verstehen. Welche Regelmäßigkeiten stellst du fest?

```
int intVal = 42;
int *pIntVal = &intVal;
*&intVal;
*&pIntVal;
&*pIntVal;
**&pIntVal;
*&*&intVal;
&*&pIntVal;
*&*&pIntVal;
```

Hinweise

- Gehe dabei von rechts nach links vor.

Aufgabe 5.3 Gültigkeit

Warum sind folgende Ausdrücke ungültig, sinnlos oder sogar gefährlich?

```
*intVal;
**pIntVal;
***&pIntVal;
&*intVal;
&42;
```

Hinweise

- Finde heraus, welchen Typ der Ausdruck hätte haben müssen.
- Nur tatsächlich angelegte Variablen haben Adressen. Ausdrücke wie `a + b` oder direkt kodierte Zahlenliterale wie `42` haben keine Adresse.

Aufgabe 5.4 Variablentausch

Schreibe eine Funktion `swap`, die zwei `int`-Variablen miteinander vertauscht. Probiere dabei beide möglichen Übergabevarianten (per Referenz, per Pointer) aus. Was würde passieren, wenn man die Variablen stattdessen per Wert übergeben würde?

Aufgabe 5.5 Programmanalyse

Sieh dir folgendes Programm an.

Aufgaben zur Speicherverwaltung in C++

```
#include <iostream>

void foo(int &i) {
    int i2 = i;
    int &i3 = i;

    std::cout << "i = " << i << std::endl;
    std::cout << "i2 = " << i2 << std::endl;
    std::cout << "i3 = " << i3 << std::endl;
    std::cout << "&i = " << &i << std::endl;
    std::cout << "&i2 = " << &i2 << std::endl;
    std::cout << "&i3 = " << &i3 << std::endl;
}

int main() {
    int var = 42;
    std::cout << "&var = " << &var << std::endl;
    foo(var);
}
```

Welche Adressen werden übereinstimmen, welche werden sich unterscheiden? Führe das Programm aus. Hast du diese Ausgabe erwartet?

Aufgabe 5.6 Const Correctness

In dieser Aufgabe setzt du dich mit der Bedeutung des Schlüsselworts **const** im Kontext von Pointern auseinander.

Versuche für jede der Variablen im folgenden Code je eine *Verwendung* zu finden, die

- gültig ist (= fehlerfrei kompiliert) und
- nicht gültig ist (= einen Compiler-Fehler wirft).

Was ist jeweils der Grund? Welche Pointer verhalten sich gleich?

```
int i = 1;
int *iP = &i;
const int *ciP = &i;
int const *ciP2 = &i;
int * const icP = &i;
const int * const cicP = &i;
```

Mehrstufige Pointer

Versuche nun das Gleiche mit den folgenden mehrstufigen Pointern.

```
int **iPP = &iP;
const int * const *cicPP = &iP;
int ** const iPcP = &iP;
```

Aufgaben zur Speicherverwaltung in C++

Aufgabe 5.7 Übergabewerte

In der letzten Aufgabe direkt zu Pointern geht es darum, das gerade erlangte Verständnis über Pointer und Referenzen zu festigen und zu kontrollieren. In Tabelle 1 sind in der ersten Spalte Funktionen mit verschiedenen Parametertypen gegeben. In der ersten Zeile findest du verschiedene Variablentypen. Deine Aufgabe ist es nun, zu den verschiedenen Funktionen die passenden Parameter aus den Variablen herzustellen. Falls eine Variable nicht ohne großartige Konversationen verwendet werden kann trage bitte ein **x** ein.

Als Beispiel dient hierfür die erste Zeile.

	int i	int *j	int const * const k	int **l	const int *m
op1(int *)	&i	j	x	*l	x
op2(int)					
op3(int &)					
op4(const int **)					

Tabelle 1: Tabelle für Übergabewerte Aufgabe

Aufgaben zur Speicherverwaltung in C++

Aufgabe 6 [S] Arrays und Zeigerarithmetik

Arrays sind zusammenhängende Speicherbereiche, die mehrere Variablen von gleichem Typ speichern können. Arrays werden in C++ folgendermaßen angelegt: <Typ> <name>[<Größe>];, zum Beispiel:

```
int arr[10]; // array of 10 integers
```

Falls das Array global ist, muss die Größe eine konstante Zahl sein, falls das Array in einer Funktion auf dem Stack angelegt wurde, kann die Größe auch durch eine Variable vorgegeben werden. Auf jeden Fall bleibt diese während der Existenz des Arrays konstant und kann sich nach dem Anlegen nicht mehr ändern.

Ein Array kann direkt bei der Deklaration initialisiert werden:

```
int arr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // array of 10 integers
```

Man kann die Größe optional auch weglassen, in diesem Fall wird sie der Compiler anhand der angegebenen Elemente selbst ermitteln. Auf die einzelnen Elemente des Arrays kann man wie gewohnt über **arr[i]** zugreifen.

Arrays und Zeiger sind in C++ stark miteinander verwandt. So ist der **Bezeichner** des Arrays gleichzeitig die **Adresse des ersten Elements**. Somit kann man sowohl durch ***arr** (Dereferenzierung) als auch durch **arr[0]** auf das erste Element zugreifen. Analog dazu kann man auch einen Zeiger auf das erste Element anlegen:

```
int *pArr = arr;
```

Da die Elemente eines Arrays direkt hintereinander stehen, kann man den Zeiger inkrementieren, um zum nächsten Element zu gelangen (sogenannte Pointerarithmetik). Beispiel:

```
int *pArr = arr;
std::cout << "Address of first element: " << pArr << std::endl;
std::cout << "Address of second element: " << pArr+1 << std::endl;
std::cout << "Address of third element: " << pArr+2 << std::endl;
```

Somit kann man auf beliebige Elemente des Arrays über den Zeiger zugreifen:

```
*(pArr + 0); // first element
*(pArr + 1); // second element
*(pArr + 2); // third element
++pArr; // increment pointer by 1
*(pArr + 0); // second(!!) element of arr
*(pArr + 2); // fourth(!!) element of arr
```

Tatsächlich ist ***(p+i)** in **jeder Hinsicht äquivalent** zu **p[i]**. Das bedeutet, dass man sowohl auf das i-te Element eines Arrays über ***(arr + i)** zugreifen kann als auch über **pointer[i]** auf das Element, auf welches der Zeiger **pointer+i** zeigt!

In C++ findet keine automatische Bereichsprüfung bei Arrayzugriffen statt. Du bist als Programmierer selbst dafür verantwortlich, dass niemals auf ein Element außerhalb der Array-Grenze zugegriffen wird. Falls doch, kann es zu Programmbürtzen oder unerwünschten Effekten wie Buffer-Overflows kommen, die ein erhebliches Sicherheitsrisiko darstellen. Bevorzuge deshalb Container-Klassen wie **std::vector** (oder **std::array** ab C++11) aus der Standardbibliothek anstelle von „rohen“ Arrays. Beachte außerdem, dass der **delete[]**-Operator zwar das Array löscht, den Zeiger jedoch **nicht** auf **NULL** setzt. Dabei entsteht ein *Dangling Pointer*, welcher dazu führen kann, dass später im Programm auf Speicherstellen zugegriffen wird, die nicht reserviert sind. Setze deshalb Zeiger nach einem **delete/delete[]** sofort auf **NULL**, um Speicherfehler zu vermeiden.

Um die Größe eines Arrays zu ermitteln, kannst du den **sizeof()**-Operator benutzen. Dieser gibt generell die Anzahl der Bytes an, die eine Variable verbraucht. Da einzelne Array-Elemente größer als ein Byte sein können, muss die Gesamtgröße des Arrays durch die Größe eines Elements geteilt werden, um auf die Anzahl der Elemente zu kommen.

Aufgaben zur Speicherverwaltung in C++

```
int arr[10];
std::cout << sizeof(arr) << std::endl; // 40 on a typical 32 or 64-bit machine
int len = sizeof(arr) / sizeof(arr[0]);
std::cout << len << std::endl; // always 10
```

Beachte, dass `sizeof()` nicht dazu verwendet werden kann, um die Größe des Arrays herauszufinden, auf die ein Zeiger zeigt. In diesem Fall wird `sizeof()` nämlich die **Größe des Zeigers** und nicht die Größe des Arrays liefern!

```
int arr[10];
int *pArr = arr;
std::cout << sizeof(pArr) << std::endl; // 4 on 32-bit machine, 8 on 64-bit
```

Aufgabe 6.1 Arrays anlegen

Lege in der `main`-Funktion ein `int`-Array mit 10 Elementen an, und initialisiere es mit den Zahlen 1 bis 10. Iteriere in einer Schleife über das Array und gib alle Elemente nacheinander aus.

Aufgabe 6.2 printElements implementieren

In C und C++ kann man Arrays nicht direkt an Funktionen übergeben. Stattdessen übergibt man einen Zeiger auf das erste Element des Arrays. Aufgrund der Äquivalenz von `*(p+i)` und `p[i]` kann man in der Funktion den Zeiger syntaktisch wie das Original-Array verwenden.

Schreibe eine Funktion, die einen `const`-Zeiger auf das erste Element eines Arrays bekommt und alle Elemente ausgibt. Da ein Array, wie bereits erwähnt, ein zusammenhängender Speicherbereich ist, hat die Funktion keine Möglichkeit, anhand des Zeigers herauszufinden, wie groß das Array ist. Denn das Ende des Arrays im Speicher ist unbekannt. Deshalb muss die Größe des Arrays durch einen weiteren Parameter übergeben werden. Dazu solltest du folgenden Typen kennen.

Der Typ `size_t`

Oft wird für die Größenangabe oder Indexierung eines Arrays der Typ `unsigned int` verwendet. Besserer Stil ist es jedoch, für Array-Indexierung oder auch als Laufvariable bei Schleifen den Standard-Typen `size_t`⁷ zu nutzen. Er ist per Definition `unsigned`, da Größen bzw. Indizes nur positiv sein können. Handelt es sich um eine Indexierung von `std::vector` oder `std::string`, so sollte das entsprechende `typedef size_type` genutzt werden⁸. Der Grund für die Verwendung dieser Typen anstelle von `unsigned int` ist folgender: Es wird damit sichergestellt, dass der Typ der Indexvariablen groß genug ist, um Objekte beliebiger Größe indexieren zu können. In diesem Fall würde zwar auch `unsigned int` reichen. Was wäre aber, wenn du keinen Einfluss auf die Größe des Arrays hättest, weil es durch einen Aufruf von außen übergeben wird? Unter extremen Umständen würde ein `unsigned int` evtl. nicht reichen. Es ist deshalb sinnvoll, sich die Verwendung dieses Typs anzueignen. Außerdem trägt es zur Selbstdokumentation deines Codes bei, da sofort ersichtlich ist, dass ein Index oder eine Größe gemeint ist.

Deine Funktion sollte also folgendermaßen aussehen:

```
void printElements(const int *const array, const size_t size);
```

Hinweise

- Damit der Typ `std::size_t` genutzt werden kann, benötigst du den Standard-Library-Header `#include <cstddef>`.

⁷ siehe http://en.cppreference.com/w/cpp/types/size_t

⁸ bei Strings nutzt man `std::string::size_type` und bei `std::vector` entsprechend `std::vector<T>::size_type`

Aufgaben zur Speicherverwaltung in C++

Aufgabe 6.3 Offset-basierte Ausgabe

Wie wir vorher gesehen haben, kann man mit Zeigern auch rechnen und diese nachträglich ändern. Anstatt mit einem Index das Array zu durchlaufen, kann man stattdessen bei jeder Iteration den Zeiger selbst inkrementieren!

```
for(const int *p = array; p != array + 10; ++p) {  
    int i = *p;      // *p contains current element  
    // ...  
}
```

Schreibe die Funktion aus der vorherigen Aufgabe so um, dass sie einen laufenden Zeiger anstatt eines Indexes verwendet.

Aufgabe 6.4 Iterator-basierte Ausgabe

Ebenso kann man auch die Arraygröße auf eine andere Weise übergeben, indem man die Adresse des Elements nach dem letzten Element angibt. Dadurch werden Schleifen der folgenden Form möglich.

```
for(const int *p = begin; p != end; ++p) {  
    int i = *p;      // *p contains current element  
    // ...  
}
```

möglich. Schreibe die Funktion aus der vorherigen Aufgabe entsprechend um. Vergiss nicht, den Zeiger als `const` zu definieren, da Elemente nur gelesen werden. Du kannst hier `const` doppelt verwenden, um auch sicherzustellen, dass der `end`-Zeiger nicht verändert wird.

Aufgabe 6.5 Subarrays ausgeben

Die obige Methode, über Elemente eines Arrays zu iterieren, mag dir zunächst etwas ungewöhnlich erscheinen. Sie hat jedoch den Vorteil, dass man anstatt des ganzen Arrays auch kleinere zusammenhängende Teile davon an Funktionen übergeben kann, indem man Zeiger auf die entsprechenden Anfangs-und Endelemente setzt. Beispiel:

```
int arr[10];  
printElements(arr+5, arr+8); // Print elements with index 5, 6, 7
```

Experimentiere etwas mit dieser Übergabemethode in deiner eigenen Funktion!

Aufgabe 6.6 Arrays auf dem Heap

Bisher haben wir das Array auf dem Stack angelegt. Mit `new[]` kann man ein Array auf dem Heap erzeugen. Dabei wird die Adresse des ersten Elements in einem Zeiger gespeichert. Mittels `delete[]` muss man den belegten Speicher nach Benutzung freigeben. Beispiel:

```
int *pArr = new int[10]; // size can be a variable  
doSomethingWith(pArr, 10);  
delete[] pArr; // <-- notice the [] !
```

Beachte die `[]` nach `delete`. Diese bewirken, dass das gesamte Array und nicht bloß das erste Element gelöscht wird. Woher weiß `delete[]` aber, wie viel Speicher freigegeben werden muss? Wenn du Speicher auf dem Heap mit `new/new[]` reservierst, merkt sich die Speicherverwaltung intern, wie groß dieser allozierte Bereich ist. Die Information wird normalerweise in einem "Head-Segment" vor dem Speicherbereich, der die eigentlichen Daten enthält, abgelegt. Dieser Speicherort ist allerdings nicht standardisiert und kann je nach Compiler variieren. Beim Aufruf von `delete/delete[]`

Aufgaben zur Speicherverwaltung in C++

wird diese Information dann ausgelesen. `delete` ruft den Destruktor des Objekts auf, das an dieser Speicherstelle liegt und gibt danach den Speicher an das Betriebssystem zurück. `delete[]` hingegen ruft für **jedes** Element des Arrays den Destruktor auf und gibt danach ebenfalls den Speicher frei.

Ein Anwendungsfall von dynamischen Arrays auf dem Heap sind Funktionen, die ein Array von vorher unbekannter Größe zurückgeben.

Schreibe nun eine Funktion, die beliebig viele Zahlen von der Konsole mittels `std::cin` einliest. Der Benutzer soll dabei zuvor gefragt werden, wie viele Zahlen er eingeben möchte. Speichere die Zahlen in einem dynamisch angelegten Array ab und lasse die Funktion einen Zeiger darauf zurückgeben. Hier ist ein Beispiel wie `std::cin` zu verwenden ist:

```
size_t size;
std::cout << "Größe: ";
std::cin >> size;
std::cout << "Gewählte Größe: " << size << std::endl;
```

Zusätzlich zum Zeiger muss die Funktion auch die Möglichkeit haben, ihrem Aufrufer die Größe des angelegten Arrays mitzuteilen. Füge der Funktion deshalb einen weiteren Parameter hinzu, in dem entweder per Referenz oder per Zeiger eine Variable übergeben wird, um dort die Größe abzulegen⁹.

Gib die eingelesenen Werte auf der Konsole aus. Vergiss nicht, am Ende den Speicher freizugeben.

⁹ Du merkst sicherlich schon jetzt, dass es umständlich/fehleranfällig ist, wenn man die Größe eines Arrays separat speichern und übergeben muss.

Aufgaben zur Speicherverwaltung in C++

Aufgabe 7 [S] Verkettete Listen

In dieser Aufgabe wollen wir eine doppelt verkettete Liste von Integern implementieren. Dazu brauchen wir zwei Klassen: `ListItem` stellt ein Element der Liste mit dessen Inhalt dar und `List` speichert die Zeiger auf Anfangs- und Endelemente und bildet den eigentlichen Zugangspunkt für die Liste.

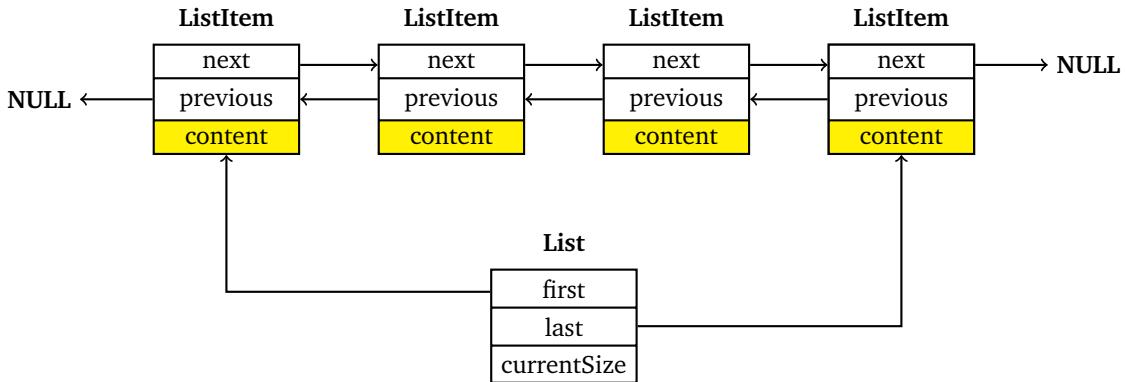


Abbildung 1: Linked List

Wir werden am Tag 4 auf dieser Aufgabe aufbauen und die Liste um weitere Funktionen erweitern. Behalte dies bitte im Hinterkopf und lösche deine Lösung nicht. Falls du mit dieser Aufgabe bis dahin nicht fertig sein solltest, kannst du natürlich auch die Musterlösung als Ausgangspunkt nehmen.

Aufgabe 7.1 Klasse ListItem

Implementiere die Klasse `ListItem`, welche die zu speichernde Zahl sowie Verweise auf das vorherige und nächste `ListItem` als Attribute hat. Verwende dazu Zeiger und keine Referenzen, da Referenzen nachträglich nicht mehr geändert werden können. Auch können Referenzen nicht `NULL` sein, was in unserem Fall nötig ist, um zu markieren, dass ein Element keine Vorgänger oder Nachfolger hat.

Der Konstruktor sollte sowohl seine eigenen `next` und `previous` Zeiger initialisieren, als auch die seiner Vorgänger- und Nachfolgerelemente. Die Methode `getContent()` soll eine Referenz auf den Inhalt zurückgeben, damit dieser durch eine Zuweisung modifiziert werden kann.

```
class ListItem {
public:
    /**
     * create a list item between two elements with a given given content
     * (also modify previous->next and next->previous)
     */
    ListItem(ListItem *prev, ListItem *next, int content);
    /**
     * delete this list item (also change previous->next and next->previous
     * to not point to this item anymore)
     */
    ~ListItem();
    int & getContent();           // get a reference to the contained data
    ListItem * getNext();        // get the next list item or NULL
    ListItem * getPrevious();    // get the previous list item or NULL
private:
    ListItem *previous;         // previous item in list
    ListItem *next;             // next item in list
```

Aufgaben zur Speicherverwaltung in C++

```
    int content;           // content of this list item
};
```

Aufgabe 7.2 Privater Copy-Konstruktor

Unsere `ListItem` Klasse hat einen kleinen Design-Fehler: Da wir keinen Copy-Konstruktor definiert haben, generiert der Compiler automatisch einen. Dieser kopiert einfach die einzelnen Attribute des Ursprungsobjekts (sogenannte „flache“ Kopie/Shallow Copy). In unserem Fall ergibt das Kopieren eines `ListItems` jedoch semantisch keinen Sinn, weil dabei ein hängendes `ListItem` entstehen würde, welches nicht mit der Liste verknüpft ist, aber dennoch auf andere Items der Liste zeigt.

Deklariere in der Headerdatei einen `private` Copy-Konstruktor und einen `private operator=`. Dadurch können beide nie aufgerufen werden und der Kompiler kann dies zur Kompilierzeit überprüfen.

```
private:
    ListItem(const ListItem &other);    // private copy constructor (without implementation)
    ListItem& operator=(const ListItem &other); // private assignment operator (w/o
                                                implementation)
```

Hinweise

- Alternativ kann man ab C++11 Funktionen explizit löschen:

```
ListItem(const ListItem &other) = delete;
ListItem& operator=(const ListItem &other) = delete;
```

Aufgabe 7.3 Klasse List

Implementiere nun die Klasse `List`. Achte bei den Methoden zum Einfügen und Entfernen von Elementen darauf, dass bei einer leeren Liste eventuell sowohl die `first` als auch `last` Zeiger modifiziert werden müssen. Vergiss nicht, `currentSize` bei jeder Operation entsprechend anzupassen.

Außerdem sollten alle erstellten `ListItems` auf dem Heap abgelegt werden.

Falls die Liste leer ist, sollten `deleteFirst()` und `deleteLast()` einfach nichts ändern. Außerdem werden in dem Fall `getFirst()`, `getLast()` und `getNthElement()` einen Segmentation fault produzieren. Lieber würde man hier einen Fehler werfen, aber Exceptions haben wir an dieser Stelle noch nicht behandelt. Die Erweiterung um Exceptions folgt dann in Aufgabe 12.3.

`operator<< implementieren`

Implementiere außerdem den `operator<<`, um bequem Listen auf der Kommandozeile auszugeben. Die übergebene Referenz ist – entgegen der üblichen Konvention für `operator<<` – nicht `const`, da wir ansonsten entsprechend eine `const`-Version des `ListIterator` benötigen würden.

Vergiss hier nicht, `operator<<` als `friend` von `List` zu deklarieren (wie zuvor bei `Vector`).

```
#include <cstddef>

class List {
public:
    List();                      // create an empty list
    ~List();                     // delete the list and all of its elements
    List(const List &other);    // create a copy of another list
```

Aufgaben zur Speicherverwaltung in C++

```
void appendElement(int i);      // append an element to the end of the list
void prependElement(int i);    // prepend an element to the beginning of the list
void insertElementAt(int i, size_t pos); // insert an element i at position pos
size_t getSize() const;        // get the number of elements in list
int & getNthElement(size_t n);   // get content of the n-th element.
int & getFirst();              // get content of the first element
int & getLast();              // get content of the last element
int deleteFirst();            // delete first element and return it (return 0 if empty)
int deleteLast();             // delete last element and return it (return 0 if empty)
int deleteAt(size_t pos);     // delete element at position pos
private:
    ListItem *first, *last;    // first and last item pointers (nullptr if list is empty)
    size_t currentSize;        // current size of the list
};

#include <iostream>

/** Print the given list to the stream. N.B. list should actually be const but then we would
    need const ListIterators */
std::ostream &operator<<(std::ostream &stream, List &list);
```

Aufgabe 7.4 Liste testen

Teste deine Implementation. Füge der Liste Elemente von beiden Seiten hinzu und lösche auch wieder welche. Kopiere die Liste und gib die Elemente nacheinander aus.

Aufgabe 7.5 ListIterator

Bisher haben wir über `getNthElement()` auf die Elemente der Liste zugegriffen. Diese Methode kann insbesondere bei langen Listen sehr langsam sein. Deshalb werden wir einen Iterator schreiben, über den man auf die Listenelemente sequentiell zugreifen kann. Der Iterator soll dabei einen Zeiger auf das aktuell betrachtete Element der Liste halten.

Um den Zugriff möglichst komfortabel zu gestalten, werden wir den Iterator als eine Art Zeiger implementieren, den man über `++` und `--` in der Liste verschieben kann. Um auf ein Element zuzugreifen, überladen wir den Dereferenzierungsoperator `operator*`. Somit können wir unsere Liste ähnlich zu `std::vector` verwenden:

```
for (ListIterator iter = list.begin(); iter != list.end(); iter++) {
    cout << *iter << endl;
}
```

Konstruktor und Operatoren

Beginne mit einer Grundversion des Iterators. Erstelle einen Konstruktor, der die Attribute des Iterators (Zeiger auf aktuelles Element und Zeiger auf die Liste) entsprechend initialisiert. Implementiere Vergleichsoperator `operator!=` sowie den Dereferenzierungsoperator `operator*`. Der Dereferenzierungsoperator sollte den Inhalt des aktuellen Items zurückgeben. Du brauchst nicht zu prüfen, ob `item` tatsächlich auf ein gültiges Element zeigt (Das machen/können Iteratoren aus der Standardbibliothek übrigens auch nicht!). Zum Vergleichen zweier Iteratoren prüfe, ob die `item` und `list` Zeiger identisch sind. Vergleiche nicht den Inhalt der Items, da der Vergleich auch dann funktionieren soll, wenn `item` `NULL` ist, wenn der Iterator also auf kein Element zeigt.

```
class ListIterator {
public:
    // create a new list iterator pointing to an item in a list
```

Aufgaben zur Speicherverwaltung in C++

```
ListIterator(List *list, ListItem *item);  
// get the content of the current element  
int& operator*();  
// check whether this iterator is not equal to another one  
bool operator!=(const ListIterator &other) const;  
private:  
List *list;  
ListItem *item;  
};
```

Zugriff von außen: ListIterator als friend-Klasse

Du wirst in den folgenden Methoden auf private Attribute von List zugreifen müssen. Um dies zu ermöglichen, könnte man öffentliche Getter für die Items der Liste schreiben. Dadurch würde jedoch jeder die Möglichkeit bekommen, direkt auf die ListItems der Liste zuzugreifen, was dem Geheimnisprinzip zuwiderläuft. Deshalb werden wir ListIterator stattdessen explizit erlauben, auf **private**-Attribute der Liste zuzugreifen. Dazu müssen wir ListIterator als **friend** von List deklarieren. Füge dazu folgende Zeile (an beliebiger Stelle, üblich ist der Anfang der Klasse) zur Klassendefinition von List hinzu:

```
friend class ListIterator;
```

Iterator vorwärts bewegen mittels operator++

Implementiere den **operator++** zum Inkrementieren des Iterators. Falls der Iterator zuvor auf kein Item zeigte (**item == NULL**), soll er nun auf das erste Element der Liste gesetzt werden. Die Prototypen dazu lauten:

```
ListIterator& operator++(); // increment this iterator and return itself (prefix++)  
ListIterator operator++(int); // increment this iterator and return the previous (postfix++)
```

Bei der Überladung des **operator++** muss eine Sonderregelung beachtet werden. Dieser Operator kann sowohl als Postfix (z.B. **iter++**) als auch Präfix (z.B. **++iter**) verwendet werden. Um den Compiler darüber zu informieren, welche Variante wir überladen, wird beim Postfix-Operator ein Dummy-Parameter vom Typ **int** definiert. Dieser dient nur der syntaktischen Unterscheidung und hat keine weitere Bedeutung.

Beachte außerdem, dass bei Präfix-Operationen der Iterator sich selbst zurückgeben sollte, während bei Postfix-Operationen eine Kopie des Iterators zurückgegeben wird, die auf das vorherige Element zeigt. Das ist auch der Grund, warum die Präfix-Form von **operator++** (und **operator--**) effizienter ist als die Postfix-Form. Daher sollte die Präfix-Form dieser Operatoren die bevorzugte Variante sein, falls kein besonderer Grund für die Postfix-Form vorliegt.

Zum besseren Verständnis ist ein Teil der Implementation gegeben:

```
// Prefix ++ -> increment iterator and return it  
ListIterator& ListIterator::operator++() {  
    if (item == NULL) {  
        item = ... // set item to first item of list  
    }  
    else {  
        item = ... // set item to next item of current item  
    }  
    return *this; // return itself  
}
```

Aufgaben zur Speicherverwaltung in C++

```
// Postfix ++ -> return iterator to current item and increment this iterator
ListIterator ListIterator::operator++(int) {
    ListIterator iter(list, item); // Store current iterator
    if (item == NULL) {
        item = ... // set item to first item of list
    }
    else {
        item = ... // set item to next item of current item
    }
    return iter; // return iterator to previous item
}
```

Iterator rückwärts bewegen mittels operator--

Überlade auf die gleiche Weise auch den `operator--` sowohl in Postfix als auch Präfix-Form.

Iteratoren in List erzeugen

Nun ist unsere Implementation fast komplett und wir brauchen nur noch Methoden, um Iteratoren zu erzeugen. Implementiere dazu die folgenden Methoden innerhalb der List Klasse:

```
ListIterator begin();
ListIterator end();
```

Die erste Methode soll einen Iterator erzeugen, der auf das erste Listenelement zeigt. `end()` hingegen soll einen Iterator erzeugen, der nicht auf ein bestimmtes Element der Liste zeigt. Stattdessen soll hier auf das "Element" hinter dem letzten Element der Liste gezeigt werden (*past-the-end element*). Genauer gesagt: Es sollte konsistent mit dem Wert sein, den `operator++` zurückgibt, wenn man ihn auf dem letzten Listenelement aufruft.

Höchstwahrscheinlich wirst du Probleme bei der Kompilierung haben. Dies liegt an der zirkulären Abhängigkeit zwischen List und ListIterator. Gehe dazu folgendermaßen vor: Verschieben die `#include`-Anweisungen für die Header von List und ListItem aus ListIterator.h nach ListIterator.cpp und füge in ListIterator.h folgendes hinzu

```
class ListItem;
class List;
```

Dies sind Vorwärtsdeklarationen (**Forward Declaration**), die dem Compiler sagen, dass die Klassen existieren, aber später definiert werden. Nun kannst du problemlos ListIterator.h in List.h einbinden.

Aufgabe 7.6 Liste mit ListIterator testen

Teste deine Implementation. Erstelle eine Liste, füge Elemente hinzu und iteriere über Listenelemente:

```
for (ListIterator iter = list.begin(); iter != list.end(); iter++) {
    cout << *iter << endl;
}
```

Warum kann man **nicht** rückwärts durch die Liste iterieren, indem man einfach die Aufrufe `list.begin()` und `list.end()` tauscht und `iter--` statt `iter++` verwendet? Denke daran, worauf die von `begin()` und `end()` zurückgegebenen Iteratoren zeigen.

Hinweise

- In der Standardbibliothek gibt es hierfür `rbegin()` und `rend()`

Aufgaben zur Speicherverwaltung in C++

Aufgabe 8 [S] Smart Pointers

In dieser Aufgabe werden wir uns mit der Benutzung von Smart Pointers vertraut machen. Dazu werden wir die Smart Pointer Klassen `std::shared_ptr` und `std::weak_ptr` verwenden. Binde hierfür den Systemheader `memory` ein.

Aufgabe 8.1

Erstelle eine Klasse `TreeNode`, die einen Knoten eines Binärbaums darstellt. Jeder Knoten hat einen Inhalt vom Typ `int` sowie einen Zeiger auf seine beiden Kindknoten. Statt „roher“ Zeiger verwenden wir Smart Pointers, die das Speichermanagement übernehmen. Dadurch wird es nicht nötig sein, Kindknoten manuell zu löschen. Sie werden automatisch entfernt, sobald der Wurzelknoten gelöscht ist und keine Zeiger mehr auf den Kindknoten zeigen.

```
#include <memory>
class TreeNode;

typedef std::shared_ptr<TreeNode> TreeNodePtr; // typedef for better readability

class TreeNode {
public:
    /** create a new tree node and make it shared */
    static TreeNodePtr createNode(int content, TreeNodePtr left = TreeNodePtr(), TreeNodePtr
        right = TreeNodePtr());
    ~TreeNode();
private:
    TreeNode(int content, TreeNodePtr left, TreeNodePtr right); // create a tree node
    TreeNodePtr leftChild, rightChild; // left and right child
    int content; // node content
};
```

Der Konstruktor von `TreeNode` ist privat, weil nur die Smart Pointer die Verantwortung für die Lebenszeit eines Objektes übernehmen sollen und bestimmen, wann es gelöscht wird. Würde man `TreeNode`-Objekte direkt auf dem Stack anlegen, kann es passieren, dass der Objektdestruktor mehrmals aufgerufen wird – einmal vom Smart Pointer und einmal beim Verlassen der Funktion. Ebenso sollten wir keine Rohzeiger auf das Objekt erzeugen, da diese das Speichermanagement der Smart Pointer umgehen. Stattdessen stellen wir eine statische Methode bereit, um `TreeNode`-Objekte auf dem Heap zu erzeugen und diese direkt einem Smart Pointer zu übergeben.

Diese statische Methode `createNode()` erhält zusätzlich zum Inhalt des Knotens noch zwei `TreeNodePtr` für das linke bzw. rechte Element. Als Defaultwert für diese Parameter ist `TreeNodePtr()` angegeben. Dies erstellt einen leeren Smart Pointer, was dem Null-Pointer bei Rohzeigern entspricht.

Implementiere den Konstruktor, Destruktor sowie `createNode`. Der Konstruktor sollte die Attribute entsprechend initialisieren. Schreibe auch eine Textausgabe, die den Zeitpunkt der Erzeugung eines `TreeNodes` deutlich macht. Der Destruktor braucht die Kindknoten nicht zu löschen, da dies bei der Zerstörung des Elternknotens automatisch geschieht. Füge auch hier eine Textausgabe ein, die die Zerstörung des Objekts sichtbar macht.

Das Schlüsselwort `static` sowie die Default-Parameter müssen bei der Implementation der Methode ausgelassen werden. Der Smart Pointer für die Rückgabe wird mit einem Zeiger auf ein `TreeNode`-Objekt initialisiert. Somit lautet der Methodenrumpf

```
TreeNodePtr TreeNode::createNode(int content, TreeNodePtr left, TreeNodePtr right) {
    return TreeNodePtr(new TreeNode(...));
}
```

Aufgaben zur Speicherverwaltung in C++

Hinweise

- Über `std::make_shared(<Konstruktorparameter>)` kann man auch einen Smart Pointer erhalten.
- Zur Dokumentation von `typedef` kannst du den Tag `@typedef` verwenden. Sonst verhält es sich mit dem Dokumentieren so wie immer.

Aufgabe 8.2

Teste, ob die einzelnen Knoten tatsächlich gelöscht werden, sobald kein Zeiger mehr auf den Elternknoten zeigt. Erstelle dafür einen kleinen Baum:

```
TreeNodePtr node = TreeNode::createNode(1, TreeNode::createNode(2), TreeNode::createNode(3));
```

Führe das Programm aus und beobachte die Ausgabe. Sobald `main` verlassen wird, wird der Zeiger `node` gelöscht, und somit auch das dahinterliegende `TreeNode`-Objekt mit all seinen Kindknoten.

Um ganz sicher zu gehen, dass der Baum tatsächlich beim Löschen des letzten Zeigers zerstört wurde und nicht etwa durch das Beenden des Programms, kannst du `node` mit einem anderen Baum überschreiben. Füge in diesem Fall am Ende des Programms eine Textausgabe hinzu, damit ersichtlich wird, dass der erste Baum noch vor Verlassen der `main` gelöscht wurde.

Aufgabe 8.3

Nun wollen wir `TreeNode` so erweitern, dass jeder Knoten Kenntnisse über seinen Elternknoten besitzt. Füge das Attribut

```
TreeNodePtr parent; // parent node
```

hinz. Da der Elternknoten beim Erzeugen eines `TreeNodes` undefiniert ist, brauchst du den Konstruktor nicht zu ändern. `parent` wird dann automatisch mit `NULL` initialisiert.

Implementiere die folgende Methode, die einem Knoten seinen Elternknoten zuweist:

```
void setParent(const TreeNodePtr &p); // set parent of this node
```

Hinweise

- `p` wird in diesem Fall nur deshalb als `const` Referenz übergeben, da es verhältnismäßig aufwändig ist, einen Smart Pointer zu kopieren. Beachte, dass im obigen Fall der Smart Pointer selbst `const` ist, und nicht das Objekt, worauf er zeigt.

Jetzt muss noch `createNode()` modifiziert werden, sodass `setParent()` auf den Kindknoten aufgerufen wird. Da ein Smart Pointer den `operator*` und den `operator->` überladen hat, lässt er sich syntaktisch wie ein normaler Zeiger benutzen. Um zu überprüfen, ob ein Smart Pointer auf ein Objekt zeigt, kann dieser implizit nach `bool` gecastet werden. Somit lautet die neue Implementation von `createNode()`:

```
TreeNodePtr TreeNode::createNode(int content, TreeNodePtr left, TreeNodePtr right) {
    TreeNodePtr node(new TreeNode(content, left, right));
    if (left) {
        left->...; // set parent node
    }
    if (right) {
        right->...; // set parent node
    }
    return node;
}
```

Aufgaben zur Speicherverwaltung in C++

Aufgabe 8.4

Teste deine Implementation. Du brauchst dazu in `main` nichts zu ändern.

Erschreckenderweise siehst du nun, dass überhaupt keine `TreeNode`-Objekte mehr gelöscht werden. Die Ursache dafür ist die zirkuläre Abhängigkeit zwischen Kind- und Elternknoten. Denn selbst wenn sie keine Zeiger auf den Wurzelknoten eines Baumes haben, verweisen die Kindknoten noch immer darauf.

Um dieses Problem zu lösen, müssen die Verweise zum Elternknoten *schwach* (`weak`) sein. Ein Knoten darf gelöscht werden, wenn nur noch schwache Zeiger (oder keine) auf ihn verweisen. Verwende dazu `std::weak_ptr` und erstelle ein neues `typedef` für einen schwachen `TreeNode` Smart Pointer:

```
typedef std::weak_ptr<TreeNode> TreeNodeWeakPtr;
```

Ändere nun den Typ von `parent` auf `TreeNodeWeakPtr`. Es müssen keine weiteren Änderungen gemacht werden, da starke Zeiger (`shared_ptr`) implizit in schwache Zeiger (`weak_ptr`) umgewandelt werden können.

Hinweise

- **Faustregel:** Wenn ein Objekt ein anderes kontrolliert oder enthält, verwende Shared Pointer vom Container/Besitzer zum anderen Objekt und einen Weak Pointer für die umgekehrte Richtung.

Aufgabe 8.5

Teste deine Implementation. Nun sollte sich `TreeNode` wie gewünscht verhalten.

Aufgaben zur Objektorientierung in C++

Aufgabe 9 [O] Vererbung und Polymorphie

In dieser Aufgabe sollst du Konzepte der Vererbung und Polymorphie unter Verwendung abstrakter Funktionen erlernen.

Aufgabe 9.1 Klasse Person

Implementiere eine Klasse Person, die eine Person mit einem Namen darstellt. Füge allen Konstruktoren und Destruktoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Person {
public:
    Person(const std::string &name);           // initialize the name of the person
    ~Person();                                // destructor
    std::string getInfo() const;               // get the name of the person
protected:
    std::string name;                         // the name of the person
};
```

Hinweise

- Verwende `#include <string>` um `std::string` zu verwenden.
- Um ein String-Literal an eine `std::string` Variable anzuhängen, musst du aus dem String-Literal zuerst ein `std::string`-Objekt machen: `std::string text = std::string("Name: ") + name;.`

Aufgabe 9.2 Klasse Student

Implementiere eine Klasse Student, die von Person erbt (`public`) und einen Studenten mit einer Matrikelnummer (ebenfalls `std::string`) modelliert. Rufe in der Initialisierungsliste den entsprechenden Konstruktor der Elternklasse Person mittels `Person(name)` auf. Füge allen Konstruktoren und Destruktoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Student: public Person { // public inheritance
public:
    Student(const std::string &name, const std::string &studentID); // init name and ID
    ~Student();                                // destructor
    std::string getInfo() const;               // Person::getInfo() - get name and
                                                // studentID
private:
    std::string studentID;                    // the student ID of the student
};
```

Hinweise

- Du kannst bei Bedarf die `getInfo()`-Implementation der Elternklasse Person von Student aus mittels `Person::getInfo()` aufrufen.

Aufgabe 9.3 Test

Erstelle nun in `main()` je eine Person und einen Studenten und gib deren Daten auf der Konsole aus. Vergewissere dich, dass bei Student auch die Matrikelnummer ausgegeben wird. Schau dir auch die Ausgaben der Konstruktoren und Destruktoren an, und versuche, diese nachzuvollziehen.

Aufgaben zur Objektorientierung in C++

Implementiere dann folgende Funktion und teste deine Implementation erneut, indem du `printPersonInfo()` mit beiden Personentypen aufruft.

```
void printPersonInfo(const Person *person); // print person information on console
```

Hinweise

- Dadurch dass `Person` als `const` Zeiger übergeben wird, können auch Unterklassen von `Person`, wie z.B. `Student`, übergeben werden.

Aufgabe 9.4 Dynamic Dispatch bei `printPersonInfo`

Du merkst, dass `printPersonInfo()` unabhängig von übergebenem Typ einer Person immer nur den Namen der Person ausgibt, aber nicht die Matrikelnummer. Der Grund dafür ist, dass `getInfo()` nicht als `virtual` deklariert wurde und deshalb auch kein dynamischer Dispatch der Methode stattfindet. Deklariere daher `getInfo()` in beiden Klassen als `virtual`.

Teste deine Implementation erneut und vergewissere dich, dass nun immer die richtige Methode aufgerufen wird.

Hinweise

- Möchte man Methoden einer Basisklasse überschreiben, muss `virtual` in der Basisklasse gesetzt werden. In den abgeleiteten Klassen kann `virtual` weggelassen werden, es wird dann vom Compiler ergänzt. Es ist aber hilfreich, auch dort der Lesbarkeit halber das Schlüsselwort zu verwenden.
- Erst ab C++11 gibt es die Möglichkeit mit dem Schlüsselwort `override` zu deklarieren, dass eine Funktion eine andere (virtuelle) überschreibt (vergleichbar mit der Annotation `@Override` in Java).

Aufgabe 9.5 Virtueller Destruktor

Lege einen Studenten mit `new` dynamisch auf dem Heap an und speichere die Adresse in einem Zeiger auf eine `Person`. Lösche die Person anschließend mit `delete`.

```
Person *pTim = new Student("Tim", "321654");
delete pTim;
```

Analysiere die Konsolenausgabe. Es wird nur der Destruktor von `Person` aufgerufen, obwohl es sich um ein Objekt vom Typ `Student` handelt. Auch hier liegt es daran, dass kein dynamischer Dispatch bei der Zerstörung erfolgt. Deklariere deshalb in beiden Klassen den Destruktor als `virtual` und teste die Korrektheit der Destruktoraufrufe.

Hinweise

- Faustregel: Besitzt eine Klasse mindestens eine virtuelle Funktion, so sollte auch der Destruktor virtuell sein.

Aufgaben zur Objektorientierung in C++

Aufgabe 10 [O] Pure-Virtual-Methoden

In dieser Aufgabe wollen wir Vererbung und Polymorphie dazu nutzen, um mathematische Ausdrücke als Bäume von Primitivoperationen zu modellieren. Dazu werden wir eine abstrakte Oberklasse `Expression` mit der abstrakten Methode `compute()` erstellen. Einzelne Knotentypen wie Addition und Subtraktion werden von `Expression` abgeleitet und implementieren `compute()`, um die jeweilige Operation zu realisieren.

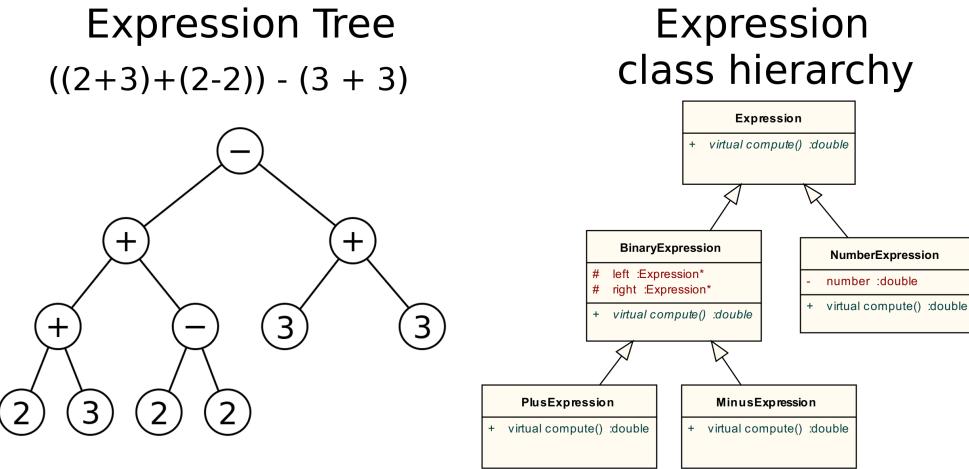


Abbildung 2: Beispelausdruck mit Ausdrucksbaum und Klassenhierarchie

- a) **Klasse Expression:** Schreibe die abstrakte Klasse `Expression`. Diese soll als Basisklasse für alle Ausdrücke dienen. Implementiere einen parameterlosen Konstruktor und einen virtuellen Destruktor, die je eine Meldung auf der Konsole ausgeben, sodass es bei der Ausführung ersichtlich wird, wann eine `Expression` erzeugt und wann zerstört wird. Deklariere außerdem eine abstrakte (pure `virtual`) Methode `virtual double compute() = 0;`, die das Ergebnis des Ausdrucks berechnen und zurückgeben soll.

Hinweise

- Anders als in Java muss die Klasse nicht explizit als `abstract` gekennzeichnet werden - es reicht, wenn sie mindestens eine `pure virtual` Methode enthält.
- b) **Klasse NumberExpression:** Schreibe die Klasse `NumberExpression`, die ein (Baum-)Blatt mit einer Zahl darstellt. Dementsprechend soll `NumberExpression` von `Expression` erben und ein Attribut zum Speichern einer Zahl besitzen, das im Konstruktor initialisiert wird. Implementiere den Konstruktor und virtuellen Destruktor und versehe auch diese mit einer Konsoleausgabe. Die Methode `compute()` gibt die gespeicherte Zahl zurück.
- c) **Klasse BinaryExpression:** Schreibe die abstrakte Klasse `BinaryExpression` mit den `protected` Attributen `Expression *left, *right`. Implementiere den Konstruktor und virtuellen Destruktor mit entsprechender Ausgabe.
- d) **Klassen PlusExpression und MinusExpression:** Schreibe die Klassen `PlusExpression` und `MinusExpression`, die von `BinaryExpression` erben und eine Addition bzw. Subtraktion realisieren. Implementiere die Kon- und Destruktoren sowie die `compute()` Methode.
- e) **Testlauf:** Teste deine Implementation. Ein gutes Beispiel findest du in Abbildung weiter oben. Schaue dir die Ausgabe genau an und versuche anhand der gegebenen Klassenhierarchie die Reihenfolge der Erzeugung und Zerstörung von Objekten nachzuvollziehen.

Aufgaben zur Objektorientierung in C++

Aufgabe 11 [O] Mehrfachvererbung

Verwende den Code der Aufgabe 9 als Basis.

Aufgabe 11.1 Klasse Employee

Schreibe die Klasse `Employee`, die einen Mitarbeiter darstellt. `Employee` soll von `Person` erben und den Namen seines Vorgesetzten als Attribut beinhalten. Erweitere auch entsprechend die Methode `getInfo()`.

Aufgabe 11.2 Klasse StudentAssistant

Schreibe nun eine Klasse `StudentAssistant`, die eine wissenschaftliche Hilfskraft modelliert. Eine wissenschaftliche Hilfskraft ist ein Student und gleichzeitig auch ein Mitarbeiter. Dementsprechend soll `StudentAssistant` sowohl von `Student` als auch von `Employee` erben. Das heißt es werden je ein `Student`- und ein `Employee`-Objekt im Konstruktor initialisiert. Weitere Attribute sind nicht nötig. Überschreibe `getInfo()`, um alle Daten auszugeben. Ändere dazu die Sichtbarkeit der Attribute sowohl von `Student` als auch von `Employee` von `private` auf `protected`.

Du wirst feststellen, dass sich die Klasse nicht kompilieren lässt, falls du das Attribut `name` direkt verwendest, da in einer `StudentAssistant`-Instanz zwei Instanzen von `Person` vorhanden sind - je eine von jeder Elternklasse. Deshalb musst du mittels dem Scope-Operator `::` angeben, welche Klasse du genau meinst.

```
Employee::name  
// or  
Student::name
```

Teste deine Implementation, indem du das Ergebnis von `getInfo()` direkt in der `main` ausgibst.

Aufgabe 11.3 Virtuelle Vererbung

Versuche nun, `printPersonInfo()` mit einer Instanz von `StudentAssistant` aufzurufen. Auch hier wird der Compiler mit einer Fehlermeldung abbrechen, da er nicht weiß, welche der beiden Basisklassen er nehmen soll. Diesmal ist es in C++ allerdings nicht mehr möglich, die Basisklasse zu spezifizieren, weshalb wir anders vorgehen werden. Wir sorgen mittels virtueller Vererbung dafür, dass `Person` nur ein Mal in `StudentAssistant` vorhanden ist.

Lasse dazu `Student` und `Employee` virtuell von `Person` erben. Noch lässt sich das Programm nicht kompilieren, denn sowohl `Student` als auch `Employee` versuchen, einen Konstruktor von `Person` aufzurufen. Da `Person` aber nur ein einziges mal in `StudentAssistant` vorhanden ist, müsste der Konstruktor demnach zwei mal aufgerufen werden – einmal von `Student` und einmal von `Employee`. Dies würde jedoch grob gegen die Sprachprinzipien verstößen. Deshalb wird der Konstruktor von `Person` weder von `Student` noch von `Employee` aufgerufen! Stattdessen müssen wir in der Initialisierungsliste von `StudentAssistant` angeben, welcher Konstruktor von `Person` aufgerufen werden soll. Die Konstruktoraufrufe innerhalb von `Student` und `Employee` laufen stattdessen ins Leere, auch wenn sie syntaktisch vorhanden sind! Füge deshalb ein `Person(name)` in die Initialisierungsliste von `StudentAssistant` hinzu.

Teste deine Implementation. Versuche auch Folgendes: Ändere die Namen in den Konstruktoraufrufen von `Student` und `Employee` in der Initialisierungsliste von `StudentAssistant` und beobachte die Ausgabe. Mache dir dadurch klar, welche Probleme Mehrfachvererbung von implementierten Klassen verursachen kann!

Aufgabe 11.4 Erklärung

Eine Alternative zur Implementationsvererbung stellt **Schnittstellenvererbung** dar, wie es in Java üblich ist. Dabei werden Schnittstellen (Klassen mit ausschließlich abstrakten Methoden und ohne Attribute) definiert und nur diese vererbt. Zusätzlich gibt es Implementierungen von diesen Schnittstellen. Man würde also `Person`, `Student`, `Employee` und `StudentAssistant` in jeweils zwei Klassen aufteilen, eine Schnittstelle und eine Implementation. Die Schnittstellen würden voneinander erben, z.B. `StudentBase` von `PersonBase`, und entsprechende pur virtuelle/abstrakte Methoden

Aufgaben zur Objektorientierung in C++

wie `virtual std::string StudentBase::GetStudentID() = 0` bereitstellen. Die Implementation würde ausschließlich von der jeweiligen Schnittstelle erben (Student von StudentBase). Diese Variante erscheint zwar aufwändiger als Implementationsvererbung, vermeidet aber viele der dabei entstehenden Probleme. Schnittstellenvererbung kann in Java eingesetzt werden, um Mehrfachvererbung zu realisieren.

Aufgaben zur Objektorientierung in C++

Aufgabe 12 [O] Exceptions

Ähnlich wie in Java können Fehler während der Programmlaufzeit in C++ mittels Exceptions signalisiert werden.

```
try {
    ...
    throw <Type>;
} catch(<Type1> <param name>) {
    ...
} catch(<Type2> <param name>) {
    ...
}
...
```

Es gibt jedoch einige Unterschiede zur Fehlerbehandlung in Java. Das aus Java bekannte `finally`-Konstrukt existiert in C++ nicht. Außerdem kann jede Art von Wert geworfen werden – sowohl Objekte als auch primitive Werte wie z.B. `int`. In der Praxis wird es jedoch empfohlen, den geworfenen Wert von `std::exception` abzuleiten oder eine der existierenden Klassen aus der Standardbibliothek zu nutzen.

Im Gegensatz zu Java kann man Objekte nicht nur *by-Reference* sondern auch *by-Value* werfen und fangen. In diesem Fall wird das geworfene Objekt nach der Behandlung im `catch`-Block automatisch zerstört. Wenn es *by-Value* gefangen wird, wird das geworfene Objekt kopiert, ähnlich wie bei einem Funktionsaufruf. Beispiel:

```
// 1. Catch by value
try {
    throw C(); // create new object of class C and throw it
} catch(C c) { // catch c by value => a copy of c is created when catching
    ...
}

// 2. Catch by reference
try {
    throw C(); // create new object of class C and throw it
} catch(const C &c) { // catch c by reference, no copy is created
    ...
}
```

In der Praxis hat es sich durchgesetzt, *by-Value* zu werfen und *by-const-Reference* zu fangen.

Aufgabe 12.1 Implementierung einer Dummy-Klasse

Erstelle eine Klasse C und implementiere einen Konstruktor, einen Copy-Konstruktor und einen Destruktor. Versehe diese mit Ausgaben auf der Konsole, so dass der Lebenszyklus während der Ausführung ersichtlich wird.

Aufgabe 12.2

Experimentiere mit Exceptions. Probiere insbesondere die beiden o.g. Fälle aus und beobachte die Ausgabe. Wann wird ein Objekt erstellt/kopiert/gelöscht? Teste auch, was passiert, wenn du mehrere `catch`-Blöcke erstellst und sich diese nur in der Übergabe unterscheiden (Wert/Referenz).

```
// multiple catch blocks
try {
    throw C();
} catch(C c) {
```

Aufgaben zur Objektorientierung in C++

```
...  
} catch(const C &c) {  
    ...  
}
```

Welcher **catch** Block wird aufgerufen? Spielt die Reihenfolge eine Rolle?

Aufgabe 12.3 Erweitern der Klasse List

Füge der Klasse **List** vom Vortag (Aufgabe 7) Bereichsprüfungen hinzu. Schreibe die Methoden **insertElementAt()**, **getNthElement()** und **deleteAt()** so um, dass eine Exception geworfen wird, falls der angegebene Index die Größe der Liste überschreitet. Verwende als Exception die Klasse **std::out_of_range**¹⁰ aus dem **stdexcept** Header.

Hinweise

- Du musst hierbei keinerlei **try/catch** Block verwenden, da es rein um das werfen einer Exception geht.

Aufgabe 12.4 Testen der Implementierung

Teste die erweiterte Implementierung der Klasse **List**. Provoziere eine Exception, indem du falsche Indices angibst, und fange die Exception als **const** Referenz mit einem **catch** Block ab (s.o.). Du kannst die Methode **what()**¹¹ benutzen, um an den Nachrichtentext der Exception zu gelangen.

¹⁰ http://en.cppreference.com/w/cpp/error/out_of_range

¹¹ <http://en.cppreference.com/w/cpp/error/exception/what>

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 13 [F] Template Funktionen

Aufgabe 13.1 Templatefunktionen implementieren

Implementiere die folgende Funktion, die das Maximum von zwei Variablen liefert:

```
template<typename T>
const T &maximum(const T &t1, const T &t2);
```

Durch die Verwendung von Templates soll die Funktion mit verschiedenen Datentypen funktionieren. Teste deine Implementation.

In der Vorlesung haben wir gesehen, dass jede Verwendung von `t1` und `t2` in `maximum` eine Schnittstelle induziert, die der Typ `T` bereitstellen muss. Das bedeutet, dass `T` alle Konstruktoren, Methoden und Operatoren zur Verfügung stellen muss, die in `maximum` genutzt werden.

Wie sieht diese Schnittstelle in diesem Fall aus? Welche Gründe gibt es, den Rückgabewert der Funktion `maximum` als konstante Referenz festzulegen?

Hinweise

- In den meisten Fällen kann anstelle von `typename` auch `class` in der Template-Deklaration verwendet werden.
- In der Regel muss die Definition von Template-Funktionen und -Methoden im Header erfolgen. Allgemeiner: zur Compilezeit in der gleichen cpp-Datei wie ihre Verwendung. Das hängt damit zusammen, dass Templates sprichwörtlich nur Vorlagen sind, deren Typparameter zur Compilezeit mit den konkret verwendeten Typen ersetzt werden. Würde man Templates in separaten cpp-Dateien implementieren, dann könnte die Verbindung zwischen der Verwendungsstelle und der Definition erst zur Linkzeit hergestellt werden – also zu spät.

Aufgabe 13.2 Explizite Angabe der Typparameter

Lege nun zwei Variablen vom Typ `int` und `short` an, und versuche, mittels `maximum()` das Maximum zu bestimmen. Der Compiler wird mit der Fehlermeldung `no matching function for call...` abbrechen, da er nicht weiß, ob `int` oder `short` der Template-Parameter sein soll. Gib deshalb den Template-Parameter mittels `maximum<int>()` beim Aufruf von `maximum()` explizit an. Die übergebenen Parameter werden dabei vom Compiler automatisch in den gewünschten Typ umgewandelt.

Aufgabe 13.3 Induzierte Schnittstelle implementieren

Erstelle eine Klasse `C`, die eine Zahl als Attribut beinhaltet. Implementiere einen passenden Konstruktor sowie einen Getter für diese Zahl. Nun wollen wir unsere Funktion `maximum()` verwenden, um zu entscheiden, welches von zwei `C`-Objekten die größere Zahl beinhaltet. Überlege dir, was zu tun ist, und implementiere es.

Hinweise

- Die Klasse `C` muss mindestens die durch `maximum` induzierte Schnittstelle implementieren.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 14 [F] Generische Vektor-Implementation

Erinnere dich an die Klasse `Vector3` aus dem ersten Praktikumstag (Aufgabe 4). Diese hat den Datentyp `double` für die einzelnen Komponenten verwendet. Schreibe die Klasse so um, dass der Datentyp der Komponenten durch einen Template-Parameter angegeben werden kann. Füge dafür der Klasse `Vector3` einen Template-Parameter hinzu und ersetze jedes Auftreten von `double` mit dem Template-Parameter. Vergiss nicht, die Implementation in den Header zu verschieben, da der Compiler die Definition einer Klasse kennen muss, um beim Einsetzen des Template-Parameters den richtigen Code zu generieren.

Verbessere außerdem die Effizienz und Sauberkeit der `Vector3`-Klasse, in dem du die Parameterübergabe in den entsprechenden Methoden auf `const` Referenzen umstellst und alle Getter als `const` deklarierst.

Du weißt bereits, dass alle `template`-Funktionen und -Methoden im Header enthalten sein müssen. Um den Code trotzdem zu strukturieren, hat es sich eingebürgert, dass man die Klassendefinition in der `.hpp`-Datei hält, ohne die Methoden zu implementieren. Im Anschluss wird eine `.tpp`-Datei inkludiert, die die Implementierung der Methoden und Funktionen enthält. Der Aufbau der Datei `Vector3.hpp` wäre also wie folgt:

```
#ifndef VECTOR3_HPP_
#define VECTOR3_HPP_

/**
* Don't forget documentation!
*/
template<typename T>
class Vector3 {
public:
    // Method declarations

    // You need to use a different template type if you want to declare the
    // overloaded output operator as friend.
    // Otherwise, you can introduce getters for the vector attributes.
    // Then there is no need to use a friend declaration.
    template<typename X>
    friend std::ostream& operator<<(std::ostream& out, const Vector3<X> rhs);

private:
    // Attributes
};

// function declarations only

#include "Vector3.tpp" // contains method and function definitions

#endif /* VECTOR3_HPP_ */
```

Hinweise

- Die Datei `Vector3.tpp` ist nicht vorgegeben, du musst diese selbst erstellen!
- Auch wenn bei reinen Template-Klassen die `.cpp`-Datei leer bleibt, ist es sinnvoll, eine solche anzulegen. Dadurch wird das Template garantiert auf Syntaxfehler überprüft. Der Inhalt der `.cpp`-Datei ist in dieser Aufgabe schlicht `#include "Vector3.hpp"`.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 15 [F] Generische Verkettete Liste

Aufgabe 15.1

Schreibe die Klassen `List`, `ListItem` und `ListIterator` aus dem zweiten Praktikumstag so um, dass man den Typen der in der Liste gespeicherten Elemente über einen Template-Parameter angeben kann.

Dazu müssen einige Änderungen gemacht werden. Zum einen sollte der Inhalt eines Elements beim Erstellen nicht als Wert sondern als `const` Referenz übergeben werden. Zum anderen sollten die Methoden zum Löschen von Elementen `void` zurückgeben, und nicht mehr das jeweilige gelöschte Element. Der Grund dafür ist, dass in diesem Fall eine temporäre Kopie des Elements gemacht werden müsste, ohne dass es der Benutzer beeinflussen kann. Je nach Elementtyp können solche Kopien problematisch und unerwünscht sein.

Hinweise

- Arbeitet die Klassen nacheinander ab, beginnend bei `ListItem`.
- Stelle sicher, dass man eine Klasse fehlerfrei kompilieren kann, bevor du zur nächsten übergehst.
- Denke daran, dass du auch hier die Implementation in eigene *.tpp-Dateien verschieben musst.

Aufgabe 15.2

Überlade den `operator<<`, sodass Listen direkt über ein `std::ostream` wie z.B. `std::cout` ausgegeben werden können.

Aufgabe 15.3

Teste deine Implementierung. Probiere auch Folgendes aus und beobachte die Ausgabe.

```
List<List<int> > list;
list.appendElement(List<int>());
list.getFirst().appendElement(1);
list.getFirst().appendElement(2);
list.appendElement(List<int>());
list.getLast().appendElement(3);
list.appendElement(List<int>());
list.getLast().appendElement(4);
list.getLast().appendElement(5);
std::cout << list << std::endl;
```

Hinweise

- In der ersten Zeile ist absichtlich ein Leerzeichen zwischen den beiden schließenden spitzen Klammern. Bis hin zu C++11 konnte der C++-Compiler nicht erkennen, ob es sich bei `>>` um den Operator oder um geschachtelte Templates handelt. Seit C++11 ist es nicht mehr nötig, ein Leerzeichen zwischen die beiden schließenden spitzen Klammern einzufügen.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 16 [F] Standard-Container

In dieser Aufgabe werden wir den Umgang mit den Containern `std::vector` und `std::list` aus der Standard Template Library üben. Es ist sinnvoll, wenn du während der Übung eine C++-Referenz zum Nachschlagen bereithältst, z.B. <http://www.cplusplus.com/>. Schaue dir auch die Vorlesungsfolien genau an, da diese nützliche Codebeispiele enthalten.

Die Klasse `std::list` stellt eine verkettete Liste dar, bei der man an beliebiger Stelle Elemente effizient löschen und hinzufügen kann. `std::vector` stellt ähnliche Funktionen bereit, allerdings liegen hier die Elemente in einem einzigen, zusammenhängenden Speicherbereich, der neu alloziert und kopiert werden muss, wenn seine aktuelle Kapazität überschritten wird. Auch müssen viele Elemente verschoben werden, wenn der Vektor in der Mitte oder am Anfang modifiziert wird. Der große Vorteil von `std::vector` ist der *wahlfreie Zugriff*, d.h. man kann auf beliebige Elemente mit konstantem Aufwand zugreifen.

- a) Schreibe zunächst eine Funktion `template<typename T> void print(const T &t)`, die beliebige Standardcontainer auf die Konsole ausgeben kann, die Integer speichern und Iteratoren unterstützen. Nutze dazu die Funktion `copy()` sowie die Klasse `std::ostream_iterator<int>`, um den entsprechenden OutputIterator zu erzeugen.
- b) Lege ein `int`-Array an und initialisiere es mit den Zahlen 1 bis 5. Lege nun einen `std::vector<int>` an und initialisiere ihn mit den Zahlen aus dem Array.
- c) Lege eine Liste `std::list<int>` an und initialisiere diese mit dem zweiten bis vierten Element des Vektors. **Tipp:** Du kannst auf Iteratoren eines Vektors (genauso wie auf Zeiger) Zahlen addieren, um diese zu verschieben.
- d) Füge mittels `std::list<T>::insert()` das letzte Element des Vektors an den Anfang der Liste hinzu.
- e) Lösche alle Elemente des Vektors mit einem einzigen Methodenaufruf.
- f) Mittels `remove_copy_if()` kann man Elemente aus einem Container in einen anderen kopieren und dabei bestimmte Elemente löschen lassen. Nutze diese Funktion, um alle Elemente, die kleiner sind als 4, aus der Liste in den Vektor zu kopieren. Beachte, dass `remove_copy_if()` keine neuen Elemente an den Container anhängt, sondern lediglich Elemente von der einen Stelle zur anderen elementweise durch Erhöhen des OutputIterator kopiert.

Deshalb kannst du `vec.end()` **nicht** als OutputIterator nehmen, da dieser "hinter" das letzte Element zeigt und weder dereferenziert noch inkrementiert werden darf. Nutze stattdessen die Methode `back_inserter()`, um einen Iterator zu erzeugen, der neue Elemente an den Vektor anhängen kann.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 17 [F] Funktionales Programmieren

In dieser Aufgabe werden Funktionen aus der funktionalen Programmierung vorgestellt. Diese sind `map`, `filter` und `reduce`. Der Ablauf ist wie folgt:

- In Aufgabe 17.1 werden erst einmal die Funktionsweisen der zu implementierenden Funktionen `map`, `filter` und `reduce` vorgestellt.
- In Aufgabe 17.2 wirst du diese Funktionen und zusätzliche Hilfsfunktionen implementieren.
- In Aufgabe 17.3 wirst du deine Hilfsfunktionen als *Funktoren* implementieren und `map`, `filter` und `reduce` entsprechend anpassen.
- In Aufgabe 17.4 wirst du den Code auf Templates umstellen, damit Funktionen und Funktoren austauschbar verwendet werden können.
- In Aufgabe 17.5 wirst du zusätzlich mit Methodenzeigern arbeiten.

Aufgabe 17.1 Erklärung `map`, `filter` und `reduce`

Arbeitet man auf iterierbaren Sequenzen, ist dies fast immer mit Schleifen über die Sequenz verbunden. Die drei Funktionen `map`, `filter` und `reduce` vereinfachen uns hierbei die Arbeit. Hierzu ein Beispiel: Haben wir einen Vektor des Typs `double` und wollen jedes Element quadrieren, endet dies meist in dem folgenden Programmcode:

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

for (std::vector<double>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    *it = square(*it); // squaring element
}

// numbers now [ 1, 4, 9, 16, 25 ]
```

map

Die Idee von der Funktion `map` ist es, genau dies zu vereinfachen. Die Funktion erhält folgende Parameter:

- Die Start- und Enditeratoren der zu modifizierenden Sequenz
- Einen Iterator, der auf eine Sequenz zeigt, in der die veränderten Elemente gespeichert werden sollen
- Einen Funktionszeiger, der auf eine Funktion zeigt, die für jedes Element der iterierbaren Sequenz aufgerufen werden soll

Ein Beispiel siehst du in folgendem Listing:

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

map(numbers.begin(), numbers.end(), numbers.begin(), square);
// numbers now [ 1, 4, 9, 16, 25 ]
```

filter

Die Funktion `filter` funktioniert analog, indem sie einen Zeiger auf eine Funktion erhält, die einen Listenelementtyp erwartet und ein Booleschen Wert (`bool`) zurückgibt. Auf alle Elemente wird diese Funktion aufgerufen und alle Elemente, für die die Funktion `true` zurückgibt, werden in die Ausgabesequenz kopiert. Der Rest wird entfernt.

Aufgaben zu fortgeschrittenen Themen in C++

```
// #include <iterator>

std::vector<double> numbers = { 1, 2, 3, 4, 5 };
std::vector<double> filteredNumbers;

filter_funcpointer(numbers.begin(), numbers.end(),
                   std::back_inserter(filteredNumbers), is_odd);

// filteredNumbers now [ 1, 3, 5 ]
```

Die Besonderheit hier ist, dass wir eine zweite Liste (`filteredNumbers`) benötigen, um das Ergebnis zu speichern, da die Ergebnisliste in der Regel kürzer sein wird als die Eingabeliste. Eine andere Lösung wäre, den letzten Stand des Ausgabeiterators zurückzugeben und die Liste entsprechend einzukürzen.

Zusätzlich zur Ausgabeliste verwenden wir hier einen `std::back_insert_iterator`¹², der die ihm geordnete Liste immer dann vergrößert, wenn ein neues Element in den Iterator hineingeschrieben wird. Die Hilfsfunktion `std::back_inserter`¹³ vereinfacht die Erzeugung des Iterators. Sowohl die Klasse `std::back_insert_iterator` als auch die Hilfsfunktion `std::back_inserter` befinden sich im Header `iterator`.

reduce

Die Aufgabe der Funktion `reduce` ist es, eine Sequenz zu einem einzelnen Element zusammenzuschrumpfen. Hierbei wird der Ausgabeiterator gegen einen Startwert ausgetauscht. Hier ein Beispiel, bei dem die Summe über die Elemente in `numbers` gebildet wird.

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

std::cout << reduce(numbers.begin(), numbers.end(), 0.0, sum) << std::endl; // 15
```

Aufgabe 17.2 Programmieren der Funktionen

Du wirst nun die drei Funktionen `map`, `filter` und `reduce` nachprogrammieren. Hierbei geht es erstmal darum, ein funktionierendes Gerüst zu erstellen, anstatt perfekt generische Algorithmen zu erhalten.

Aufgabe 17.2.1 map

Schreibe eine Funktion `map` die folgende Signatur besitzt.

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, double(*func)(double d));
```

Hierbei ist der letzte Parameter der Funktionszeiger. Die Klammern um `*func` sind notwendig, damit der Compiler den übergebenen Parameter als Funktionszeiger einer Funktion mit Rückgabewert `double` interpretiert und nicht als Funktion mit Rückgabewert `double *`¹⁴. Diese Funktion hat zusätzlich noch ein `double d` als Parameter.

Du kannst dich bei der Implementierung von `map` an dem Schleifengerüst zu Anfang von Aufgabe 17.1 orientieren.

¹² http://en.cppreference.com/w/cpp/iterator/back_insert_iterator

¹³ http://en.cppreference.com/w/cpp/iterator/back_inserter

¹⁴ Welche folgende Signatur hätte: `double *(*func)(double d)`

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 17.2.2 filter

Die von dir zu schreibende Funktion `filter` soll der folgenden Signatur folgen.

```
template<typename InIt, typename OutIt>
OutIt filter(InIt first, InIt last, OutIt out_first, bool(*pred)(int i));
```

Die Implementierung von `filter` wird sehr ähnlich zur Implementierung von `map` aussehen. Der Hauptunterschied ist, dass `pred` in einer `if`-Bedingung eingesetzt werden muss, um zu entscheiden, ob der aktuelle Wert in den Ausgabeiterator (`out_first`) geschrieben werden soll.

Aufgabe 17.2.3 reduce

Erstelle eine Funktion `reduce`, die der folgenden Signatur folgt.

```
template<typename InIt, typename RetT>
RetT reduce(InIt first, InIt last, RetT initialValue, RetT(*func)(RetT i, double j));
```

Hierbei muss ein passender initialer Wert übergeben werden, der mit dem Rückgabewert und dem ersten Argument der übergebenen Funktion zusammenpasst.

Aufgabe 17.2.4 Hilfsfunktionen implementieren

Implementiere in dieser Aufgabe drei Hilfsfunktionen, die den Anforderungen der jeweiligen Signaturen der Funktionszeiger in den Funktionen `map`, `filter` und `reduce` folgen. Du kannst dir dabei gerne eigene Funktionen ausdenken oder dich an die Funktionen in den Beispielen halten (bspw. `square` für `map`, `isOdd` für `filter`, `sum` für `reduce`).

Teste anschließend deine Implementierungen mithilfe deiner Hilfsfunktionen.

Aufgabe 17.3 Funktoren

Es gibt außerdem noch die Möglichkeit, Funktionen in einem Funktionsobjekt (*Funktior*) zu kapseln. Dabei überlädt man den Operator `operator()` der Funktior-Klasse, welcher eine bestimmte Funktion ausführt. Schaut man sich in unserem Beispiel die Funktion `square` mit der Definition `double square(double i);` an, würde der Funktior folgendermaßen aussehen:

```
class Square {
public:
    double operator()(double i) { return i*i; }
};
```

Die Funktion `map` würde wie folgt umgeschrieben werden müssen, um den Funktior zu akzeptieren:

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, Square s);
```

Erstelle für jede deiner Hilfsfunktionen eine Funktior-Klasse und füge neue Implementierungen für `map`, `filter` und `reduce` hinzu, die mit den Funktoren kompatibel sind. Vergleiche die Ausgabe der Funktionszeiger- und Funktoren-basierten Implementierungen.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 17.4 Verwendung von Templates

Die derzeitige Implementierung funktioniert entweder mit Funktoren einer bestimmten Klasse oder mit Funktionszeigern, die einem bestimmten Typen angehören, der durch die Signatur der Funktion festgelegt ist. Diese Verdoppelung des Codes ist unschön und um das Problem zu lösen, kann man die Funktionszeiger-/Funktiorvariable durch einen Templateparameter ersetzen. Damit ist der Parametertyp flexibel; der Nachteil ist, dass nur noch aus der eigentlichen Implementierung der Funktion hervorgeht, was der Typ des Templateparameters anbietet muss (*Induzierte Schnittstelle*).

Erstelle Varianten der Funktionen `map`, `filter` und `reduce`, deinen weiteren Templateparameter angeben, der flexibel mit Funktionszeigern oder Funktoren belegt werden kann.

```
template <typename InIt, typename OutIt, typename ...>
```

Vergleiche die Ausgabe deiner Template-basierten Lösung mit den Ergebnissen der Funktionszeiger- und Funktior-basierten Lösungen.

Aufgabe 17.5 Methodenzeiger

Es gibt auch die Möglichkeit, Methoden¹⁵ via Zeiger auszuführen (sogenannte **Methodenzeiger**). Hierzu muss man zusätzlich zu dem Funktionsnamen noch ein Objekt übergeben, auf das die Methode angewendet wird.

In unserem Beispiel von Square fügen wir noch eine weitere Methode `squareroot` hinzu, welche das Inverse der Quadratur ausführt. Unsere Klasse Square verändert sich dementsprechend zu

```
class Square {
public:
    double operator() (double i)
    double squareroot(double i)
};
```

und unser `map` zu

```
template <typename InIt, typename OutIt, typename ObjType>
OutIt map(InIt first, InIt last, OutIt out_first, ObjType *object, double (ObjType::* method)(
    double));
```

Deine Aufgabe ist es nun, eine neue Implementation von `map` hinzuzufügen, deinem für `map` geschriebenen Funktor eine weitere Methode hinzuzufügen und anschließend deine Implementation mit allen implementierten Methoden zu testen.

Nachwort zu dieser Aufgabe

Für produktive C++-Programme bietet die Standardbibliothek fertige Funktionen und Klassen, um die gerade erlerten Prinzipien dieser Aufgabe zu realisieren, z.B. `std::function<...>`¹⁶ und `std::bind()`¹⁷. Diese können mit einer beliebigen Anzahl von Parametern umgehen und beinhalten viele weitere Features.

¹⁵ Hier ist eine einfache Erklärung zu dem Unterschied von Funktion und Methode zu finden <http://stackoverflow.com/a/155655>

¹⁶ <http://en.cppreference.com/w/cpp/utility/functional/function>

¹⁷ <http://en.cppreference.com/w/cpp/utility/functional/bind>

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 18 [F] Callbacks

Motivation für Callbacks

In dieser Aufgabe werden mehrere Methoden zur Realisierung von Callbacks in C++ vorgestellt und implementiert. Callbacks können als Alternative zum Observer Pattern¹⁸ eingesetzt werden. Beispielsweise kann man einem GUI-Button eine Callback-Funktion übergeben, die aufgerufen werden soll, sobald der Button gedrückt wird. Wir werden Callbacks dazu verwenden, um den Benutzer bei jedem Schritt eines laufenden Algorithmus über den aktuellen Fortschritt zu informieren.

Aufgabe 18.1 Basisalgorithmus

Implementiere folgenden Algorithmus, der das Problem der Türme von Hanoi löst.¹⁹

```
funktion hanoi (Number i, Pile a, Pile b, Pile c)
if i > 0 then
    hanoi(i-1, a, c, b); // Move i-1 slices from pile "a" to "b"
    Move slice from "a" to "c";
    hanoi(i-1, b, a, c); // Move i-1 slices from pile "b" to "c"
end
```

Du brauchst keine Türme zu modellieren und zu verschieben. Es reicht, lediglich die Schritte auf der Konsole auszugeben. Bei einem Aufruf von `hanoi(3, 1, 2, 3)` soll folgende Ausgabe erfolgen:

```
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
```

Aufgabe 18.2 Callbacks mit Funktionszeigern

Nun wollen wir die fest einprogrammierte Ausgabe durch ein Callback ersetzen. Dadurch wird es möglich, die Funktion auszutauschen und z.B. eine graphische Ausgabe zu implementieren, ohne jedoch den Algorithmus selbst zu ändern.

Eine simple Art des Callbacks, die auch in C verfügbar ist, ist die Übergabe eines Funktionszeigers, der die Adresse der aufzurufenden Funktion beinhaltet. Ändere deine Implementation entsprechend um:

```
void hanoi(int i, int a, int b, int c, void(*callback)(int from, int to)) {
    ...
    callback(a, c);
    ...
}
```

Nun können wir eine Funktion mit zwei Parametern an `hanoi()` übergeben.

```
void print(int from, int to) {
    cout << from << " -> " << to << endl;
}
...
hanoi(3, 1, 2, 3, print);
```

¹⁸ http://de.wikipedia.org/wiki/Observer_Pattern

¹⁹ http://de.wikipedia.org/wiki/Turm_von_Hanoi

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 18.3 Callbacks mit Funktoren

Ein Nachteil der vorherigen Implementation ist, dass nur reine Funktionen als Callback übergeben werden können. Eine Möglichkeit dies zu umgehen ist die Verwendung von Templates. Der Callback-Typ wird dabei durch einen Template-Parameter spezifiziert:

```
template<typename T>
void hanoi(int i, int a, int b, int c, T callback) {...}
```

Dadurch kann an `hanoi()` fast alles übergeben werden, was sich syntaktisch mittels

```
callback(a, c);
```

aufrufen lässt, also auch Objekte, bei denen der ()-Operator überladen ist (sog. *Funktoren*²⁰). Dabei müssen nicht einmal die Parametertypen (`int`) exakt übereinstimmen, solange eine implizite Umwandlung durch den Compiler möglich ist.

Teste deine Implementation mit einem Funktor. Schreibe dafür eine einfache Klasse und überlade deren `operator()`:

```
void operator()(int from, int to);
```

Aufgabe 18.4 Callbacks mit Callback-Klasse

Probleme der bisherigen Implementation

Die Verwendung von Templates hat uns zwar eine sehr flexible und syntaktisch ansprechende Möglichkeit für Callbacks geliefert, beherbergt jedoch mehrere, teils gravierende, Schattenseiten.

Zum einen ist es dadurch immer noch nicht möglich, beliebige Methoden einer Klasse als Callback zu übergeben. Durch Methodencallbacks könnten Klassen mehrere unabhängige Callback-Methoden besitzen. Zum anderen ist `hanoi` nun an den Callback-Typ **gekoppelt**. Wenn wir also `hanoi` selbst an eine Funktion/Methode übergeben wollen, muss der Callback-Typ bei der Übergabe mit angegeben werden und zerstört somit die Unabhängigkeit der Funktion von ihrem Callback. Dies kann sich insbesondere bei komplexeren Anwendungen von Callbacks sehr negativ widerspiegeln. Stell dir vor, du hättest ein GUI-Framework mit verschiedenen Elementen, die Callbacks nutzen, z.B. Buttons. Dann wäre die Button-Klasse ebenfalls an den Callback-Typ gekoppelt. Immer wenn ein Button als Parameter an eine Funktion übergeben wird, müsste diese Funktion den Callbacktyp ebenfalls als Template-Parameter entgegennehmen:

```
template<typename T>
void doSomethingWithButton(Button<T> &btn);
```

Dieser Stil würde sich durch das gesamte Framework ziehen, und sowohl den Entwicklungsaufwand als auch die Verständlichkeit beeinträchtigen. Ein weiterer Nachteil wäre, dass der Callback-Typ bereits zur Kompilierzeit festgelegt werden müsste und es unmöglich wäre, diesen während der Laufzeit zu ändern.

Lösung mittels Callback-Klasse

Deshalb werden wir eine Klasse schreiben, die beliebige Callbacks kapseln kann (`Callback`), und nach außen hin allein von den Übergabeparametern des Callbacks abhängig ist. Ziel ist es, folgendes zu ermöglichen:

```
void hanoi(..., Callback callback) {
    ...
    callback(a, c);
    ...
}
```

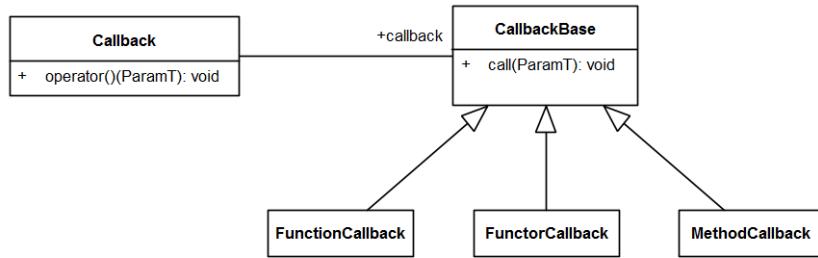
²⁰ <https://de.wikipedia.org/wiki/Funktionsobjekt>

Aufgaben zu fortgeschrittenen Themen in C++

```
}

...
hanoi(..., Callback(print)); // function callback
hanoi(..., Callback(c)); // functor callback
hanoi(..., Callback(&C::print, &c)); // method callback
```

Die Idee dahinter ist Folgende: Wir definieren eine abstrakte Klasse `CallbackBase`, die eine abstrakte Methode `void call()` enthält. Für jeden Callback-Typ (Funktionszeiger, Funktor und Methodenzeiger) wird eine Unterklasse erstellt, die `call()` entsprechend reimplementiert.



CallbackBase

Fange mit der Klasse `CallbackBase` an. Damit man beim Aufrufen des Callbacks einen Parameter übergeben kann, füge `call()` einen Parameter vom Typ `ParamT` hinzu, wobei `ParamT` ein Template-Parameter von `CallbackBase` sein soll. Der Klassenrumpf lautet also

```
template<typename ParamT>
class CallbackBase {
public:
    ...
    virtual void call(ParamT t) = 0;
};
```

Falls ein Callback eigentlich mehrere Parameter erfordert, müssen diese entsprechend in ein Containerobjekt gepackt werden. Generische Callback-Wrapper mit variabler Parameteranzahl sind zwar möglich, würden aber den Rahmen dieses Praktikums sprengen.

Hinweise

- Du kannst diese und alle nachfolgenden Klassen in einem einzigen Header implementieren, weil die Klassen sehr kurz sind und außerdem semantisch stark zusammenhängen.

Aufgabe 18.5 Klasse FunctionCallback

Implementiere nun die erste Unterklasse `template<typename ParamT> FunctionCallback`, die von `CallbackBase<ParamT>` erbt. `FunctionCallback` soll einen entsprechenden Funktionszeiger als Attribut besitzen, der bei der Konstruktion initialisiert wird. Ebenso soll `call(ParamT t)` implementiert werden, in der der gespeicherte Funktionszeiger mit dem gegebenen Argument aufgerufen wird.

Aufgaben zu fortgeschrittenen Themen in C++

Teste deine Implementation. Lasse `hanoi()` einen Zeiger auf `CallbackBase` erwarten, übergebe aber die Adresse eines `FunctionCallback` Objektes. Du kannst folgende Vorlage verwenden:

```
#include<utility>
typedef std::pair<int, int> intpair;

void hanoi(..., CallbackBase<intpair> *callback) {
    // ...
    callback->call(intpair(a, c));
    // ...
}

int main() {
    // ...
    CallbackBase<intpair> *function =
        new FunctionCallback<intpair>(printMovePair);
    hanoi(3,1,2,3, function);
    // ...
}
```

Aufgabe 18.6 Klasse FunctorCallback

Implementiere nun die Unterklasse `template<typename ParamT, typename ClassT> FunctorCallback`. Zusätzlich zum Parameter-Typ muss hier auch der Typ der Funktorklasse angegeben werden. Speichere das zu verwendende Funktorko-Objekt als Referenz ab, um Kopien zu vermeiden. Achte auch im Konstruktor darauf, dass keine Kopien des Funktors gemacht werden. Teste deine Implementation!

Aufgabe 18.7 Klasse MethodCallback

Implementiere nun die letzte Unterklasse `template<typename ParamT, typename ClassT> MethodCallback`. Beachte, dass nun zwei Attribute nötig sind - ein Methodenzeiger und ein Zeiger auf das zu verwendende Objekt. Teste deine Implementation.

Hinweise

- Verwende beispielsweise folgende Signatur für den Konstruktor von `MethodCallback`: `MethodCallback(void(ClassT::*method)(ParamT), ClassT *object)`
- Gegeben einen Zeiger `object` auf ein Objekt, einen Zeiger `method` auf eines seiner Methoden und einen Parameter `p` für die Methode, sieht ein Aufruf von `method` wie folgt aus: `(object->*method)(p);`

Aufgabe 18.8 Klasse Callback

Wir haben jetzt den Typ des Callbacks vollständig von seiner Verwendung entkoppelt. Jedoch muss ein Callback-Objekt per Zeiger/Referenz übergeben werden, sodass das dir schon bekannte Problem der Zuständigkeit für die Zerstörung eines Objekts entsteht. Außerdem muss man beim Erstellen eines Callbacks explizit den Typ der Unterklasse angeben. Es wäre also sinnvoll, einen entsprechenden Wrapper zu schreiben, der sich um die Speicherverwaltung von Callbacks kümmert und bei der Konstruktion die passende Unterklasse selbst aussucht.

Schreibe eine Klasse `template<typename ParamT> Callback`, die einen Smart Pointer auf ein `CallbackBase`-Objekt als Attribut hat. Der Smart Pointer soll die Speicherverwaltung übernehmen. Überlade den `operator()`, der den Aufruf einfach an das `CallbackBase`-Objekt hinter dem Smart Pointer weiterleitet.

Aufgaben zu fortgeschrittenen Themen in C++

Implementiere nun für jede Callback-Art je einen Konstruktor, der eine Instanz der entsprechenden UnterkLASSE erzeugt und im Smart Pointer speichert. Der erste Konstruktor soll also einen Funktionszeiger entgegennehmen und ein FunctionCallback instantiiieren. Der zweite Konstruktor soll eine Referenz auf ein Funktor-Objekt erwarten und FunctorCallback instantiiieren, und der dritte entsprechend ein MethodCallback. Beachte, dass die beiden letztgenannten Konstruktoren selbst Template-Methoden sind, da die Callback-KLASSE nur an den Parameter-Typ gekoppelt ist.

Teste deine Implementation in Zusammenhang mit der hanoi-Funktion. Du kannst das Callback-Objekt auch per Wert übergeben, da intern nur Zeiger kopiert werden.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 19 [F] Eigene Arrays (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Nachdem du bei unseren Übungen zu Arrays gesehen hast, dass es störend ist, wenn man die Größe eines Arrays immer getrennt zu den gespeicherten Daten verwalten muss, ist ein sinnvoller Schritt, eine eigene Array-Klasse zu implementieren, die Daten und Größe des Arrays zusammen speichert.

Eine möglicher Anwendungsfall sieht so aus:

```
#include "Array.h"
#include <iostream>
#include <string>

template<typename T>
void printFirst(const Array<T> &array) {
    std::cout << array[0] << std::endl;
}

int main() {
    Array<std::string> stringArray(10);
    stringArray[0] = "Hello World";
    printFirst(stringArray);
}
```

Hinweise

- Überlege dir, welche Operatoren/Methoden das obige Code-Beispiel von Array verlangt. Unter anderem musst du jeweils einen `const` und einen nicht-`const operator[]` implementieren.
- Du kannst auch Exceptions (z.B. `std::out_of_range` aus `<stdexcept>`) verwenden, um falsche Indices korrekt abzufangen.
- Eine fortgeschrittene Übung ist es, Iteratoren oder `operator+(unsigned int)` für Array bereitzustellen, sodass du z.B. die Funktion `std::copy` aus der Standardbibliothek verwenden kannst, um ein Array zu kopieren:

```
#include <algorithm> // copy
#include <iterator> // back_inserter
#include <vector>
// ...
Array<int> array(10);
std::vector<int> vector;
std::copy(array, array + 4, std::back_inserter(vector));
```

- Denke daran, für Indizes den Typ `size_t` zu verwenden.
- Diese Idee ist natürlich nicht neu. Seit C++11 gibt es eine Array-Implementation in der C++-Standardbibliothek (`std::array`²¹). Du findest die gleiche Klasse auch als `boost::array` in Boost.

²¹ http://www.boost.org/doc/libs/1_55_0/doc/html/array.html

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 20 [F] Makefiles

In dieser Übung erkunden wir, wie man Makefile-Projekte in CodeLite aufbaut. Wir bauen dafür einen Teil unseres Aufzugsimulators nach, um zu sehen, wo die Herausforderungen in echten Projekten mit Makefiles gelöst werden können.

Das Programm *make*²² wird in großen Softwareprojekten verwendet, um Programme zu kompilieren. Makefiles geben *make* dabei Informationen wie das Programm gelinkt und compiliert werden muss. Außerdem kann es verwendet werden, um weitere Nebenaufgaben während der verschiedenen Kompilationsphasen zu definieren, wie zum Beispiel das automatische Löschen der später unnötigen Kompilationsdateien. All das werden wir in dieser Aufgabe ausprobieren.

Aufgabe 20.1 Projekt anlegen:

Wähle *Workspace* → *New project* und wähle als Projekttyp **CPPP/C++ Projekt**.

Das erzeugte Projekt enthält bereits eine Datei **Makefile** (im Ordner **resources**) und eine **main.cpp**.

In dieser Übung wollen wir unser eigenes Makefile erstellen. Öffne dafür die Datei **Makefile** und lösche ihren Inhalt.

Make erwartet, dass das Makefile ein Target mit dem Namen **all** hat. Um unser Projekt zu testen, geben wir zunächst eine einfache Meldung auf der Kommandozeile aus. Lege nun das Target **all** an und füge den folgenden Befehl an (Vergiss dabei nicht den Tab vor jedem Befehl!):

```
all:  
    @echo "Running all..."
```

Wenn du jetzt *Build* aufrufst, sollte in der Konsole in etwa Folgendes erscheinen:

```
-----Building project:[ mkttest - Debug ]-----  
Running all...  
====0 errors, 0 warnings====
```

Aufgabe 20.2 Erster Kompiliervorgang

Jetzt ist es an der Zeit, ein Programm mittels *make* zu kompilieren. Lege dazu eine C++-Sourcedatei **main.cpp** mit einer **main**-Funktion an, die etwas sinnvolles ausgibt.

Entgegen unserer bisherigen Erfahrung musst du nun manuell im **Makefile** eintragen, dass **main.cpp** gebaut werden soll. Ersetze die Dummy-Ausgabe daher durch einen Compiler-Aufruf an **g++**:

```
all:  
    g++ -o main.exe main.cpp
```

Wenn du jetzt *Build* aufrufst, wird dein Programm kompiliert und als **main.exe** im Projekthauptverzeichnis abgelegt.

Hinweise

- Die Option **-o** für **g++** erwartet als ersten Parameter den Namen der gewünschten Zielfile.

²² Dokumentation <https://www.gnu.org/software/make/manual/make.html>

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 20.3 Klasse Building:

Jetzt fügen wir die Klasse Building zu unserem Projekt hinzu, die allerdings nur minimale Funktionalität bietet.

Building.hpp:

```
#pragma once
#include <string>

class Building {
public:
    Building(unsigned int numFloors);
    const std::string toString() const;
private:
    unsigned int numFloors;
};
```

Building.cpp:

```
#include "Building.hpp"
#include <iostream>

Building::Building(unsigned int numFloors):
    numFloors(numFloors) {}

const std::string Building::toString() const{
    std::stringstream output;
    output << "A building with " << numFloors;
    output << " floors" << std::endl;
    return output.str();
}
```

Erzeuge in der main-Funktion eine zweistöckige Instanz von Building und gib diese mittels Building::toString auf der Konsole aus. Damit das Projekt kompiliert, muss auch Building im Makefile eingetragen werden. Passe dazu den Kompileraufruf an:

```
all:
    g++ -o main.exe main.cpp Building.cpp
```

Wenn du das Projekt gebaut hast und ausführst, sollte auf der Konsole eine Ausgabe deines Gebäudes erscheinen.

Aufgabe 20.4 Compiler-Aufrufe auslagern:

In der Vorlesung haben wir gesehen, dass *make* anhand der Zeitstempel von Dateien dazu in der Lage ist, zu erkennen, wann ein Programmteil neu gebaut werden muss. Aktuell nutzen wir diese Möglichkeit noch nicht: Egal ob wir *main.cpp*, *Building.cpp* oder *Building.h* verändert haben, immer wird das gesamte Projekt neu gebaut. In diesem Schritt zerlegen wir die Abhängigkeiten zu den einzelnen Dateien.

Mache das Target all jetzt abhängig von den Objektdateien *main.o* und *Building.o* und erzeuge für jede Objektdatei ein eigenes Ziel, welches diese baut (Das Flag *-c* sorgt dafür, dass die Sourcedateien nur kompiliert, aber nicht gelinkt werden).

```
all: main.o Building.o
g++ -o main.exe main.o Building.o
```

Aufgaben zu fortgeschrittenen Themen in C++

```
main.o: main.cpp  
g++ -c -o main.o main.cpp  
  
Building.o: Building.cpp  
g++ -c -o Building.o Building.cpp
```

Baue das Projekt nun erneut, du solltest drei Aufrufe von g++ sehen:

```
make all  
g++ -c -o main.o main.cpp  
g++ -c -o Building.o Building.cpp  
g++ -o main.exe main.o Building.o
```

Baust du das Projekt nun ein weiteres Mal, so wird nur noch der Linker aufgerufen:

```
make all  
g++ -o main.exe main.o Building.o
```

Aufgabe 20.5 Linker-Aufruf auslagern

Wie können wir diesen an sich unnötigen Aufruf ebenfalls noch loswerden? Eine Lösung ist es, das Target all von main.exe abhängig zu machen und ein neues Ziel main.exe zu definieren:

```
all: main.exe  
  
main.exe: main.o Building.o  
g++ -o main.exe main.o Building.o  
  
# ...
```

Wenn du das Projekt jetzt baust, erhältst du erfreulicherweise die Rückmeldung, dass nichts zu tun ist:

```
make all  
make: Nothing to be done for 'all'.
```

Aufgabe 20.6 Inkrementelles Bauen:

Wir erproben jetzt, wie sich Veränderungen an einer der drei Dateien auf die Ausführung von make auswirken. Mache nacheinander kleine Änderungen – das können auch Kommentare sein – an den Dateien main.cpp, Building.h und Building.cpp und baue das Projekt nach jeder Änderung.

Dir fällt auf, dass Änderungen an Building.h von make nicht bemerkt werden; die Datei taucht ja nirgendwo explizit im Makefile auf.

Wir sehen uns jetzt an, welche Tragweite dieses Problem haben kann.

Aufgaben zu fortgeschrittenen Themen in C++

Aufgabe 20.7 Header als Abhängigkeiten

Du hast kennengelernt, dass man Implementierungen auch `inline` in einem Header machen kann, zum Beispiel wenn diese klein sind.

Bewege `toString` nun nach `Building.h`:

```
#include <iostream>
// ...
const std::string toString() const{

    std::stringstream output;
    output << "A building with " << this->numFloors << " floors" << std::endl;
    return output.str();
}
```

Bau das Projekt; es kompiliert nicht! Warum? Genau aus dem Grund, dass `make` das Header-File nicht „kennt“. Jetzt gibt es im Projekt keine Definition von `toString` wie uns der Linker auch mitteilt:

```
main.o:main.cpp:(.text+0x5c): undefined reference to 'Building::toString() const'
collect2: ld returned 1 exit status
```

Das Problem lässt sich lösen, indem wir im Makefile angeben, dass `main.o` nicht nur abhängig von `Building.cpp`, sondern auch von `Building.h` ist:

```
# ...
main.o: main.cpp Building.h
    g++ -c -o main.o main.cpp
# ...
```

Ist das eine schöne Lösung? Sicherlich nicht, denn ab sofort müssten wir manuell alle Header ins Makefile eintragen, die wir per `#include` in eine Sourcedatei einbinden. Schlimmer noch: Wir müssten über rekursive Inkludierungen Bescheid wissen, z.B. wenn `Building.h` einen anderen veränderlichen Header wie `Floor.h` einbindet.

Glücklicherweise hilft uns `g++` bei diesem Problem.

Aufgabe 20.8 Header automatisch als Abhängigkeiten deklarieren:

Wir automatisieren jetzt die Erkennung von Headern als Abhängigkeiten. Lösche dazu die Abhängigkeit `Building.h` des Targets `main.o` und füge in den Compiler-Aufrufen die Parameter `-MMD -MP` hinzu. Binde außerdem die Dateien `Building.d` und `main.d` ein wie unten dargestellt:

```
all: main.exe

main.exe: main.o Building.o
        g++ -c -o main.exe main.o Building.o

main.o: main.cpp
        g++ -c -MMD -MP -o main.o main.cpp

Building.o: Building.cpp
        g++ -c -MMD -MP -o Building.o Building.cpp

#include Building.d main.d
```

Aufgaben zu fortgeschrittenen Themen in C++

Um den Effekt dieser Lösung zu sehen, müssen wir alle generierten Dateien löschen (`main.exe`, `main.o`, `Building.o`). Das anschließende Bauen sollte nun funktionieren. Der Trick ist, dass `g++` beim Kompilieren für jede Sourcedatei ein Makefile generiert, das dessen eingebundene Header als Abhängigkeiten enthält (`main.d`, `Building.d`).

Wenn du jetzt Änderungen an der `toString`-Methode durchführst, werden diese anhand des Zeitstamps von `Building.h` erkannt.

Aufgabe 20.9 Target clean

Bisher mussten wir hin und wieder die kompilierten Dateien manuell löschen, wenn wir unser Projekt neu bauen wollten. Diese Aufgabe lässt sich mittels `make` ebenfalls automatisieren. Legt dazu ein neues Target `clean` ohne Abhängigkeiten an und fügt einen entsprechenden Befehl zum Löschen ein:

```
clean:  
    rm -rf main.o Building.o main.d Building.d main.exe  
  
.PHONY: clean
```

Das Spezial-Target `.PHONY` dient dazu, `make` zu signalisieren, dass `clean` keine Datei ist, die gebaut werden soll. Würden wir dieses Target auslassen und eine Datei mit Namen `clean` erzeugen, würde `make` die Regel nie ausführen, weil die Datei ja existiert und keine Abhängigkeiten besitzt. Du solltest `all` ebenfalls als `.PHONY` deklarieren. Probiere es ruhig aus!

Um `clean` auszuführen klicke neben dem Build-Symbol auf den Auswahlpfeil und wähle **Project only - Clean** aus. In der Konsole siehst du die Ausgabe von `make clean`.

Es ist auch möglich, andere Targets als `all` oder `clean` auszuführen. Diese müssen in CodeLite allerdings erst eingerichtet werden: **Rechtsklick auf das Projekt → Settings... → Customize**. Klicke hier auf `New...` und gib einen Namen für den Target ein und den auszuführenden Befehl (z.B. `make main.o` um nur die Datei `main.cpp` zu kompilieren aber nicht zu linken).

Aufgabe 20.10 Generisches Compiler-Target

Dir ist sicherlich aufgefallen, dass wir zwei Targets haben, die mehr oder weniger identisch sind: `main.o` und `Building.o`.

`make` bietet für solche Situationen generische Regeln an, die mittels Wildcards beschrieben werden.

Ersetze die beiden spezifischen Targets durch folgendes generisches:

```
.o: %.cpp  
    g++ -MMD -MP -c $< -o $@
```

Die etwas kryptischen Ausdrücke `\$<` und `\$@` werden durch die aktuelle Abhängigkeit und Target ersetzt.

Lösche alle automatisch generierten Dateien (`make clean`) und baue das Projekt neu.

Aufgabe 20.11 Variablen in make

Im Moment sieht unser Makefile in etwa so aus:

```
all: main.exe  
  
main.exe: main.o Building.o  
        g++ -o main.exe main.o Building.o
```

Aufgaben zu fortgeschrittenen Themen in C++

```
.o: %.cpp
    g++ -MMD -MP -c $< -o $@

-include Building.d main.d

clean:
    rm -rf main.o Building.o main.d Building.d main.exe

.PHONY: clean all
```

Dir ist sicherlich eine andere Form der Redundanz aufgefallen: Noch immer haben wir die Tatsache, dass es im Moment zwei Sourcedateien gibt, an unterschiedlichen Stellen im Makefile festgelegt. Wenn wir nun als nächstes die Floor-Klasse entwerfen, müssten wir diese hinzufügen

- als Abhängigkeit von `main.exe`,
- zum Linker-Aufruf in `main.exe`,
- in der `-include`-Direktive und
- im Target `clean`, und das gleich doppelt!

Das ist natürlich immer noch ziemlich fehleranfällig.

Wir würden also gerne nur an *einer* Stelle definieren, welche Sourcedateien Teil unseres Projektes sind.

Da die Sourcedateien nirgendwo im Makefile auftreten, fangen wir mit den Objekt-Dateien an. Lege am Anfang des Makefiles eine Variable mit dem Inhalt '`main.o Building.o`' und ersetze das Auftreten der beiden Objekt-Dateien mit dieser Variablen:

```
OBJECTS=main.o Building.o

all: main.exe

main.exe: $(OBJECTS)
    g++ -o main.exe $(OBJECTS)

# and so on...
```

Wiederhole die Prozedur für die Abhängigkeiten (`DEPEND=main.d Building.d`) und das ausführbare Programm (`BINARY=main.exe`). Lasse dein Programm zwischendurch immer wieder vollständig neu bauen, um sicherzustellen, dass nichts kaputt geht.

Am Ende sollte dein Makefile in etwa so aussehen:

```
BINARY=main.exe
OBJECTS=main.o Building.o
DEPEND=main.d Building.d

all: $(BINARY)

$(BINARY): $(OBJECTS)
    g++ -o $(BINARY) $(OBJECTS)

%.o: %.cpp
    g++ -MMD -MP -c $< -o $@
```

Aufgaben zu fortgeschrittenen Themen in C++

```
-include $(DEPEND)

clean:
    rm -rf $(OBJECTS) $(DEPEND) $(BINARY)

.PHONY: clean all
```

Wie wir die verbliebene Redundanz auflösen, sehen wir in der nächsten Teilaufgabe.

Aufgabe 20.12 Wildcard-Ausdrücke

Die beiden Variablen OBJECTS und DEPEND sind strukturell ähnlich – wieder etwas, das wir loswerden wollen. Außerdem wäre es doch viel schöner, an einer Stelle die Sourcedateien zu definieren, oder?

Lege dazu eine neue Variable SOURCES mit den beiden Sourcedateien an. Der folgende Snippet zeigt, wie man nun mittels Suffix-Ausdrücken die anderen beiden Variablen erzeugt:

```
BINARY = main.exe
SOURCES = main.cpp Building.cpp
OBJECTS = $(patsubst %.cpp, %.o, $(SOURCES))
DEPEND = $(patsubst %.cpp, %.d, $(SOURCES))
```

Es geht sogar noch allgemeiner: Du kannst per regulärem Ausdruck²³ definieren, dass *alle* Sourcedateien im aktuellen Ordner verwendet werden sollen, indem du SOURCES wie folgt definierst:

```
SOURCES=$(wildcard ./*.cpp)
```

Aufgabe 20.13 Zusammenfassung

Das Produkt unserer Bemühungen in dieser Aufgabe ist ein Makefile, das unabhängig davon ist, wie viele Sourcedateien du im aktuellen Verzeichnis hältst und wie sie genau heißen – wichtig ist nur die Endung .cpp.

Hier nochmal das vollständige Makefile:

```
BINARY = main.exe
SOURCES = main.cpp Building.cpp
OBJECTS = $(patsubst %.cpp, %.o, $(SOURCES))
DEPEND = $(patsubst %.cpp, %.d, $(SOURCES))

all: $(BINARY)

$(BINARY): $(OBJECTS)
    g++ -o $(BINARY) $(OBJECTS)

%.o: %.cpp
    g++ -MMD -MP -c $< -o $@

-include $(DEPEND)

clean:
```

²³ https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck

Aufgaben zu fortgeschrittenen Themen in C++

```
rm -rf $(OBJECTS) $(DEPEND) $(BINARY)
```

```
.PHONY: clean
```

Aufgabe 20.14 Nachwort

Dies hier sind nur äußerst wenige der Möglichkeiten, die `make` bietet.²⁴ In der Praxis existieren Build-Tools, die eine wesentlich besser zu verstehende Beschreibungssprache verwenden und daraus Makefiles generieren. Beispiele sind `cmake`²⁵ oder `qmake`²⁶ (Bestandteil von Qt).

²⁴ Für einen besseren Eindruck, sieh dir die Doku an: https://www.gnu.org/software/make/manual/html_node/index.html

²⁵ http://www.cmake.org/cmake/help/cmake_tutorial.html

²⁶ <http://qt-project.org/doc/qt-4.8/qmake-tutorial.html>

Aufgaben zu Embedded C

[C] Vorbemerkungen zum (Embedded-)C-Teil

Wir sind nun im zweiten Teil des Praktikums angekommen. In diesem Teil wollen wir zum einen die Unterschiede zwischen C und C++ kennenlernen (Aufgabe 21). Zum anderen wollen wir uns ansehen, wie man einen gängigen Microcontroller programmiert, also Digitaleingänge setzt, Digitalausgänge ausliest und den Analog-Digital-Wandler ansteuert.

Klausurrelevant (auf diesem Aufgabenblatt) ist ausschließlich Aufgabe 21.

Aufgabe 21 [C] Die Programmiersprache C im Vergleich zu C++

In den nächsten Tagen werden wir Programme für eine Embedded-Plattform in C entwickeln. Da C++ aus C entstand, sind viele Features von C++ nicht in C enthalten. Im Folgenden sollen die Hauptunterschiede verdeutlicht werden.

- Keine OO-Konzepte (Vererbung, ...)
- Strukturen (`struct`) statt Klassen (`class`)
- Keine Templates
- Keine Referenzen, nur Zeiger und Werte
- Kein `new` und `delete`, sondern `malloc()` und `free()` (`#include <stdlib.h>`)
- Je nach Sprachstandard müssen Variablen am Anfang der Funktion deklariert werden (Standard-Versionen bis einschließlich C99)
- Parameterlose Funktionen müssen `void` als Parametertyp haben, leere Klammern (bspw. `int foo();`) bedeuten, dass beliebige Argumente erlaubt sind.
- Keine Streams, stattdessen (`f`)`printf` zur Ausgabe auf Konsole und in Dateien (`#include <stdio.h>`)
- Kein `bool`-Datentyp, stattdessen wird 0 als `false` und alle anderen Zahlen als `true` gewertet
- Keine Default-Argumente
- Keine `std::string` Klasse, nur `char`-Arrays, die mit dem Nullbyte ('\0') abgeschlossen werden.
- Keine Namespaces

Da einige dieser Punkte sehr entscheidend sind, werden wir auf diese im Detail eingehen. Wichtig ist hierbei, dass alle im Folgenden vorgestellten Konzepte auch in C++ zur Verfügung stehen. Die Header der C-Bibliothek sind alle auch in C++ verfügbar. Möchte man bspw. `malloc` nutzen, kann man dies in C++ über `#include <stdlib.h>` oder über `#include <cstdlib>` (Allgemeines Muster: vorangestelltes 'c' und fehlendes '.h'). Im zweiten Fall sind alle Funktionen im Namensraum `std` eingebettet, man muss als `std::malloc` nutzen.

Aufgabe 21.1 Kein OO-Konzept

In C gibt es keine Klassen, weshalb die Programmierung in C eher Pascal statt C++ ähnelt. Stattdessen gibt es Strukturen (`struct`), die mehrere Variablen zu einem Datentyp zusammenfassen, was vergleichbar mit Records in Pascal oder – allgemein – mit Klassen ohne Methoden und ohne Vererbung ist.

Die Syntax dafür lautet

```
struct MyStruct {  
    <Type1> <Name11>, <Name12>, ...;  
    <Type2> <Name21>, <Name22>, ...;  
};
```

Aufgaben zu Embedded C

Zum Beispiel

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

Die Sichtbarkeit aller Attribute ist automatisch **public**. Um den definierten **struct** als Datentyp zu verwenden, muss man zusätzlich zum Namen das Schlüsselwort **struct** angeben:

```
void foo(struct Point *p) {  
    ...  
  
int main(void) {  
    struct Point point;  
    foo(&point);  
}
```

Um den zusätzlichen Schreibaufwand zu vermeiden, wird in der Praxis oft ein **typedef** auf den **struct** definiert:

```
typedef struct Point Point_t;  
Point_t point;
```

Man kann die Deklaration eines **struct** auch direkt in den **typedef** einbauen:

```
typedef struct {  
    int x;  
    int y;  
} Point;
```

Aufgabe 21.2 Kein **new** und **delete**

Anstelle von **new** und **delete** werden die Funktionen **malloc** und **free** verwendet, um Speicher auf dem Heap zu reservieren. Diese sind im Header **stdlib.h** deklariert.

```
#include <stdlib.h>  
Point *points = malloc(10 * sizeof(Point)); // reserve memory for 10 points  
// ...  
free(points);
```

Aufgabe 21.3 Ausgabe auf Konsole per **printf**

Um Daten auf der Konsole auszugeben, kann die Funktion **printf** verwendet werden. **printf** nimmt einen Format-String sowie eine beliebige Anzahl weiterer Argumente entgegen. Der Format-String legt fest, wie die nachfolgenden Argumente ausgegeben werden. Mittels **\n** kann man einen Zeilenvorschub erzeugen. Um **printf** zu nutzen, muss der Header **stdio.h** eingebunden werden.²⁷ Der folgende Codeausschnitt zeigt, wie man Zahlen und Zeichen mit **printf** ausgeben kann. Wenn die Variablen **i** und **c** nicht definiert werden, ist die Aufgabe undefiniert. Zu beachten ist auch, dass **printf** nicht automatisch einen Zeilenumbruch einfügt.

²⁷ Weitere mögliche Parameter etc. der Funktion **printf** findest du unter <http://www.cplusplus.com/reference/cstdio/printf/>.

Aufgaben zu Embedded C

```
#include <stdio.h>
printf("Hello Welt\n"); // Print 'Hello World' and add a line break

int i;
printf("i = %d\n", i); // Print integer
printf("i = %3d\n", i); // Print integer with a minimum width of 3

char c;
printf("c = %c, i = %d\n", c, i); // Print character and integer
```

Im Folgenden werden wir untersuchen, welche Folgen es hat, wenn man das „per Konvention“ erwartete Null-Byte ('\\0') entfernt. In diesem Fall wird der Speicher Byte-weise solange ausgegeben, bis ein Null-Byte angetroffen wird.

- Beginne mit einer leeren main-Funktion.
- Lege einen Puffer der Größe 6 an:

```
char *buffer = malloc(6 * sizeof(char));
```

- Kopiere mittels strcpy (aus dem Header string.h) den String "Hello" in den Puffer:

```
strcpy(buffer, "Hello");
```

- Nun gib den Inhalt des Puffers mittels printf aus:

```
printf("%s\n", buffer);
```

Wenn du das Programm jetzt kopierst und ausführst, sollte nur der String Hello erscheinen.

- Jetzt beginnt der spannende Teil: Überschreibe das Null-Byte mit einem beliebigen Zeichen (bspw. '_').

```
buffer[5] = '_';
```

Wenn du jetzt den Inhalt des Puffers ausgibst, kann alles passieren („Undefined Behavior“). Die Funktion printf wird solange den Speicher auslesen, bis sie ein Null-Byte trifft. Dieses Experiment zeigt, dass es sehr wichtig ist, bei der Manipulation von C-Strings gut aufzupassen – zahlreiche Sicherheitslücken basieren auf dieser Schwäche!

- Ändere zum Abschluss die Art, wie der Puffer allokiert wird wie folgt:

```
char *myString = "Hello";
```

Jetzt liegt der Puffer nicht mehr auf dem Heap (wie bei malloc) sondern im data-Segment. Wenn du den Code nun kompilierst und ausführst, solltest du einen Speicherzugriffsfehler („Segmentation Fault“) erhalten, da Schreibzugriffe in diesem Speicherbereich verboten sind.

Es ist auch möglich, einen C-String *einzukürzen*, indem man das Null-Byte verschiebt innerhalb des Puffers nach vorne verschiebt.

- Verwende den Code aus dem vorherigen Abschnitt und diejenige Zeile an, in der der Puffer manipuliert wird. Statt '_' an Index 5 zu platzieren, platziere jetzt das Null-Byte ('\\0') an Index 2.
- Bei der Ausführung sollte jetzt nur noch He ausgegeben werden.

Aufgabe 21.4 Strings zusammenbauen

Die Funktion sprintf dient dazu, formulierte Strings zusammenzusetzen und ist syntaktisch eng mit der Funktion printf verwandt.²⁸

²⁸ Weitere mögliche Parameter etc. der Funktion sprintf findest du unter <http://www.cplusplus.com/reference/cstdio/sprintf/>.

Aufgaben zu Embedded C

- Beginne mit einer leeren main-Funktion.
- Lege erneut einen Puffer auf dem Heap an:

```
char *buffer = malloc(100 * sizeof(char));
```

- Gib den Puffer direkt mittels printf aus:

```
printf(buffer);
```

Dies zeigt, dass es sehr gefährlich ist, mit uninitialisierten C-Strings zu hantieren. Sicherer wäre es, direkt nach der Allokation ein Null-Byte an Index 0 des Puffers zu platzieren:

```
buffer[0] = '\0';
```

Wenn du anschließend erneut printf aufrufst, sollte die Ausgabe leer bleiben.

- Verwende den folgenden Aufruf, um einen Gruß mittels sprintf zusammenzusetzen. Lege dazu im Vorfeld eine Variable name an.

```
sprintf(buffer, "Hello, %s!\n", name);
```

- Den so gefüllten Puffer gibst du wie gewohnt über printf aus:

```
printf(buffer);
```

- Die Funktion sprintf kann natürlich auch zur Ausgabe (bspw.) von Zahlen und Zeichen genutzt werden:

```
sprintf(buffer, "c = %c, i = %3d", c, i);
```

Beachte auch hier, dass der Puffer auf dem Heap (statt im data-Segment) liegen muss, damit die Funktion sprintf hineinschreiben kann. Andernfalls erhältst du einen Segmentation Fault.

Aufgabe 21.5 Zahlen formatiert ausgeben

Schreibe ein C-Programm, welches alle geraden Zahlen von 0 bis 200 formatiert ausgibt. Die Formatierung soll entsprechend dem Beispiel erfolgen:

```
2   4   6   8   10  
12  14  16  ...
```

Mache die Spaltenzahl und Spaltenbreite mithilfe von Variablen konfigurierbar, sodass es auch leicht möglich ist, 15 Spalten und/oder Zahlen bis 10 000 auszugeben.

Aufgabe 21.6 Bit-Operatoren in C (und C++)

In dieser Aufgabe machst du dich mit den Bit-Operatoren (&, |, ~, >>, <<) in C vertraut. Alle Operatoren können exakt gleich in C++ verwendet werden. Bit-Operatoren sind für ganzzahlige Typen definiert (bspw. (`unsigned`) `int`, (`unsigned`) `char`). Ein Bit-Operator bezieht sich dabei auf jedes einzelne Bit. Im Gegensatz dazu beziehen sich logische Operatoren (||, &&) immer auf den gesamten Wert.

Zum Experimentieren stellen wir dir eine Vorlage zur Verfügung: https://github.com/Echtzeitsysteme/tud-cpp/blob/master/exercises/templates/05_c/ex21_6_BitAndLogicOperations/ex21_6_BitAndLogicOperations.c. Die Funktion fmt wandelt ein Byte in einen String um, der das Bit-Pattern darstellt. Zum Beispiel ist die Ausgabe von fmt(23) der String "`0b00010111`" (19 = 1 + 2 + 4 + 16).

Aufgaben zu Embedded C

- Füll zunächst die mit `// TODO implement me` gekennzeichneten Zeilen aus, sodass die Ausgabe korrekt ist. Beispiele sind für NEG und NOT gegeben.

Stelle sicher, dass deine Ergebnisse für $a = 23$, $b = 3$ mit den erwarteten Ergebnisse in folgender Tabelle übereinstimmen

Ausdruck	Ergebnis
$a \text{ AND } b$	3
$a \text{ OR } b$	23
$a \text{ XOR } b$	20
$a \text{ RIGHT } s$	5
$a \text{ LEFT } s$	92

Die folgende Tabelle enthält die erwarteten Ergebnisse der logischen Operatoren für alle Wertkombinationen von a und b:

Ausdruck	Ergebnis			
	$a=1, b=1$	$a=1, b=0$	$a=0, b=1$	$a=0, b=0$
$a \text{ LAND } s$	1	0	0	0
$a \text{ LOR } s$	1	1	1	0
$a \text{ XOR } s$	0	1	1	0
$a \text{ IMP } s$	1	0	1	1
$a \text{ BIIMP } s$	1	0	0	1

- Im Allgemeinen entspricht eine Verschiebung nach links/rechts einer Multiplikation mit/Division durch 2. Dazu werden jeweils von rechts-links 0-Bits eingeschoben. Unter welchen Umständen gilt dies nicht mehr? Was passiert, wenn du den Datentyp von a auf `unsigned char` setzt? Beobachte auch, ob das angezeigte Bitmuster mit dem ausgegebenen Wert übereinstimmt.
- Ändere den Wert der Variablen a in -1. Was passiert bei einem Rechts-Shift? Was bei einem Links-Shift? Wie unterscheidet sich das Verhalten im Vergleich zu positivem a?
- Ändere den Wert von s zu -1. Wie sieht das Ergebnis des Rechts-/Links-Shifts aus? Ein Shift um einen negativen Wert ist „Undefined Behavior“. Welche Ergebnisse sind demnach erlaubt?

Aufgabe 21.7 C++-Aufgaben nachprogrammieren (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Dies ist eine freie Aufgabe, in der du versuchst, Programme der vergangenen Tage in reinem C auszudrücken. Der Schwierigkeitsgrad ist dabei sehr unterschiedlich!

Niedrigerer Schwierigkeitsgrad

- Sternen- oder Buchstabenmuster** ausgeben (Aufgabe 2): Diese Aufgabe ist sehr ähnlich zu Aufgabe 21.5.
- Werte analysieren** und mit **Arrays** arbeiten (Aufgabe 5 und Aufgabe 6): Abgesehen von der fehlenden Unterstützung für Referenzen sollten die Ergebnisse sich nicht unterscheiden.
- Funktionszeiger** (Aufgabe 17 und Aufgabe 18): Funktionszeiger in C arbeiten genauso wie Funktionszeiger in C++. Du kannst dies testen, indem du die Programmlogik der vorigen Aufgabe in eine separate Funktion auslagernst, die neben der Spaltenbreite und Obergrenze der anzulegenden Zahlen zusätzlich einen Funktionszeiger-Parameter hat, der festlegt, was mit der jeweiligen Zahl vor der Ausgabe geschehen soll (bspw. verdoppeln, quadrieren).

Aufgaben zu Embedded C

Höherer Schwierigkeitsgrad

- **Vector** (Aufgabe 4): C bietet keine Objektorientierung, aber du kannst eine **struct** zur Datenhaltung anlegen und die Methoden der Klasse als Funktionen realisieren, deren Namen bspw. immer mit dem Präfix **Vector_** beginnen und die als zusätzlichen Parameter einen Pointer auf eine **Vector-struct** erhalten.
- **Verkettete Listen** (Aufgabe 7): Die reinen Datenstrukturen für die Liste (**List**) und für Listenelemente (**ListItem**) lassen sich als **struct** abbilden. Methoden kannst du wieder auf Funktionen mit Namenskonvention und zusätzlichem Pointer-Parameter abbilden.
- **Generischer Vektor und Liste** (Aufgabe 14 und Aufgabe 15): Auch wenn es in C keinen eingebauten Mechanismus wie die C++-Templates gibt, kannst du die Vector und List-Klasse generisch machen, indem du die Einträge des Vektors bzw. den content der ListItems als **void*** deklarierst.
- **Eigene Array-Klasse** (Aufgabe 19): Ebenso wie bei generischem Vektor und Liste kannst du natürlich auch deine eigene Array-Klasse schreiben.

Aufgabe 22 [C] Testprogramm auf den Microcontroller laden (optional)

Alle nun folgenden, mit [C] markierten Aufgaben sind nicht klausurrelevant. Sie dienen dazu, dich in die Welt der Embedded-C-Programmierung einzuführen. Embedded C ist hierbei genau die gleiche Programmiersprache wie C. Jedoch gibt es einige technische Besonderheiten zu berücksichtigen.

Aufgabe 22.1 Überblick

Für die Arbeit mit dem Evaluationsboard nutzen wir die Entwicklungsumgebung *WinIDEA Open*²⁹. Im Vergleich zu CoDeLite ist diese Umgebung speziell auf die Entwicklung von Embedded C zugeschnitten. Der Bauprozess für Embedded-C-Programme sieht teilweise anders aus als bei C++-Programmen:

- a) Das Ergebnis der **Link-Phase** ist kein auf dem PC ausführbares Programm, sondern ein sogenanntes *Image*. Dieses Image wird in den statischen Speicher des Microcontrollers geladen.
- b) Nach der Link-Phase folgt die **Flash-Phase** (in WinIDEA auch „Download“ genannt). Während dieser Phase wird das Image auf den Microcontroller übertragen.
- c) Anschließend beginnt die **Ausführung** direkt oder man muss den *Reset*-Knopf des Boards drücken, um den Programmzähler zurückzusetzen.
- d) Standardmäßig geht WinIDEA hierbei direkt in den **Debug-Modus**. Das bedeutet, dass die Ausführung des Programms am Beginn der `main`-Funktion angehalten wird.

Die weiteren Besonderheiten vom Embedded C sehen wir uns anhand eines einfachen Testprogramms an.

Aufgabe 22.2 Testprogramm

Für diese und alle weiteren Aufgaben stellen wir dir ein Codetemplate zur Verfügung, das von dir ergänzt wird. Wir beginnen mit einem kleinen fertigen Programm, das die RGB-LED des Evaluationsboards periodisch blinken lässt. Dies ist das „Hello World“-Programm der Embedded-C-Welt.

- a) Kopiere zunächst den **vorbereiteten WinIDEA-Workspace** (`/exercises/templates/05_WinIdeaWorkspace-Template`) in ein Verzeichnis **außerhalb** deines Benutzerverzeichnisses (bspw. nach `C:\tmp`). ³⁰ Falls du deinen eigenen PC verwendest, ist es sehr wichtig, dass der WinIDEA-Workspace und die verwendeten Bibliotheken (bspw. `C:\PortableApps\Cypress\PDL`) auf dem gleichen Laufwerk liegen.

²⁹ <http://www.isystem.com/download/winideopen>

³⁰ Leider ist es aus technischen Gründen nicht möglich, mit WinIDEA im Benutzerverzeichnis zu arbeiten, da dieses auf ein Netzlaufwerk abgebildet wird.

Aufgaben zu Embedded C

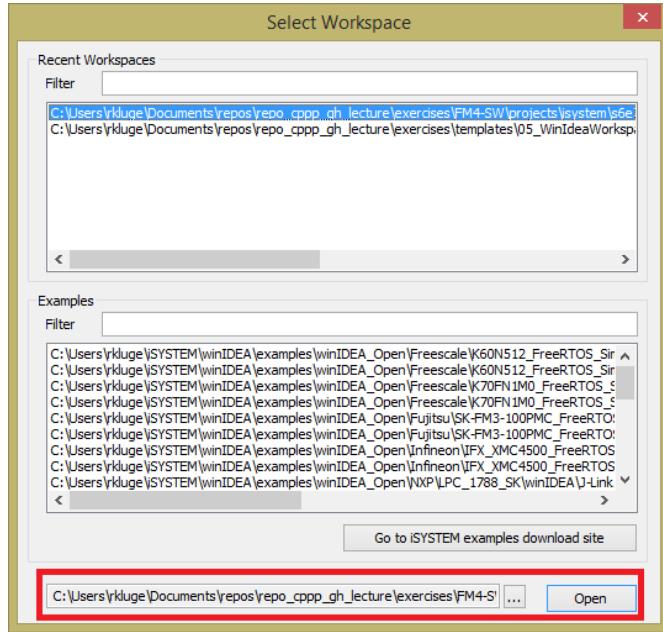


Abbildung 3: Workspace-Auswahl in WinIDEA

- b) **Öffne WinIDEA:** Das Programm liegt unter C:\PortableApps\iSYSTEM\winIDEAOpen9\winIDEA.exe. Bei Bedarf kannst du dir eine Desktop-Verknüpfung erstellen.
- c) **Wähle** in dem erscheinenden Dialogfenster den soeben **kopierten WinIDEA-Workspace** aus wie in Abbildung 3 gezeigt (roter Rahmen).
- d) Nun öffnet sich der eigentliche Workspace. Du findest links einen baumartig aufgebauten Datei-Browser, der die Datei-Gruppen lib, src und Dependencies enthält.
- Die Gruppe lib enthält manuell konfigurierte Abhängigkeiten deines Projekts, die du normalerweise nicht anpassen musst.
 - Die Gruppe src enthält den Quelltext, mit dem du arbeiten wirst.
 - Die Gruppe Dependencies enthält automatisch von WinIDEA aufgelöste Abhängigkeiten und muss nicht angepasst werden.

Öffne nun die Datei **main.c** in der Gruppe **src**.

- e) Hier findest du die **main-Funktion**, deren Inhalt du im Folgenden immer wieder anpassen wirst. Im Moment delegiert sie an die Funktion BlinkMain. Diese ist in der Datei blink.h deklariert und in blink.c definiert. Öffne nun die Datei **blink.c**.
- f) Du findest den folgenden Quelltext (Listing 1) vor:

```
1 #include "blink.h"
2
3 #include <stdint.h>
4 #include "delay.h"
5 #include "s6e2ccxj.h"
6
7 #include "pins.h"
8
9 int BlinkMain() {
```

Aufgaben zu Embedded C

```
10 LED_BLUE_DDR |= (1 << LED_BLUE_PIN); // Configure blue LED pin as output.
11 LED_BLUE_DOR |= (1 << LED_BLUE_PIN); // Turn LED off.
12
13 const uint32_t sleepTime = 1000000;
14
15 // Main loop
16 while (1) {
17     // Clear bit -> Switch LED on
18     LED_BLUE_DOR &= ~(1 << LED_BLUE_PIN);
19     microDelay(sleepTime);
20
21     // Set bit -> Switch LED off
22     LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
23     microDelay(sleepTime);
24 }
25 }
```

Listing 1: blink.c

- Zu Beginn wird der Pin, an den die blaue LED angeschlossen ist, als Ausgang konfiguriert (Zeile 10).
 - Dann wird der Pin auf 1 gesetzt, was die LED ausschaltet (Zeile 11).
 - In einer Endlosschleife wird die LED dann immer wieder ein- und ausgeschaltet. Zwischen den Schaltvorgängen wird eine Pause von einer Sekunde eingelegt.
- g) Um das Projekt jetzt zu **compilieren** und zu **linken**, wähle im Menü *Project* den Befehl *Make* (oder drücke F7). Dieser Befehl wird – seinem Namen entsprechend – nur diejenigen Teile neu kompilieren, die sich seit dem letzten Aufruf verändert haben. Mit dem Befehl *Project → Rebuild* wird das Projekt von Grund auf neu gebaut. Der Vorgang sollte im *Output*-Fenster³¹ mit **0 Error(s)** **0 Warning(s)** enden.
- h) Um das erzeugte Image auf den Microcontroller zu flashen, verbinde den **Micro-USB-Anschluss CN2** des Boards mit dem **Data-USB-Port** des PCs. Die grüne LED in der Nähe des 2x5-Pin-Multicon-Blocks (CN12) sollte nun dauerhaft leuchten. Wähle anschließend in WinIDEA **Debug → Download** (oder Ctrl + F3).
- i) In WinIDEA wird nun ein **gelber Pfeil** neben der Signatur der **main-Funktion** erscheinen. Dies deutet an, dass du jetzt auf dem Microcontroller debuggen kannst.³² Setze die Ausführung einfach fort, indem du **Debug → Run Control → Run** wählst (oder F5). Die blaue LED sollte nun blinken.
- j) Falls dein Programm einmal „unsauber“ startet oder du es neu starten möchtest, kannst du den Programmzähler mithilfe des **Reset Buttons** zurücksetzen. Er befindet sich links oberhalb der MCU und ist auch mit „Reset“ beschriftet. Ganz in der Nähe befindet sich auch der **User Button**, den wir im Folgenden noch kennenlernen werden.

Herzlichen Glückwunsch – du bist nun bereit, selber Hand an die Aufgaben zu legen!

Aufgabe 22.3 LED bunt blinken lassen

Als erste eigenständige Aufgabe erweiterst du die Funktion `BlinkMain` so, dass abwechselnd alle drei möglichen LEDs angesteuert werden. Führe dazu die folgenden Schritte aus:

- Öffne die Dateien Dateien `blinkrainbow.h` und `blinkrainbow.c` (in der Gruppe `src`).

³¹ Kann über *View → Output* oder Alt+2 geöffnet werden.

³² Dieses Verhalten lässt sich auch ausschalten: <https://github.com/Echtzeitsysteme/tud-cppp/wiki/WorkingWithWinIDEAOpen#how-can-i-prevent-winidea-from-stopping-at-main-after-downloading>

Aufgaben zu Embedded C

- b) Nutze die Dateien `blink.h` und `blink.c` als Ausgangspunkt, um diese Aufgabe zu lösen.
- c) Erweitere nun den Code so, dass du auch auf die Ports der roten und grünen LED zugreifst und die LED abwechselnd rot, grün und blau leuchtet.

Listing 2 zeigt am Beispiel der blauen LED und des linken Joystick-Buttons wie man auf IO-Pins zugreifen kann. Die Bit-Operatoren wurden bereits in Aufgabe 21.6 behandelt. Die Namen der verschiedenen Register für die LEDs und Buttons sind in der Datei `pins.h` definiert. Ausdrücke wie `LED_BLUE_DDR |= (1 << LED_BLUE_PIN)` oder `BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN)` werden verwendet, um ein einzelnes Bit in einem Register zu setzen oder zu löschen.

```
1 #include "s6e2ccxj.h"
2 #include "pins.h"
3
4 int io_example(void)
5 {
6     /*
7     Configure the pin of the blue LED as output by setting the corresponding
8     bit in the Data Direction Register (DDR).
9     */
10    LED_BLUE_DDR |= (1 << LED_BLUE_PIN);
11    /*
12    Set the pin to 1 by setting the corresponding bit in the Data Output
13    Register (DOR). This turns the LED off.
14    */
15    LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
16
17    /*
18    Configure the pin of the blue LED as output by clearing the corresponding
19    bit in the Data Direction Register (DDR).
20    */
21    BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN);
22    /*
23    Enable the pull-up resistor in the pin by setting the corresponding bit
24    in the Pullup Configuration Register (PCR).
25    */
26    BUTTON_LEFT_PCR |= (1 << BUTTON_LEFT_PIN);
27
28    /*
29    Check the pin status by combining the Data Input Register (DIR) with the
30    corresponding bitmask.
31    The expression is inverted because the pin is 0 when the button is pressed.
32    */
33    if(!(BUTTON_LEFT_DIR & (1 << BUTTON_LEFT_PIN))) {
34        /*
35        Toggle the LED when the button is pressed. Toggling is done by XORing
36        the current pin state with 1.
37        */
38        LED_BLUE_DOR ^= (1 << LED_BLUE_PIN);
39    }
40 }
```

Listing 2: Beispiele zur Ansteuerung von Ein-/Ausgabepins

Aufgaben zu Embedded C

Hinweise

- Für diese und die folgenden Kennenlernaufgaben stellen wir dir Musterlösungen direkt im WinIDEA-Workspace bereit. Du findest sie in der Gruppe **solution**. Beachte, dass die (nicht-statischen) Funktionen der Musterlösung und deren entsprechende Implementierungsdateien stets auf `_s` enden, um Namenskonflikte mit den Vorlagefunktionen zu vermeiden.

Aufgabe 23 [C] Taster abfragen (optional)

In dieser Aufgabe erweiterst du die vorherige Aufgabe um eine Benutzerinteraktion über den Taster des linken Joysticks (**Joystick 1**). Ziel dieser Aufgabe ist es, mithilfe des Tasters die RGB-LED in zwei verschiedenen Szenarien zu kontrollieren. Im ersten Szenario soll der Taster als Lichtschalter arbeiten: Wird der Taster einmal betätigt, schaltet sich die blaue LED ein; wird der Taster erneut betätigt, schaltet sie sich wieder ab. Im zweiten Szenario soll die blaue LED solange leuchten, wie der Taster gedrückt gehalten wird.

- a) In dieser Aufgabe wirst du mit den Dateien **button.c** und **button.h** arbeiten.
- b) Zunächst werden wir die nötigen Variablen in **button.c** deklarieren: Um den aktuellen Zustand der LED zu speichern wird eine vorzeichenlose 8-Bit-Integer-Variable **ledStatus** angelegt.
- c) Implementiere zunächst die Funktion **initLED()**. Diese soll ledStatus und den Pin der blauen LED initialisieren.
 - Der LED-Status soll zu Beginn 0 (= „aus“) sein.
 - Der Pin der blauen LED muss als Ausgang konfiguriert werden.
 - Das Daten-Register der blauen LED soll entsprechend initialisiert sein, dass die LED ausgeschaltet ist.
- d) Implementiere nun die Funktion **toggleBlueLED()**. Diese soll den Status von ledStatus umkehren: War der Wert zuvor 1, soll er danach 0 sein und umgekehrt. Der aktuelle Status der LED soll mit **setBlueLED(uint8_t status)** gesetzt werden.
- e) Implementiere nun die Funktion **isButtonPressed**, die zurück gibt, ob der Taster gerade gedrückt ist. Beachte, dass der Taster durch den Pull-Up-Widerstand genau dann gedrückt ist, wenn am Pin ein niedriger Pegel (0) anliegt. Ein Beispiel für die Abfrage des Tasters findest du in Listing 2.
- f) Implementiere nun mithilfe der zuvor erstellten Hilfsfunktionen die Hauptfunktionen **ButtonToggleBlueLED()** und **ButtonHoldBlueLEDOn()**. Die erste soll dem Button die Funktion eines Lichtschalters geben und die zweite soll die LED zum Leuchten bringen, solange der Taster gedrückt gehalten wird.

Aufgabe 24 [C] Display ansteuern (optional)

In diesem Abschnitt lernst du die Ansteuerung des Displays kennen. Das Display hat eine Auflösung von 480 * 320 Pixeln. Zunächst wirst du den Bildschirm in verschiedenen Farben ausfüllen und die dafür benötigte Farbcodierung kennenlernen. Anschließend wirst du Funktionen implementieren, um auf dem Bildschirm regelmäßige Muster und Texte auszugeben. Alle in dieser Aufgabe verwendeten Funktionen **müssen in der Datei display.c implementiert werden**. Tabelle 2 dokumentiert das erwartete Verhalten der zu implementierenden Funktionen.

Aufgabe 24.1 Bildschirm umfärben

Das Display verwendet eine RGB565-Codierung für Farben (auch als „High Color“ bekannt³³). Bei der RGB565-Codierung werden 5 Bits für Rot, 6 Bits für Grün und 5 Bits für Blau verwendet (siehe Abbildung 4).

Um auch eigene Farben nutzen zu können, implementierst du zunächst die Funktion **color565(uint8_t r, uint8_t g, uint8_t b)**, welche RGB888-Farben in RGB565-Farben umwandelt. Der Umwandlungsalgorithmus arbeitet für drei gegebene Bytes (uint8_t oder unsigend char) r, g, b wie folgt:

³³ https://en.wikipedia.org/wiki/High_color

Aufgaben zu Embedded C

Tabelle 2: Wichtige Funktionen und Variablen für das Display

Funktionen/Variablen	Beschreibung
<code>void drawRect(int16_t x, int16_t y, int16_t w, int16_t h, uint16_t color)</code>	Zeichnet die Umrandung eines Rechtecks in der Farbe color mit der Breite w und der Höhe h an die Stelle x,y.
<code>void fillRect(int x1, int y1, int w, int h, uint16_t fillcolor)</code>	Zeichnet ein ausgefülltes Rechteck in der Farbe fillcolor mit der Breite w und der Höhe h an die Stelle x,y.
<code>void drawPixel(int16_t x, int16_t y, uint16_t color)</code>	Zeichnet ein Pixel an x,y mit der Farbe color.
<code>void fillScreen(uint16_t color)</code>	Füllt den gesamten Bildschirm mit der Farbe color.
<code>uint16_t color565(uint8_t r, uint8_t g, uint8_t b)</code>	Umwandlung von RGB in HEX 565.

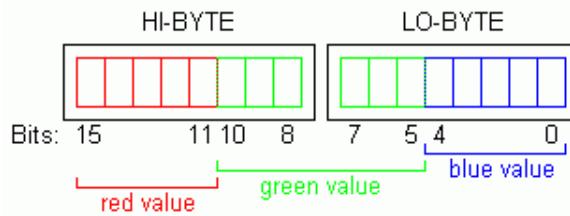


Abbildung 4: RGB565 Codierung

- a) Extrahiere die höchstwertigen 5 Bits von r, die höchstwertigen 6 Bits von g und die höchstwertigen 5 Bits von b. Nutze den Bit-Und-Operator mit einer passenden Bitmaske, um nur die entsprechenden Bits beizubehalten und alle anderen Bits auf 0 zu setzen. Verwende anschließend den Shift-Operator, um die extrahierten Bits ans untere Ende des Bytes zu schieben. Es folgt eine Skizze für den Rot-Kanal.

```
uint8_t r          = 0xA7;      // --> 0b10100111
uint8_t hiR        = r ____; // --> 0xA0; 0b10100000
uint8_t hiRShifted = r ____; // --> 0x14; 0b00010100
```

- b) Füge die extrahierten Bits mittels des Bit-Oder-Operators zusammen, um den RGB565-Wert zu erhalten: Beachte dabei, dass du mithilfe des Links-Shift-Operators die einzelnen Teile korrekt ausrichtest. Beispielsweise muss hiRShifted vor der Veroderung um 11 Bits nach links verschoben werden. Beachte, dass das Ergebnis 16 Bit lang sein muss und wir deshalb den Typ uint16_t verwenden.

```
uint16_t result = (hiRShifted ____) | (hiGShifted ____ ) | (hiBShifted ____);
```

Tabelle 3: RGB565-Farbwerthe

Farbe	RGB888	RGB565
Cyan	0x00EAFF	0x075F
Rosa	0xFC00FF	0xF81F
Orange	0xFFFF400	0xFDA0

Aufgaben zu Embedded C

Zum Testen deiner Implementation kannst du die Vergleichswerte in Tabelle 3 nutzen. Gehe dazu wie folgt vor:

- Lege dir eine leere main-Funktion an und binde den Header display.h ein.

- Füge die folgende Zeile ein, um deinen Code für die Farbe Cyan zu testen:

```
color565(0x00, 0xEA, 0xFF);
```

- Nun erstelle das Projekt wie gewohnt, aber, bevor du das Programm auf den Mikrocontroller lädst, setze einen *Breakpoint* in color565. Öffne dazu die Datei display.c. Rechtsklicke in die Zeile, in der die Variable result zurückgegeben wird. Wähle *Set Breakpoint* (oder drücke F9). Links neben der Zeile erscheint ein rotes Rechteck.

- Lade nun den Code auf den Microcontroller. Die Ausführung sollte an dem eingestellten Breakpoint anhalten (gelber Pfeil).

- Bewege nun den Mauszeiger über die Variable result. Es erscheint ein kleiner Tooltip mit dem Inhalt result =1887. Dies ist das richtige Ergebnis, aber leider in einer nicht-hexadezimalen Darstellung.

- Um dies zu ändern, öffne die *Watch View* mittels *View → Watch* (oder Alt + 3).

- Betätige die dritte Schaltfläche von links („Toggle Hex Mode“) am oberen Rand des *Watch*-Fensters.

- Wenn du mit dem Mauszeiger nun erneut auf result zeigst, erfährst du, dass result=0x075F.

- Um das Programm weiterlaufen zu lassen, wähle *Debug → Run Control → Run* (oder F5).

Die Datei display.h enthält einige vordefinierte Farben, die du ebenfalls bei den folgenden Aufgaben nutzen kannst:
BLACK (0x0000), BLUE (0x001F), RED (0xF800), GREEN (0x07E0), YELLOW (0xFFE0), WHITE (0xFFFF).

Im letzten Teil dieser Aufgabe färbst du den Bildschirm mit der Farbe deiner Wahl.

- Beginne wieder mit einer leeren main-Funktion.

- Binde die Header init.h (aus lib) ein und füge den folgenden Initialisierungscode für das Board in main ein:

```
initBoard();
```

- Um das Display in Cyan einzufärben, füge folgende Zeile an:

```
fillScreen(color565(0x00, 0xEA, 0xFF));
```

Aufgabe 24.2 Regelmäßiges Muster ausgeben

In diesem Abschnitt implementierst du die Funktion void printPattern(backgroundColor, foregroundColor). Diese Funktion ordnet auf dem Display Quadrate (4 x 4 Pixel) als Schachbrettmuster an.

Hinweise

- Nutze die Funktion fillScreen, um zunächst den Bildschirm in der Farbe backgroundColor zu füllen. Nutze anschließend die Funktion fillRect, um die einzelnen Quadrate in der Farbe foregroundColor zu erzeugen.
- Eine einfache Möglichkeit ist, zwei geschachtelte **for**-Schleifen zu verwenden, die den Schleifenzähler jeweils um die doppelte Blockgröße weiterschalten.

Aufgabe 24.3 Text ausgeben

In diesem Aufgabenteil wirst du einen Cursor implementieren, der es erlaubt, auf einfache Art und Weise Zeichen auf dem Bildschirm auszugeben. Zum Anzeigen von Buchstaben auf dem Bildschirm stellen wir dir eine kleine Fontbibliothek im Extended-ASCII 5*7-Format zur Verfügung. Du findest diese in der Datei glcdfont.h (Gruppe lib). Die Bibliothek besteht

Aufgaben zu Embedded C

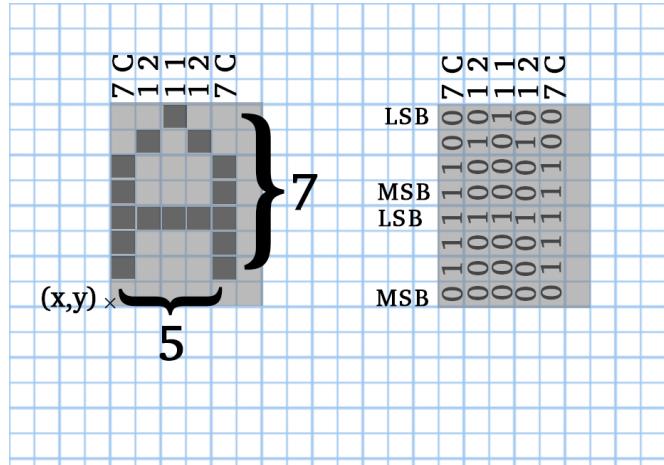


Abbildung 5: Schematische Darstellung des Zeichens 'A' in einer ASCII-5*7-Schriftart

aus einem Array mit 255 Buchstaben, die jeweils als 5 Bytes abgespeichert sind (siehe Abbildung 5). Das 5*7-Format eignet sich für den 480 * 320 großen Bildschirm, da die Darstellung dieser Schriftart mit einer Fläche von 35 Pixeln je Buchstaben immer noch gut lesbar ist. Wenn man von einem Pixel Abstand zwischen den Buchstaben ausgeht, passen auf das Display bis zu 3200 Zeichen.

Du wirst in dieser Aufgabe mehrere Funktionen in der Datei `display.c` implementieren. Der Koordinatenursprung (0,0) des Bildschirms ist links unten. In der Vorlagendatei `display.c` ist die Fontbibliothek `glcdfont.h` bereits eingebunden. Um globale Grundeinstellungen für die Schriftart festzulegen, wurden bereits die Variablen `cursorX`, `cursorY`, `textColor`, `textSize` und `textBackground` deklariert.

- a) Implementiere die Setter für den Cursor (`setCursor(int16_t x, int16_t y)`), die Textfarbe (`setTextColor(uint16_t c)`), die Textgröße (`setTextSize(uint8_t s)`) sowie die Hintergrundfarbe (`setBackgroundColor(int bg)`).

Zudem sollen in der Funktion `initCursor` folgende Grundeinstellungen vorgenommen werden: Der Cursor ist im Koordinatenursprung, die Textfarbe ist Weiß, die Hintergrundfarbe ist Schwarz und die Textgröße ist 2.

- b) Die Funktion `drawChar(x, y, c, color, bg, size)` soll einen Buchstaben `c` in ASCII-Schriftart auf dem Display an der Position `x, y` in der Farbe `color` mit der Hintergrundfarbe `bg` und in der Größe `size` abbilden.

- Zunächst muss überprüft werden, ob die angegebene Position gültig ist. Sofern die Werte die Grenzen des Bildschirms überschreiten, soll die Funktion `drawChar` beendet werden.
- Als nächster Schritt soll auf die richtige Stelle des Arrays `font` zugegriffen werden und die gesetzten Bits der Hexadezimalwerte als farbiges Pixel interpretiert werden. Abbildung 5 zeigt den Aufbau der Daten im Array `font` am Beispiel des Buchstabens 'A'. Ein beliebiger Buchstabe `c` ist in `font` an der Position `c * 5` gespeichert. Für diesen Buchstaben werden 5 Byte/40 Bits durchteriert. Für jedes gesetzte Bit wird an der entsprechenden Stelle ein Pixel (für `size == 1`) oder Rechteck (für `size > 1`) auf dem Display ausgegeben.
- Zusätzlich soll nach dem Buchstaben ein Leerraum gesetzt werden. Der Leerraum soll die Breite `size` haben.

- c) Um die Textausgabe auf dem Display für Strings zu ermöglichen, müssen weitere Funktionen implementiert werden, die einen automatischen Cursor verwenden. Dieser Cursor speichert die Position des letzten geschriebenen Buchstabens. Die Funktion `writeAuto(char c)` soll die Aufgabe übernehmen, einen Buchstaben `c` auf dem Display mithilfe von `drawChar` zu schreiben und die Cursorposition zu verändern. Hierbei sind Display-Grenzen und Zeilenumbrüche zu beachten.

- d) Implementiere die Funktion `writeText(const char *text)`, welche einen C-String als Parameter erhält und diesen mithilfe von `writeAuto` auf das Display schreibt.

Aufgaben zu Embedded C

- e) Implementiere abschließend die Funktion `writeTextln(const char *text)`, die sich ähnlich wie `writeText` verhält, am Ende der Textausgabe jedoch zusätzlich einen Zeilenumbruch einfügt.
- f) Implementiere zur Ausgabe von Zahlen die Funktionen `writeNumberOnDisplay(const uint8_t *value)` und `writeNumberOnDisplayRight(const uint8_t *value)`. Erstere soll die per Pointer übergebene Zahl linksbündig ausgeben, Zweitere soll die Zahl rechtsbündig ausgeben. Rechtsbündig bedeutet hier, dass die Ausgabe immer die gleiche Breite hat, egal ob der Wert 0 oder 65535 ist (also: 5 Ziffern). Verwende die Funktion `itoa` (aus `stdlib.h`³⁴), um den per Zeiger übergebenen Wert in ein `char`-Array zu schreiben und anschließend mittels `writeText` auf dem Display aus.

Hinweise

- Falls du Sonderzeichen darstellen möchtest, kannst du dich an folgender Tabelle orientieren: <http://www.theasciicode.com.ar/american-standard-code-information-interchange/ascii-codes-table.png>.

Um beispielsweise ein 'ß' auszugeben, definierst du es einfach als `char ss = '\xE1';`

Aufgabe 25 [C] Joysticks abfragen (optional)

In diesem Abschnitt nutzen wir die Funktionen der vorigen Aufgabe, um die analogen Werte der Joysticks auszugeben. Darauf aufbauend entwickelst du eine Aufgabe, die mithilfe des Joysticks die Farbe der RGB-LED verändert. Die zu implementierenden Funktionen befinden sich in der Datei `joystick.c`. Solltest du die vorherige Aufgabe nicht (vollständig) bearbeitet haben, kannst du auf die Musterlösung in der Datei `display_s.c` zurückgreifen. Um diese statt deiner eigenen Lösung zu nutzen, hängst du an jeden Funktionsnamen das Suffix `_s` an (bspw. `writeChar_s` statt `writeChar`).

Aufgabe 25.1 Analoge Werte auf dem Display anzeigen

Jeder Joystick besitzt zwei analoge Leitungen, welche die X- oder Y-Position des Steuerknüppels auslesen. Die analogen Werte entsprechen dabei der Spannung des jeweiligen Drehpotentiometers der Achse, welche zwischen 0 V und 5 V liegt. Die Spannungswerte der Joysticks werden durch den Analog-Digital-Wandler des Microcontrollers auf einen 8-Bit-Wert abgebildet (Wertebereich: 0 bis 255). Die Aufteilung der Wertebereiche sowie die Orientierung der X- und Y-Richtung sind in Abbildung 6 dargestellt. Die Aufteilung ist nicht gleichmäßig. Stattdessen ergibt sich in Neutralstellung der Wert

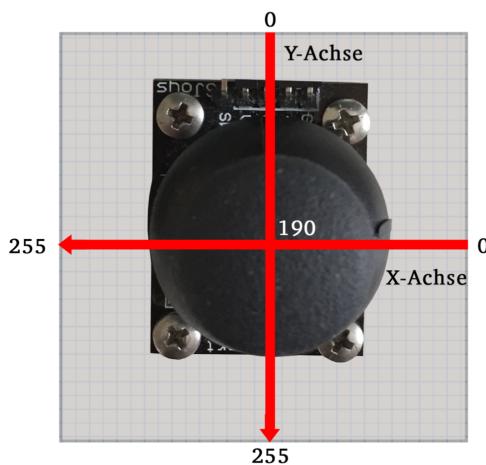


Abbildung 6: Wertebereich der Joysticks

(190, 190). Joystick 1 ist mit den analogen Anschlüssen AN16 (X) und AN19 (Y) und Joystick 2 ist mit AN13 (X) und AN23 (Y) verbunden.

³⁴ <http://wwwcplusplus.com/reference/cstdlib/itoa/>

Aufgaben zu Embedded C

- a) Implementiere die Funktion `printValues`, welche über Zeiger auf die analogen Leitungen AN13, AN16, AN19 und AN23 die Werte der Joysticks ausliest und diese auf dem Bildschirm ausgibt. Nutze dazu die Funktionen in Tabelle 4 und folgendes Codefragment, mithilfe dessen man die Analog-Kanäle ausliest:

```
// #include "analog.h"
uint8_t analog11;
uint8_t analog12;
uint8_t analog13;
uint8_t analog16;
uint8_t analog19;
uint8_t analog23;
uint8_t analog17;
getAnalogValues(&analog11, &analog12, &analog13, &analog16, &analog17, &analog19, &
analog23);
```

- b) Um fortlaufend die Positionsdaten des Joysticks auszugeben, rufe `printValues` in einer Schleife auf:

```
#include "init.h"

int main(){
    initBoard();
    while(1) {
        printValues();
        delay(1000);
    }
}
```

Tabelle 4: Wichtige Funktionen und Variablen für die Verwendung der Joysticks

Funktionen/Variablen	Beschreibung
<code>setCursor(0, 319)</code>	Setzt den Cursor auf die linke obere Ecke
<code>void writeTextln(char *text)</code>	Schreibt <code>text</code> auf das Display und verschiebt den Cursor mit Zeilensprung
<code>void writeText(char *text)</code>	Schreibt <code>text</code> auf das Display und verschiebt den Cursor ohne Zeilensprung
<code>void writeNumberOnDisplayRight(const uint8_t *number)</code>	Schreibt die per Pointer referenzierte Zahl <code>number</code> an die Position des Cursors und verschiebt den Cursor

Aufgabe 25.2 LED mit Joystick 1 kontrollieren

In dieser Aufgabe soll die Funktion `controlLEDs` geschrieben werden, um die Farbe der RGB-LED durch die Bewegung des Joysticks 1 nach links oder rechts zu verändern. Table 5 zeigt die verschiedenen möglichen Wertebereiche mit den anzusteuernden Ports und Pins der LEDs (wie in `pins.h` definiert).

Tabelle 5: Anzeigebereiche der LEDs

Position des Joystick	LED-Farbe	Werbereich AN16	Daten-Port	Ausgabe-Pin
Links	Grün	255 ... 200	LED_GREEN_D0R	LED_GREEN_PIN
Mitte	Blau	200 ... 180	LED_BLUE_D0R	LED_BLUE_PIN
Rechts	Rot	180 ... 0	LED_RED_D0R	LED_RED_PIN

Aufgaben zu Embedded C

- a) Implementiere zunächst die Hilfsfunktion `controlLedsInit`, welche die Leitungen der RGB-LEDs initialisiert. Die analogen Kanäle der LEDs sollen ausgeschaltet werden. Definiere die Pins der LEDs als Ausgänge und initialisiere sie mit 1u (= „aus“).
- b) Implementiere nun die Funktion `controlLeds`, welche die Position der X-Achse des Joystick über den analogen Kanal AN16 ausliest und die Farbe der RGB-LED gemäß Tabelle 5 verändert.
- c) Dein Testcode sollte in etwa wie folgt aussehen:

```
#include "init.h"

int main(){
    initBoard();
    controlLedsInit();
    while(1) {
        controlLeds();
        delay(1000);
    }
}
```

Aufgabe 26 [C] Touchscreen ansteuern (optional)

Eingebaut im mit dem Mikrocontroller verbundenen Bildschirm ist eine resistive 4-Wire Touchschicht. Der Name setzt sich zusammen aus den Eigenschaften dieser Schaltung. Resistiv steht für die Messung von Widerständen zum Erkennen von Touch-Gesten und 4-Wire bedeutet, dass für diese Technik vier Datenleitungen gebraucht werden. Ein resistiver 4-Wire-Touchscreen ist wie in Abbildung 7 aufgebaut.

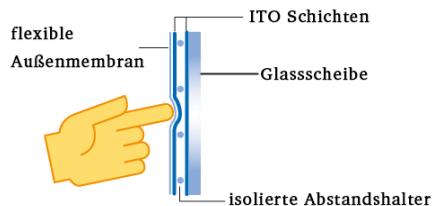


Abbildung 7: Aufbau eines resistiven Touchscreens

Dieser besteht aus a) einer Glas- oder Acryl-Schicht, b) einer äußeren resistiven Schicht, die mit Indium-Zinn-Oxid („indium tin oxide“, ITO) beschichtet ist, c) isolierenden Punkten, d) der inneren resistiven Schicht aus ITO und e) einem Polyester-Film. Die beiden resistiven Schichten sind jeweils an 2 Polen angeschlossen (Abbildung 8).

Die Schichten sind in Bezug auf ihre Pole um 90 Grad zueinander gedreht. Dies ist wichtig, um später die X- und Y-Koordinaten des Druckpunkts zu lesen. Sobald ein Objekt die oberste Glas- oder Acryl-Schicht berührt und genügend Druck ausübt, wird sich die oberste ITO-Schicht mit der unteren verbinden. Die X- und Y-Koordinate des Druckpunkts wird bestimmt, indem die Spannungen an den Polen gemessen werden. Zur Messung des X-Wertes werden X+ und X- über Gleichspannung geschaltet. Das heißt, X+ ist beispielsweise auf Vcc und X- ist mit GND verbunden. Durch die Verbindung der beiden ITO-Schichten entsteht ein Stromfluss durch beide Schichten und es kommt zu einem Spannungsteiler in der X-Schicht. Die Spannungen zwischen X+ und dem Druckpunkt sowie dem Druckpunkt und X- lassen sich durch das Ausmessen von Y- und Y+ bestimmen. Diese Information wird vom Microcontroller ausgelesen, der die gemessene Spannung in Relation zur Auflösung des Displays setzt. Die Y-Koordinate wird gemessen, indem Y+ und Y- an eine Gleichspannung gelegt und die Spannungen an X+ und X- ausgelesen werden.

Für dein Projekt stellen wir dir die Funktionen `readTouchX()`, `readTouchY()` und `readTouchZ()` zur Verfügung, die X-, Y- und Z-Werte eines Druckpunkts auf dem Touchscreen auslesen können (`analog.h`). Ist der Z-Wert größer als ein bestimmter Grenzwert, kann von einer Berührung des Touchscreens ausgegangen werden.

Aufgaben zu Embedded C

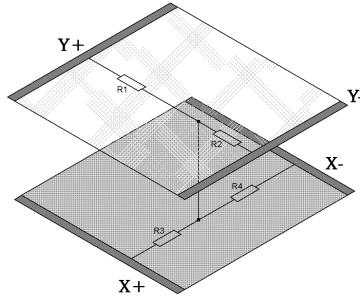


Abbildung 8: Aufbau eines 4-Wire resistiven Touchscreens

Aufgabe 26.1 Werte des Touchscreens debuggen

Implementiere zunächst die Funktion `debugTouch()`, die kontinuierlich die X-, Y- und Z-Werte des Touchscreens auf dem Bildschirm ausgibt.

Aufgabe 26.2 Zeichnen auf dem Touchscreen

In diesem Abschnitt implementierst du eine kleine Mal-Anwendung für den Touchscreen. Mit dieser soll es möglich sein, verschiedene Farben auszuwählen und mithilfe des Fingers auf dem Bildschirm zu zeichnen. Vervollständige hierfür die Funktion `paintTouch()` sowie `loopPaintTouch()`. Bereits implementiert sind die Farbpaletten und der Lösch-Button auf der unteren Seite des Bildschirms. Die Funktion `loopPaintTouch()` muss noch um eine Touch-Logik ergänzt werden, die Berührungs punkte auf dem Bildschirm erkennt und diese korrekt interpretiert. Hierbei gibt es folgende Szenarien:

- Die Farbpalette wird berührt und somit verändert sich die aktuelle Malfarbe.
- Bild erneuern wurde gedrückt und der Malbereich wird zurückgesetzt.
- Wird der freie Zeichen-Bereich berührt, so soll an dieser Stelle in der ausgewählten Farbe ein ausgefüllter Kreis mit dem Radius `PENRADIUS` gezeichnet werden.

Abbildung 9 zeigt, wie die fertige Anwendung aussehen kann.



Abbildung 9: Mal-Anwendung auf dem Touchscreen

Aufgabe 27 [C] Eigenes Microcontroller-Projekt umsetzen (optional)

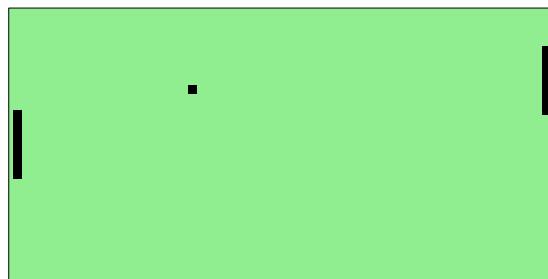
Nachdem du einige der Ein- und Ausgabemöglichkeiten des Boards kennengelernt hast, besteht deine Aufgabe in diesem Teil darin, ein kleines Projekt deiner Wahl umzusetzen. Du hast hierbei die freie Wahl, die folgenden Vorschläge sollen nur als Anregung dienen.

Aufgaben zu Embedded C

Vorschlag: Pong

Zwei Gegner sollen je einen Balken (Rechteck) am linken oder rechten Rand des Spielfeldes mit den Schieberegeln steuern können, um einen Ball (ein Quadrat) im Spiel zu halten. Erreicht der Ball den linken oder rechten Rand des Spielfelds, so bekommt der Spieler auf der anderen Seite einen Punkt und der Ball wird an seine Anfangsposition (die Mitte des Spielfelds) zurückversetzt. Erreicht der Ball den oberen oder unteren Rand sowie einen der Balken der Spieler, so wird der Ball reflektiert - verlässt also niemals das Spielfeld.

Gewonnen hat der Spieler, der zuerst eine definierte Anzahl an Punkten erreicht. Der aktuelle Punktestand könnte ebenfalls auf dem Display angezeigt werden.

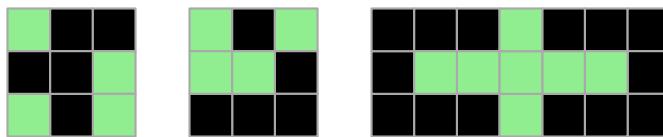


Vorschlag: Game of Life

„Game of Life“³⁵ besteht aus einem zweidimensionalen Spielfeld. Jedes Feld steht für eine Zelle, die *tot* oder *lebendig* ist. Die Farben für den Zustand kannst du natürlich frei wählen. Jede Zelle hat acht Nachbarzellen, die ebenso tot oder lebendig sein können. Zu Beginn gibt es eine vordefinierte Anfangsgeneration. Durch festgelegte Regeln wird die nachfolgende Generation ermittelt:

- Eine **lebende Zelle** ...
 - mit 1 oder 0 lebenden Nachbarn stirbt aus Einsamkeit.
 - mit 4 oder mehr lebenden Nachbarn stirbt wegen Übervölkerung.
 - mit 2 oder 3 lebenden Nachbarn bleibt am Leben.
- Eine **tote Zelle** mit genau 3 lebenden Nachbarn wird in der nächsten Generation geboren werden, andernfalls bleibt sie tot.

Als Anfangsgeneration eignen sich zufällige Populationen oder eine der folgenden Figuren:



Hinweise

- Da das Spielfeld begrenzt ist, soll es torusförmig aufgebaut werden. Das heißt: Alles, was am unteren Rand des Spielfelds verschwindet, kommt oben wieder heraus – das gleiche gilt für den linken und rechten Rand.
- Verwende als Spielfeld ein mehrdimensionales Array
- Ein weiteres mehrdimensionales Array bietet sich an, um die zukünftige Generation erzeugen zu können.
- Achte beim torusförmigen Feld unbedingt darauf, dass du nicht über die Grenzen des Spielfelds hinaus zugreifst! Das kann zu unvorhersehbarem und schwer zu debuggendem Verhalten des ganzen Displays führen!

³⁵ siehe auch http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Aufgaben zu Embedded C

Vorschlag: Regentropfen

Das Touch-Display ist ein kleiner Teich; wenn du eine Stelle mit dem Finger berührst, breitet sich von dort eine konzentrische Welle aus. Die Geschwindigkeit der Welle kannst du zusätzlich abhängig machen vom ausgeübten Druck.

Weitere Tipps und Vorschläge

Asteroids

<https://goo.gl/aE7mgC>

Ausweichspiele à la Hugo

<https://goo.gl/Ab8Go2>

Pacman

<https://goo.gl/kXthKj>

Moorhuhn

<https://goo.gl/X2xems>

Labyrinth

<https://goo.gl/VCf85t>

Snake

<https://goo.gl/iL2L5M>

Hinweise

- Der folgenden Wiki-Artikel beschreibt, wie man schnell bestehende Bilder in das eigene Projekt einbinden kann:
<https://github.com/Echtzeitsysteme/tud-cpp/wiki/RGB565-mit-Gimp>.

Zusatzaufgaben zum Aufzugsimulator

Aufgabe 28 [A] Aufzugsimulator – Teil 1 (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

In dieser Aufgabe soll ein Grundgerüst für den in der Vorlesung vorgestellten Aufzugsimulator geschaffen werden. Bei der Bearbeitung dieser Aufgaben geben wir dir keinerlei zeitliche Vorgabe. Du kannst diese Aufgaben direkt an dem Tag lösen, an dem ihr die dafür benötigten Konzepte gelernt habt oder auch nach hinten verschieben und als ausführliche Klausurvorbereitung nutzen.

Aufgabe 28.1 Klasse Person

Implementiere die Klasse Person, die eine Person mit einem gewünschten Zielstockwerk darstellt. Füge allen Konstruktoren und Destruktoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Person {
public:
    Person(int destinationFloor);           // create a person with given destination
    Person(const Person& other);           // copy constructor
    ~Person();                            // destructor
    int getDestinationFloor() const;       // get the destination floor of this person
private:
    int destinationFloor;                 // destination floor of this person
};
```

Aufgabe 28.2 Klasse Elevator

Implementiere die Klasse Elevator, die einen Aufzug mit einer beliebigen Anzahl an Personen darstellt. Wenn sich der Aufzug bewegt, solltest du die verbrauchte Energie bei einer Bewegung sinnvoll anpassen. Addiere beispielsweise den Betrag der Differenz zwischen dem aktuellen und dem Zielstockwerk hinzu.

```
class Elevator {
public:
    Elevator();                           // create an elevator at floor 0, no people
    inside and 0 energy consumed
    int getFloor();                      // get number of floor the elevator is
    currently at
    double getEnergyConsumed();          // get consumed energy
    void moveToFloor(int floor);         // move the elevator to given floor (consumes
    energy)
    int getNumPeople();                 // get number of people in Elevator
    Person getPerson(int i);             // get i-th person in Elevator
    void addPeople(std::vector<Person> people); // add people to Elevator
    std::vector<Person> removeArrivedPeople(); // remove people which arrived
private:
    int currentFloor;                   // current floor number
    std::vector<Person> containedPeople; // people currently in elevator
    double energyConsumed;              // energy consumed
};
```

Um die Klasse `std::vector` aus der Standardbibliothek zu nutzen musst du noch den Systemheader `vector` einbinden. Der Container `std::vector` kapselt ein Array und stellt eine ähnliche Funktionalität wie Javas `Vector` Klasse bereit.

Zusatzaufgaben zum Aufzugsimulator

Der Typ in spitzen Klammern (<Person> in std::vector<Person>) ist ein Template-Parameter und besagt, dass in dem Container Person-Objekte gespeichert werden sollen.

Folgende Funktion der Klasse std::vector könnten dir von Nutzen sein. Weitere findest du z.B. unter <http://www.cplusplus.com/reference/vector/vector/>. Der Typ size_type ist der größtmögliche vorzeichenloser Integer (`unsigned int`), den die verwendete Plattform unterstützt (bspw. 16, 32 oder 64 Bit).

```
size_type size() const;                                // get size of the vector
reference at(size_type n);                            // get the i-th element of the vector
void push_back(const value_type& val);               // add an element to the vector
void clear();                                         // remove all elements from the vector
```

Hinweise

- Da containedPeople leer initialisiert werden soll, brauchst du dafür keinen expliziten Aufruf in der Initialisierung.
- Um die Leute aussteigen zu lassen, die an ihrem Zielstockwerk angekommen sind, erstelle in der Methode zwei temporäre std::vector-Container stay und arrived. Iteriere nun über alle Leute im Aufzug und prüfe, ob das Zielstockwerk der Person mit dem aktuellen Stockwerk des Aufzugs übereinstimmt. Wenn ja, lasse die Person aussteigen, indem du sie zu der arrived-Liste mittels `push_back()` hinzufügst. Andernfalls muss die Person im Aufzug verbleiben (stay-Liste). Gib am Ende die arrived-Liste zurück, und ersetze containedPeople durch stay.

Aufgabe 28.3 Klasse Floor

Implementiere die Klasse Floor, die ein Stockwerk mit einer beliebigen Anzahl an wartenden Personen darstellt.

```
class Floor {
public:
    int getNumPeople();                                     // get the number of people on this floor
    Person getPerson(int i);                               // get the i-th person on this floor
    void addWaitingPerson(Person h);                      // add a person to this floor
    std::vector<Person> removeAllPeople();                // remove all persons from this floor
private:
    std::vector<Person> containedPeople;                 // persons on this floor
};
```

Aufgabe 28.4 Klasse Building

Schreibe eine Klasse Building, die einen Aufzug besitzt der sich zwischen einer definierbaren Menge an Stockwerken bewegt und Personen befördert.

```
class Building {
public:
    Building(int numberOffFloors);           // create a Building with given number of floors
    int getNumOfFloors();                    // get number of floors
    Floor& getFloor(int floor);             // get a certain floor
    Elevator& getElevator();                // get the elevator
private:
    std::vector<Floor> floors;              // floors of this building
    Elevator elevator;                     // the elevator
};
```

Zusatzaufgaben zum Aufzugsimulator

Aufgabe 28.5 Komfortfunktionen

Erweitere die Klasse Building um folgende **public** Funktionen, um die Benutzung des Simulators von außen zu vereinfachen und lange Aufrufketten wie

```
b.getElevator().addPeople(b.getFloor(b.getElevator().getFloor()).removeAllPeople());
```

zu vermeiden (*Law of Demeter*³⁶). Der Simulator sollte nur mit Methoden der Klasse Building kommunizieren.

```
void letPeopleIn();                                // let people on current floor into
                                                     elevator
void moveElevatorToFloor(int i);                  // move the elevator to a given floor
void addWaitingPerson(int floor, Person p); // add a person to a given floor
std::vector<Person> removeArrivedPeople(); // remove people which arrived at their
                                             destination from the elevator on the current floor
```

Aufgabe 28.6 Beförderungsstrategie

Teste deine Implementation. Erstelle dazu zunächst ein Gebäude und füge einige Personen hinzu.

```
Building b(3);
b.addWaitingPerson(0, Person(2)); // person in floor 0 wants to floor 2
b.addWaitingPerson(1, Person(0)); // person in floor 1 wants to floor 0
b.addWaitingPerson(2, Person(0)); // person in floor 2 wants to floor 0
```

Implementiere nun folgende Beförderungsstrategie. Diese sehr einfache (und ineffiziente) Strategie fährt alle Stockwerke nacheinander ab, sammelt die Leute ein und befördert sie jeweils zu ihren Zielstockwerken.

```
for Floor floor in Building do
    Move elevator to Floor floor;
    Let all people on floor into elevator;
    while elevator has people do
        Move Elevator to destination Floor of first Person in Elevator;
        Remove arrived people;
    end
end
```

Gib am Ende auch die verbrauchte Energie aus. Schau dir die Ausgabe genau an und versuche nachzuvollziehen, warum Personen so oft kopiert werden. Denke daran, dass diese bei einer Übergabe als Argument kopiert werden.

Hinweise

- Falls du wie vorgeschlagen als Modell für den Energieverbrauch den Betrag der Differenz der abgefahrenen Stockwerke verwendest, solltest du am Ende consumedEnergy = 8 erhalten.

³⁶ https://en.wikipedia.org/wiki/Law_of_Demeter

Zusatzaufgaben zum Aufzugsimulator

Aufgabe 29 [A] Aufzugsimulator – Teil 2 (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

In dieser Aufgabe erweitern und verbessern wir unseren Aufzugsimulator, sodass das Kopieren von Personen wegfällt. Dies werden wir erreichen, indem wir nicht direkt mit Person-Objekten oder -rohzeigern sondern mit Smart Pointern arbeiten. Dadurch müssen wir beim Verschieben von Personen in den Aufzug nur die Smart Pointer kopieren, während die Person-Objekte selbst bestehen bleiben.

Ein weiterer Vorteil ist, dass wir von jeder Person genau ein Exemplar im Speicher halten. Möchten wir beispielsweise den Namen einer Person ändern, ist dies überall, wo die Person auftaucht sofort und konsistent sichtbar. Nutzt man überall Kopien von Personen, haben wir keine Kontrolle darüber und wären gezwungen die Klasse Person immutabel zu machen.

Hinweise

- Am Ende dieser Aufgabe hast du die Möglichkeit, die Performanz der alten und der neuen Implementation zu vergleichen. Dazu ist es nötig, dass du jetzt eine Kopie von deinem aktuellen Code machst. Gehe dabei folgendermaßen vor:
 - Lege eine Kopie deines Projektordners an.
 - Benenne dein Projekt um, z.B. in Aufzug_Alt (**Rechtsklick auf das Projekt → Rename Project**).
 - Importiere den kopierten Projektordner in CodeLite (**Workspace → Add an existing project**).
 - In deinem Workspace sollten jetzt zwei Projekte sein, Aufzug_Alt und Aufzug.
- Du wirst in den folgenden Aufgaben deinen Code stark verändern. Versuche, nicht alle Änderungen auf einmal zu machen, sondern gehe stückweise vor (also bspw. nur einen Getter umstellen und alle Folgefehler beheben).

Aufgabe 29.1 Refactoring mit Referenzen und `const`

Als Erstes verbessern wir die Sauberkeit des vorhandenen Codes mithilfe der bisher kennengelernten Mittel wie Referenzen und `const`. Es ist sinnvoll, dass du die Änderungen stückweise im Code durchführst und zwischendurch testest, ob alles noch korrekt funktioniert.

Deklariere dafür sämtliche Getter in `Building`, `Elevator`, `Floor` und `Person` als `const`, z.B. `Building::getFloor()` und `Elevator::getEnergyConsumed()`. Passe außerdem die Methode `Elevator::addPeople()` so an, dass die Liste `people` nicht mehr als Wert sondern als `const` Referenz übergeben wird.

Es kann sein, dass du für bestimmte Zwecke weiterhin einen Getter brauchst, der dir ein nicht-`const` Objekt zurückgibt. Solche Getter sollten typischerweise `private` deklariert werden. Beispielsweise wird deine `Building`-Klasse zwei Getter für `Floor` enthalten:

```
class Building {
public:
    // ...

    /** Gets a certain, const floor */
    const Floor& getFloor(int floor) const;

private:
    /** Returns the floor with the given number.
```

Zusatzaufgaben zum Aufzugsimulator

```
* This non-const variant of the getter is for 'private' purposes only.  
*/  
Floor& getFloor(int floor);  
  
// ...  
};
```

Hinweise

- Um über eine `const` Liste zu iterieren, verwende `vector<T>::const_iterator` anstatt `vector<T>::iterator` als Iterator-Typ.

Aufgabe 29.2

Um nicht immer wieder `std::shared_ptr<Person>` schreiben zu müssen, definiere ein `typedef` `PersonPtr` für diesen Typen. Binde den Header `memory` in `Person.h` ein und definiere den neuen Typen `PersonPtr` hinter der Klassendefinition von `Person`:

```
typedef std::shared_ptr<Person> PersonPtr;
```

Aufgabe 29.3 Effizientere Listen

Ändere in der Klasse `Elevator` alle Vorkommen von `vector` nach `list` um, da wir nun eine verkettete Liste verwenden werden, um Personen zu speichern. Dadurch kann man Personen auch in der Mitte der Liste effizient löschen.

Die `list`-Klasse enthält keine Methode `at()`. Diese ist auch gar nicht nötig: Wir traversieren die Liste stattdessen mit einem Iterator. Lösche dazu die Methode `getPerson()` und füge die folgende Methode hinzu, die eine `const` Referenz auf die enthaltenen Personen zurückgibt:

```
/** return a const reference to the list of contained people */  
const std::list<PersonPtr>& getContainedPeople() const;
```

Dadurch kann von außen lesend auf die Leute im Aufzug zugegriffen werden. Ändere außerdem den Typen des Containers von `Person` auf `PersonPtr`, da wir Smart Pointer auf Personen speichern werden und nicht die Personen direkt. Passe die Signaturen aller Methoden in `Elevator` entsprechend an.

Aufgabe 29.4

Jetzt müssen wir die Methode `removeArrivedPeople()` anpassen. Da wir beliebige Elemente aus `containedPeople` löschen können, brauchen wir den Umweg über die temporäre Liste `stay` nicht mehr.

Gehe dazu folgendermaßen vor: Iteriere mit einem ListenIterator vom Typ `std::list<PersonPtr>::iterator` über die Personen im Aufzug und prüfe für jede, ob sie an ihrem Zielstockwerk angekommen ist. Du kannst auf das Element, auf das der Iterator zeigt, durch den Dereferenzierungsoperator (`*iter`) zugreifen. Dieses Element ist selbst ein Smart Pointer. Deshalb muss der Iterator für den Zugriff auf die Person **doppelt** dereferenziert werden. Wenn die Person in ihrem Zielstockwerk angekommen ist, wird sie aus `containedPeople` gelöscht und zu `arrived` hinzugefügt. Um ein Element zu löschen, verwende `containedPeople.erase(iter)`.

Hinweise

- Der bisherige Iterator ist nach dem Löschen nicht mehr gültig. Die Methode `std::vector<T>::erase` gibt einen Iterator auf das Element hinter dem gelöschten zurück.

Zusatzaufgaben zum Aufzugsimulator

Als Grundgerüst kann folgendes Codeschnipsel dienen:

```
... iter = containedPeople. ...;      // create iterator for containedPeople
// iterate through all elements
while (iter != ...) {
    PersonPtr person = ... iter;      // get person smart pointer at current position
    // check whether person has reached it's destination Floor
    if (...) {
        // erase person pointer from containedPeople
        // no need for ++iter since iter will already point to next item
        ... = containedPeople.erase(iter);
        // remember arrived person
        ...
    }
    else {
        ++iter; // check next person
    }
}
```

Aufgabe 29.5

Passe auch die Klassen Floor und Building entsprechend an, sodass nur noch Listen und Smart Pointer auf Personen verwendet werden.

Aufgabe 29.6

Passe die Simulation des Aufzugs entsprechend an. Du wirst auf die erste Person im Aufzug nun auf eine andere Art und Weise zugreifen müssen als vorher. Benutze die Methode `getContainedPeople()` des Aufzugs, um an die Liste der Personen zu kommen. Nun kannst du auf den Inhalt des ersten Elements mittels `front()` zugreifen. Vergiss nicht, dass dieser Inhalt ein `PersonPtr` und nicht die Person direkt ist. Entweder du dereferenzierst das Element doppelt und verwendest den Operator `.` oder du nutzt wie üblich bei Pointern den Operator `->`.

Schau dir die Ausgabe an. Nun werden Personen nicht mehr kopiert, sondern nur noch gelöscht, sobald sie tatsächlich den Aufzug verlassen haben und kein Zeiger mehr auf sie zeigt.

Aufgabe 29.7 Vergleich der alten und neuen Implementation (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Es ist natürlich interessant zu erfahren, ob sich der ganze Aufwand des Refactorings gelohnt hat.

Laufzeit

Eine relativ simple Art der Laufzeitmessung ist es, die verstrichene Prozessorzeit zu messen. Der Header `ctime` stellt hierfür die Funktion `clock()` zur Verfügung, die einen Zähler vom Typ `clock_t` zurückgibt. Mithilfe der Konstanten `CLOCKS_PER_SEC` kann man aus der Anzahl von Prozessorzyklen die Laufzeit berechnen.

Erzeuge nun ein hinreichend großes Beispiel und teste dessen Laufzeit für die alte und neue Implementation.

Hinweise

- Es gibt auch ausgefeilte Möglichkeiten, die Laufzeit zu messen. Dazu stellt Boost unter anderem den Header `boost/chrono.hpp` zur Verfügung. Für nähere Informationen siehe http://www.boost.org/doc/libs/1_48_0/doc/html/chrono/users_guide.html.

Zusatzaufgaben zum Aufzugsimulator

Speicherverbrauch

Ein weiteres Argument gegen das Kopieren von Objekten kann der Speicherverbrauch sein. Das ist in unserem Fall allerdings weniger interessant, da die meisten kopierten Objekte relativ kurz leben und dann wieder gelöscht werden.

Im Gegensatz zur Laufzeit gibt es leider keinen sehr einfachen Weg, den Speicherverbrauch des Programms direkt auszugeben.

Du könntest hierfür kurz vor dem Ende von `main` die Ausführung pausieren (mittels `std::cin`) und dir im Task Manager ansehen, wie hoch der Speicherverbrauch des Programms ist – das ist aber sicherlich nur eine Notlösung.

Zusatzaufgaben zum Aufzugsimulator

Aufgabe 30 [A] Aufzugsimulator – Teil 3 (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Unser bisheriger Aufzugsimulator hat eine feste Strategie, nach der die einzelnen Stockwerke abgefahrene werden. Mit Hilfe von Polymorphie können wir den Simulator so erweitern, dass die Strategie austauschbar wird.

Aufgabe 30.1 Vorbereitung

Lagere die bereits existierende Simulation des Aufzugs aus der main-Funktion in eine eigene Funktion runSimulation() aus. Die Funktion sollte das volle Gebäude als Parameter entgegennehmen und eine Liste (std::list<int>) der angefahrenen Stockwerke zurückgeben. Überlege dir, auf welche Art das Gebäude idealerweise übergeben werden sollte. Teste deine Implementation.

Aufgabe 30.2 Klasse ElevatorStrategy

Implementiere die Klasse ElevatorStrategy. Diese soll die Basisklasse für verschiedene Aufzugstrategien sein. Damit die Strategie das Gebäude nicht selbst modifizieren kann, wird Building per **const** Pointer übergeben.

```
// Elevator strategy class: Determines to which floor the elevator should move next.
class ElevatorStrategy {
public:
    virtual ~ElevatorStrategy();
    virtual void createPlan(const Building*); // create a plan for the simulation - the
                                                // default implementation does nothing but saving the building pointer
    virtual int nextFloor() = 0; // get the next floor to visit
protected:
    const Building *building; // pointer to current building, set by createPlan()
};
```

Aufgabe 30.3 Eine einfache Aufzugstrategie

Implementiere eine einfache Aufzugstrategie, indem du eine neue Klasse erzeugst, die von ElevatorStrategy erbt. Diese soll folgendermaßen vorgehen: Falls der Aufzug momentan leer ist, soll zum tiefsten Stockwerk gefahren werden, wo sich noch Personen befinden. Falls der Aufzug nicht leer ist, wird das Zielstockwerk einer der Personen im Aufzug ausgewählt.

Aufgabe 30.4 Implementation von runSimulation

Ändere nun runSimulation() entsprechend um, sodass die Simulation anhand der gegebenen Strategie durchgeführt wird. Folgender Pseudocode kann dir als Denkhilfe dienen:

```
while People in Building or Elevator do
    Calculate next floor;
    Move Elevator to next floor;
    Let all arrived people off;
    Let all people on floor into Elevator;
end
```

Teste die einfache Aufzugstrategie

Zusatzaufgaben zum Aufzugsimulator

Aufgabe 30.5 Neue Aufzugstrategien (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Entwickle eigene Aufzugstrategien, indem du erneut eine neue Klasse erzeugst die von `ElevatorStrategy` erbt. Versuche, verschiedene Größen zu optimieren, wie z.B. die Anzahl der Stopps oder die verbrauchte Energie. Hierfür könnte Backtracking verwenden³⁷, eine einfache Methode, um eine optimale Lösungen durch Ausprobieren zu finden. Beachte, dass der Aufzug auch kopiert werden kann, um verschiedene Strategien zu testen.

Aufgabe 30.6 Schlüsselwort auto (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

In der Vorlesung wurde das in C++11 neu hinzugekommene Schlüsselwort `auto` angesprochen. Dieses Schlüsselwort kann überall dort anstelle eines Typbezeichners eingesetzt werden, wo der Compiler den Typ automatisch ableiten kann. Das ist besonders dann sinnvoll, wenn die Typbezeichnung aufgrund geschachtelter Templates oder geschachtelter Namensräume besonders lang und unleserlich wird, bspw. `const std::vector<std::pair<const Person*, boolean (Floor, int)>>::const_iterator x;`.

Versuche, an möglichst vielen Stellen im Aufzugsimulator das Schlüsselwort `auto` zu verwenden.

³⁷ Siehe <http://de.wikipedia.org/wiki/Backtracking>