

C++ EINFÜHRUNG

Programmieren für Anfänger

Dieses Skript gibt eine Einführung in die Softwareentwicklung mit Hilfe der Programmiersprache C++. Es werden Sprachelemente bis zur Objektorientierung dargestellt. Jeder Abschnitt ist mit Kontrollfragen und Aufgaben ergänzt.

Jakob Bauer

*cpp@bauer-sw.de
www.bauer-sw.de*

Jan-19

Inhaltsverzeichnis

1	Vorlesung	4
1.1	Ziele	4
1.2	Anmerkungen.....	4
1.3	Aufgaben.....	4
2	Überblick	5
2.1	Was ist C / C++?	5
2.2	Unterschied zwischen C und C++	6
2.3	Entwicklungsprozess.....	6
2.4	Darstellungsformen	7
2.4.1	Struktogramm.....	7
2.4.2	Programmablaufplan	8
2.4.3	UML Diagramme	9
2.5	Codierung	10
3	Einführung.....	11
3.1	Header-Dateien.....	12
3.2	Einstiegspunkt.....	12
3.3	Namensbereiche	14
3.4	Datentypen.....	15
3.5	Variablen.....	16
3.6	Berechnungen.....	17
3.7	Enumerationen.....	19
3.8	Aufgabe: Grundrechenarten	20
3.9	Funktionen und Methoden.....	21
3.10	Ein- und Ausgabe.....	22
3.10.1	Konsole.....	23
3.10.2	Datei	23
3.11	Aufgabe: Variablen Initialisierung	24
3.12	Aufgabe: Kreisberechnung	24
3.13	Aufgabe: Benzinverbrauch	25
3.14	Aufgabe: Winkelfunktionen.....	25
4	Kontrollstrukturen	26
4.1	Bedingungen.....	26
4.2	Wenn-Dann (if-else)	27

4.3	Aufgabe: Gasverbrauch	28
4.4	Behauptungen.....	28
4.5	Aufgabe: Parallelschaltung.....	29
4.6	Aufgabe: Quadratische Gleichung.....	29
4.7	Aufgabe: Tageszins.....	30
4.8	Schalter (switch-case)	30
4.9	Aufgabe: Taschenrechner	31
4.10	Aufgabe: Mehrwertsteuer	31
4.11	Schleifen	31
4.11.1	Kopfgesteuerte-Schleife (while).....	31
4.11.2	Fußgesteuerte-Schleife (do-while).....	32
4.11.3	Zählschleife (for)	32
5	Zeiger, Felder und Zeichenketten.....	34
5.1	Zeiger (Pointer)	34
5.2	Felder (Array).....	34
5.2.1	Statische Felder	34
5.2.2	Dynamische Felder	35
5.3	Aufgabe: Histogramm.....	36
5.4	Zeichenketten.....	37
5.5	Aufgabe: Palindrom.....	38
5.6	Weitere Datenstrukturen	38
6	Objektorientiertes Programmieren.....	40
6.1	Klassen definieren.....	40
6.2	Vererbung	44
6.3	Aufgabe: Rechtschreibung	46
6.4	Aufgabe: Würfel	46
7	Vorlagen (Templates)	48
8	Umwandeln (Casts).....	50
8.1	static_cast	50
8.2	dynamic_cast	50
8.3	const_cast.....	50
8.4	reinterpret_cast	51
9	Operatoren	52
10	(Fehler)quellen.....	53

11	Aufgabenpool	54
11.1	Aufgabe: Fibonacci Folge	54
11.2	Aufgabe: Datenkapselung	55
11.3	Aufgabe: Übergabearten	55
11.4	Aufgabe: Blaue Donau	57
11.5	Aufgabe: Vektor	58
11.5.1	Einfacher Vektor mit dem Datentyp int	58
11.5.2	Copy-On-Write Mechanismus	59
11.5.3	Vektor für beliebige atomare Datentypen	60
11.6	Aufgabe: Bildbearbeitung	61
11.7	Aufgabe: Sudoku	62
11.8	Aufgabe: Prüfsumme	63
11.9	Aufgabe: Scaleable Vector Graphic	64
11.9.1	Anforderungen	65
11.9.2	Kurzreferenz	65

1 Vorlesung

Die weit verbreitete objektorientierte Programmiersprache **C++** eignet sich vor allem zum Entwickeln hocheffizienter Software für technisch-wissenschaftliche und hardwarenahe Aufgaben.

Die Programmiersprache **C++** ist als eine "hybride" Erweiterung der Programmiersprache **C** zu sehen. Es ist möglich **C** Programme in **C++** einzubinden.

Dieser Kurs legt seinen Schwerpunkt in praktische Übungen mit der Programmiersprache **C++**.

Es werden verschiedene Lösungswege diskutiert und bewährte Tipps aus der Praxis vermittelt.

1.1 Ziele

Die Studierenden kennen ...

- ... den Unterschied zwischen **C** und **C++**.
- ... die Grundelemente der objektorientierten Programmierung.
- ... die Syntax und Semantik der Sprache **C++** und können ein Softwaredesign selbstständig entwerfen, codieren und ihr Programm auf Funktionsfähigkeit testen.
- ... verschiedene Designpattern sowie vorhandene Datenstrukturen und können diese exemplarisch anwenden.
- ... die Fähigkeiten der Entwicklungsumgebung und können diese sinnvoll einsetzen.

1.2 Anmerkungen

Jeder hier vorgestellte Code ist eine Empfehlung, Idee oder eine mögliche Lösung zum Ziel. In der Informatik gibt es nicht (immer) die eine richtige Lösung. Es ist abzuwägen, welcher Lösungsweg am geeignetsten ist.

Dieses Skript hat den Anspruch eine Einführung in die Programmiersprache C++ zu geben, sowie einen kleinen Einblick in die STL. Tiefere Kenntnisse über die Sprachmerkmale (C++ Standards) sowie STL Fähigkeiten sind in Fachliteraturen/Online Hilfen zu finden.

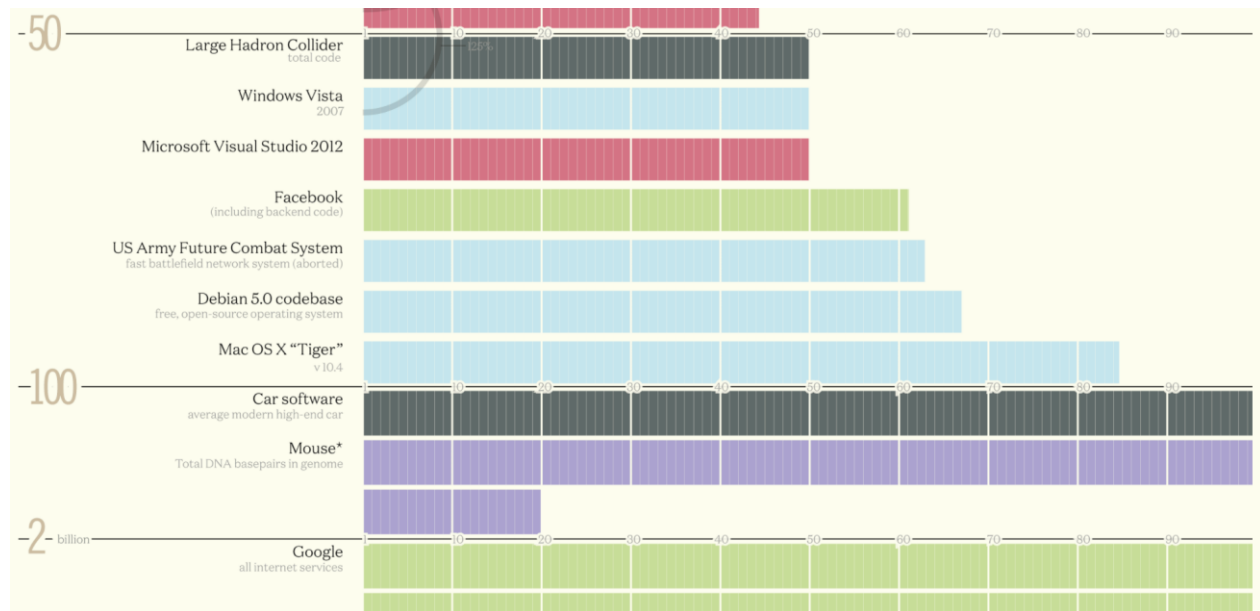
Die Inhalte basieren auf dem **C++** Skript der technischen Ausbildung von Airbus Friedrichshafen, meiner Berufserfahrung und meinen Programmierkenntnissen.

1.3 Aufgaben

Die dargestellten Aufgaben sind so gestellt, dass das Wissen in den vorherigen Kapiteln bereits diskutiert wurden. Zur jeder Aufgabe wird empfohlen vordem starten der Entwicklungsumgebung sich bereits Gedanken über den Ablauf zu machen, wie z.B. mit einem Struktogramm.

2 Überblick

Das heutige Leben ist ohne Software nicht mehr vorstellbar, selbst der Fernseher, Kühlschrank oder Rauchmelder benötigen regelmäßige Softwareupdates. In der folgenden Grafik wird dargestellt aus wieviel Millionen Zeilen Code ein Softwaresystem bestehen kann.



Wie in dem Ausschnitt der Grafik von <https://informationisbeautiful.net/visualizations/million-lines-of-code/> dargestellt, sind mit die am Meisten geschriebenen Codezeilen in einem Auto zu finden. Der Spitzenreiter ist Google mit über 2 Milliarden Codezeilen.

Im Weiterem wird ein Überblick über die Programmiersprache C/C++ und Darstellungsformen in der Softwareentwicklung gegeben.

2.1 Was ist C / C++?

C++ ist eine universale Programmiersprache, die von der populären Sprache **C** abstammt. Mit geringen Einschränkungen enthält **C++** den gesamten Sprachumfang von **C** und unterscheidet sich hauptsächlich durch die Verwendung von Klassen von der Programmiersprache **C**. Mit Hilfe von Klassen lassen sich komplexe Zusammenhänge einfacher darstellen. **C++** ist eine sogenannte objektorientierte Sprache, da sie die objektorientierte Programmierung ermöglicht, **C** dagegen ist eine strukturierte Programmiersprache.

Entstanden ist **C++** in den frühen 80er Jahren und wurde für Unix-Maschinen entwickelt. Bjarne Stroustrup entwickelte 1982 „C with Classes“, ein Jahr später **C++** als Erweiterung zu **C**. Ein erstes Release von **C++** wurde 1985 freigegeben, 1988 erschienen die ersten kommerziellen Compiler. Letztendlich wurde **C++** im Jahr 1998 als ISO/IEC 14882 standardisiert.

In den vergangenen Jahren hat das **C++** Komitee neue Standards verabschiedet. Die wesentliche Verbesserung besteht in der Erweiterung des Sprachumfangs (z.B. Lambda Funktionen und Multi-Threading) von **C++**. 2012 wurde der **C++11** Standard als ISO/IEC 14882:2011 verabschiedet. Anfang dieses Jahrs wurde der **C++14** Standard in der Norm ISO/IEC 14882:2014 veröffentlicht.

Eine Auflistung der Verbesserungen zum C++ Standard von C++98 und dem C++11 Standard sind auf heise.de/developer zu finden.

In den Musterlösungen dieses Skripts ist größtenteils mit den C++98 Standard kompatibel. In wenigen Zeilen wird der C++11 Standard und die neue STL Fähigkeiten verwendet.

2.2 Unterschied zwischen C und C++

C <i>Prozedurale, strukturierte Programmierung</i>	C++ <i>Objektorientierte Programmierung</i>
Programmablauf steht im Vordergrund.	Das Wesen der OOP besteht in der Behandlung der Daten und Prozeduren als Objekt.
	Das Programm ist ein Verbund kommunizierender Objekte.
Die Trennung von Daten und Prozeduren ist mit zunehmender Datenmenge schwieriger.	Objekt vereint Datenbestand und Funktionalität.
Häufig werden neue Lösungen für alte Probleme gesucht.	Das Objekt bietet Dienste (Methoden) an, die z.B. zur Datenmanipulation verwendet werden können.

2.3 Entwicklungsprozess

Der Entwicklungsprozess eines jeden Programms besteht grundsätzlich aus drei Schritten:

Schritt 1: Problemanalyse

Exakte Formulierung des Problems sowie die Darstellung einer möglichen Lösung in einem Struktogramm, Programmablaufplan, UML-Diagramm oder einer einfachen Skizze.

Die Problemanalyse ist notwendig, um bei der Codierung das Problem exakt zu lösen. Wird das Problem bei der Analyse nicht richtig erfasst, so kann die spätere Softwarearchitektur die Lösung nicht korrekt abbilden.

Während der Problemanalyse können unter anderem folgende Fragen gestellt werden:

- Was ist das exakte Problem?
- Ist es möglich, das Problem in Teilprobleme aufzugliedern?
- Wie könnte sich das Problem in naher Zukunft gestalten?
- Wie sehen die möglichen Lösungen aus?
- Existiert für das Problem bereits eine Lösung? Wie hoch ist der Aufwand eine bereits bestehende Lösung an das Problem anzupassen bzw. einzubinden?
- Welche Darstellungsform ist die richtige, um die angestrebte Lösung zu visualisieren?
- Welche Variablen existieren?
- Was sind die Wertebereiche der Variablen?

Schritt 2: Codierung und Test

Hier findet das sogenannte programmieren statt. Nach jeder (Teil-) Entwicklung, sollte diese getestet werden. Es gibt unterschiedliche Testmethodiken (Black-Box-Test, Unit-Test, usw.).

Schritt 3: Dokumentation

Dieser Teil wird leider oft vernachlässigt, gehört aber grundsätzlich zu den wichtigsten Bestandteilen eines Programmes.

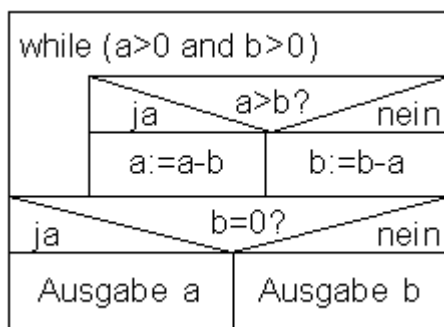
2.4 Darstellungsformen

Es gibt unterschiedliche Darstellungsformen von Lösungen. Es sollte eine passende Darstellung für die angestrebte Lösung ausgewählt werden. Dazu gibt es folgende Leitlinien:

- **Struktogramm oder Programmablaufplan:** Darstellung eines (kompakten) Algorithmus. Stammt aus der Zeit von imperativen Programmiersprachen.
- **UML Diagramm:** Es existieren unterschiedliche UML-Diagramme, je nach Anwendungsfall kann hierbei eines davon nützlich sein. UML sollte der "Standard" sein, um in Entwicklerteams objektorientierte Softwarearchitekturen, Ablaufpläne, Prozesse usw. zu beschreiben.
- **Skizze:** Bei einer unkomplizierten Problemstellung reicht meistens eine Skizze. Eine Skizze muss sich nicht an Standards halten und bietet somit die Möglichkeit schnell eine Lösung mit Stift und Papier aufzuzeichnen.

2.4.1 Struktogramm

Das folgende Struktogramm zeigt den euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teilers:



Der dazugehörige C++ Code Repräsentation sieht wie folgt aus:

```
// laufe solange a und b größer 0 sind
while( a > 0 && b > 0 ) {
    // wenn a größer b ist ...
    if( a > b ) {
        // ... dann ziehe b von a ab
        a = a - b;
    }
}
```



```

    } else {
        // ... sonst ziehe a von b ab
        b = b - a;
    }
}

// prüfe ob b den Wert 0 besitzt
if( b == 0 ) {
    // Ausgabe der Variable a an die Konsole.
    std::cout << a;
} else {
    // Ausgabe der Variable a an die Konsole.
    std::cout << b;
}

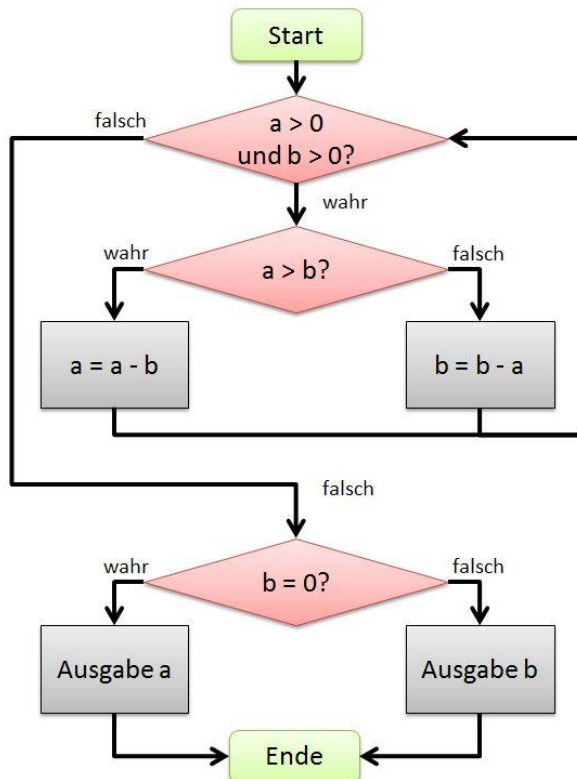
```

Eine Übersicht über die verfügbaren Elemente eines Struktogramms finden sie unter folgendem Link, sowie eine Auswahl von Software:

<https://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm>

2.4.2 Programmablaufplan

Der Programmablaufplan des euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teilers sieht wie folgt aus:



Weitere Informationen zum Programmablaufplan sind unter <https://de.wikipedia.org/wiki/Programmablaufplan> zu finden. Programmablaufpläne können unter anderem in Microsoft Power Point erstellt werden.

2.4.3 UML Diagramme

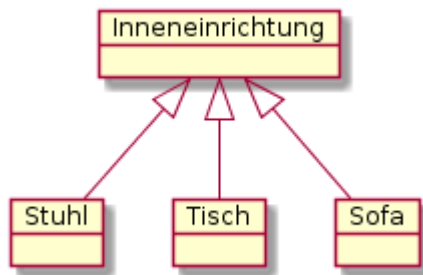
Im Rahmen dieses Kurses wird auf UML nicht näher eingegangen. Die verwendeten Skizzen lehnen sich an die UML-Darstellungen an.

Weitere Informationen zu UML sind auf folgenden Webseiten zu finden:

- https://de.wikipedia.org/wiki/Unified_Modeling_Language
- https://commons.wikimedia.org/wiki/Unified_Modeling_Language?uselang=de
- <http://www.omg.org/spec/UML/>

Für die UML-Diagrammgenerierung (ähnlich wie GraphViz <http://www.graphviz.org/>) gibt es das Tool PlantUML <http://plantuml.sourceforge.net/>. Es existieren genügend Programme, die eine Benutzeroberfläche für die UML-Diagrammerstellung behilflich sind.

Hier ein Beispiel von einem Klassendiagramm:



Die **C++** Repräsentation dieses Diagrammes sieht wie folgt aus:

```
// Klassendefinition
class Inneneinrichtung {
};

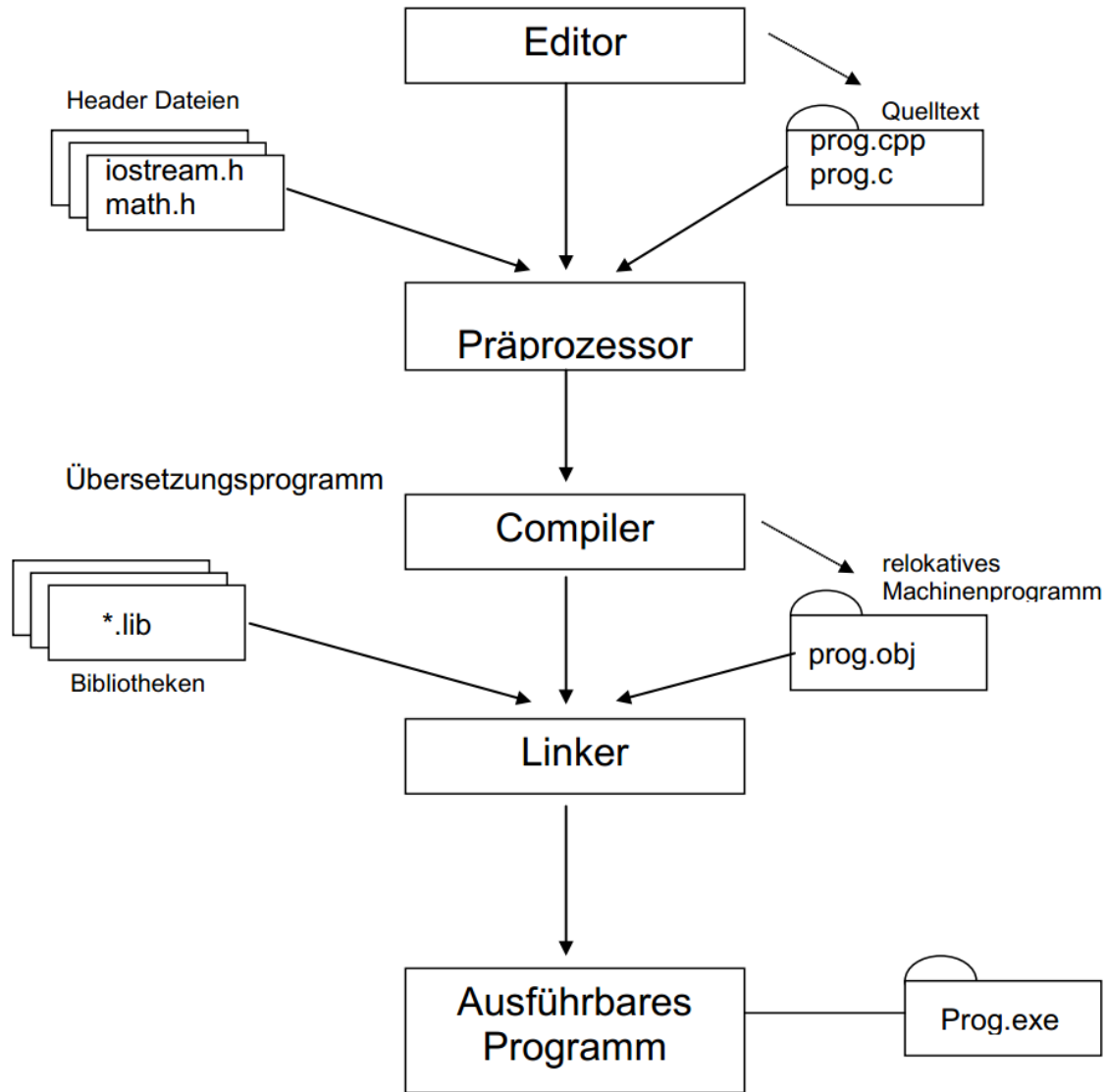
// Klassendefinition mit Ableitung
class Stuhl : public Inneneinrichtung {
};

class Tisch : public Inneneinrichtung {
};

class Sofa : public Inneneinrichtung {
};
```

2.5 Codierung

Im Folgenden wird erklärt, wie vom Quelltext im Editor ein ausführbares Programm entsteht.



- **Editor**: Im Editor (Entwicklungsumgebung) erstellt der Entwickler sein Programm.
- **Präprozessor**: Im Präprozessor werden Compiler spezifischen Sprachmerkmale und Makrodefinitionen ausgewertet/expandiert.
- **Compiler**: Hier wird der expandierte Code kompiliert.
- **Linker**: Nach dem Erstellen des Codes werden die Methoden miteinander verbunden.
- **Ausführbares Programm**: Am Ende (sofern in den Schritten 2, 3 und 4 keine Fehler aufgetreten sind) steht das ausführbare Programm zur Verfügung.

3 Einführung

C++ ist eine formatfreie Sprache. Das bedeutet, dass dem Aussehen des Quelltextes im weitesten Sinne keine Grenzen gesetzt sind. Dennoch sollte auf eine saubere, ordentliche und vor allem dokumentierte Programmstruktur geachtet werden, um das erstellte Programm selbst nach einigen Wochen noch verstehen zu können und um den möglichen Teammitgliedern das Einlesen in den Fremdensegmentcode zu erleichtern.

Während **C++** im äußeren Aufbau sehr großzügig ist, muss auf die Groß-Klein-Schreibung von Funktionen, Klassennamen, Methoden und Variablennamen geachtet werden.

Ein allgemeiner Aufbau einer **C++** Datei kann wie folgt aussehen:

```
[Präprozessor-Direktiven]
[Header-Dateien (Includes)]
[Klassen/Funktionen/Konstanten-Definitionen]
[Präprozessor-Direktiven]
```

Ein mehrzeiliges Kommentar wird mit den Zeichen `/* ... */` markiert. Ein einzeiliges Kommentar wird mit `//` markiert.

```
// Lorem ipsum

/*
    Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
    sed diam nonumy eirmod tempor invidunt ut labore et
    dolore magna aliquyam erat, sed diam voluptua.
    At vero eos et accusam et justo duo dolores et ea rebum.
*/
```

Das beliebte Hello World Programm wird vermutlich die meist programmierte Applikation der Welt sein. Hier die **C++** Version:

```
// ** Header Dateien
// Benötigt um auf die Eingaben und Ausgaben auf einen Stream
// wie den cout (Konsolenausgabe) zu lenken.
#include <iostream>

// ** Einstiegspunkt in die Applikation
// argc = Anzahl der Argumente mit der die Applikation aufgerufen wird
//      Der Programmname selbst, wird auch mit als Argument geliefert,
//      somit ist dieser Wert immer mindestens 1.
// argv = Pointer auf die übergebene Argumente (Datentyp: C-String)
int main(int argc, char* argv[]) {
    // ** Ausgabe 'Hello World!' und beende mit einem Zeilenumbruch (std::endl)
    std::cout << "Hello World!" << std::endl;

    // ** Beende die Applikation mit dem Rückgabewert 0
    return 0;
}
```

3.1 Header-Dateien

Header-Dateien, auch als Include-Dateien bezeichnet, enthalten Funktionsprototypendeklarationen für Bibliotheksfunktionen. Die Anweisung `#include` ist kein C++-Sprachbestandteil, sondern eine Anweisung, die der Präprozessor versteht. Dieser sucht im Quelltext nach Anweisungen, die mit einer Raute beginnen und fügt hier in unserem Fall die Headerdateien `iostream` ein.

Es gibt zwei Möglichkeiten, um die Präprozessordirektive `#include` zu schreiben.

- **`#include <header_file>`**: Hier wird die Header-Datei global in der Umgebung gesucht.
- **`#include "header_file"`**: Hier wird die Header-Datei lokal innerhalb den Projekteinstellungen gesucht.

Die STL (**S**tandard **l**ibrary) Fähigkeiten werden über die globale Einbindung `#include <header_file>` und die eigen entwickelten über die lokale Definition `#include "header_file"` eingebunden. Die Header-Dateien haben typischerweise die Dateiendung `h`. Dies ist jedoch von der jeweiligen Bibliothek abhängig.

3.2 Einstiegspunkt

```
// ** Einstiegspunkt in die Applikation
// argc = Anzahl der Argumente mit der die Applikation aufgerufen wird
//      Der Programmname selbst, wird auch mit als Argument geliefert,
//      somit ist dieser Wert immer mindestens 1.
// argv = Pointer auf die übergebene Argumente (Datentyp: C-String)
int main(int argc, char* argv[]) {
    return 0;
}
```

Die Funktion **main** muss in jedem ausführbaren Programm genau einmal vorkommen, da sie der Startpunkt des Programmes ist und somit die Hauptfunktion darstellt. Die Funktionsdeklaration der Funktion **main** kann auch noch wie folgt aussehen:

```
// ** Keine Übergabeparameter
int main()
// ** Mit Übergabeparameter
int main(int argc, char* argv[])
int main(int argc, _TCHAR* argv[])
```

Funktionen beginnen immer mit einer öffnenden geschweiften Klammer und enden mit einer schließenden geschweiften Klammer. Die geschweiften Klammern haben in **C++** die Aufgabe, einen Bereich (Scope) von Anweisungen zu bilden. Bereiche können innerhalb von Bereichen gebildet werden. Wird dies innerhalb einer Funktion/Methode angewendet, so sind die definierten Variablen nur innerhalb des Bereiches (oder den Kindbereichen) sichtbar.

```
int main(int argc, char* argv[]) {
    // Definiere und initialisiere Rückgabewert
    int returnCode = 0;
    {
        // returnCode ist in diesem Bereich sichtbar
        int test = -1;
    }
    {
        // returnCode ist in diesem Bereich sichtbar
        // Die Variable test hingegen nicht
    }
    return returnCode;
}
```

Wird folgendes Programm über die Konsole mit Parametern aufgerufen

Beispiel.exe a b c d e:

```
int main(int argc, char* argv[])
{
    // ** Ausgabe vorhandene Argumenten
    std::cout << "Anzahl Argumenten: " << argc << std::endl;

    // ** Ausgabe der übergebenen Argumenten
    for (int i = 0; i < argc; ++i) {
        std::cout << "Argument[" << i << "] = " << argv[ i ] << std::endl;
    }

    return 0;
}
```

Somit wird folgende Ausgabe erzeugt:

```
Anzahl Argumenten: 6
Argument[0] = C:\Entwicklung\Beispiel\Beispiel.exe
Argument[1] = a
Argument[2] = b
Argument[3] = c
Argument[4] = d
Argument[5] = e
```

3.3 Namensbereiche

In **C++** existieren Namensbereiche *namespace*. Die Namensbereiche dienen dazu Fähigkeiten zu strukturieren. Um Funktionen, Klassen o. Ä. von einem speziellen Namensbereich aufzurufen gibt es den *Operator ::*. Hier ein Beispiel mit *std::cout*:

```
// ** Ausgabe 'Hello World!' und beende mit einem Zeilenumbruch (std::endl)
std::cout << "Hello World!" << std::endl;
```

std ist der Namensbereich der STL Fähigkeiten und *cout* der Stream der aufgerufen wird. Es gibt die Möglichkeit, sich die Schreiarbeit zu vereinfachen, um auf die Elemente in den Namensbereichen zuzugreifen. Mit Hilfe der Anweisung *using namespace <Namensbereich>* können die Funktionen global (für den aktuellen Bereich) verfügbar gemacht werden.

```
// ** Header Dateien
// Benötigt um auf die Eingaben und Ausgaben auf einen Stream
// wie den cout (Konsolenausgabe) zu lenken.
#include <iostream>

// ** Namensbereich inkludieren
// Alle STD Klassen sind jetzt ohne std:: im aktuellen Namensbereich
// verfügbar.
using namespace std;

// ** Einstiegspunkt in die Applikation
// argc = Anzahl der Argumente mit der die Applikation aufgerufen wird
//      Der Programmname selbst, wird auch mit als Argument geliefert,
//      somit ist dieser Wert immer mindestens 1.
// argv = Pointer auf die übergebene Argumente (Datentyp: C-String)
int main(int argc, char* argv[]) {
    // ** Ausgabe 'Hello World!' und beende mit einem Zeilenumbruch (std::endl)
    cout << "Hello World!" << endl;

    // ** Beende die Applikation mit dem Rückgabewert 0
    return 0;
}
```

Es empfiehlt sich allerdings ausfolgenden Gründen auf die *using namespace* Anweisung zu verzichten:

- Dies erhöht die Lesbarkeit, wo sich der aufgerufene Code befindet. ?
 - Bei verschieben des Codes in eine andere Datei muss dieser nicht angepasst werden.
 - Namenskonflikten, z.B. bei zwei gleichnamige Klassen in unterschiedlichen Bereichen, wird vorgebeugt.
-



Die Namensbereiche können verschachtelt werden. Ein Beispiel:

```
namespace UserInterface {

    namespace Settings {

        int readSetting() {
            return 5;
        }

    }

}
```

Der Aufruf der Funktion *readSetting* erfolgt über folgenden Befehl:

```
UserInterface::Settings::readSetting();
```

3.4 Datentypen

In **C++** gibt es folgende atomare Datentypen:

Typ	Größe (Bytes)	Minimum	Maximum
bool	1	true, false	
(signed) char	1	-128	+127
unsigned char	1	0	+255
(signed) short int	2	-32768	+32767
unsigned short int	2	0	+65535
(signed) int	4	-2147483648	+2147483647
(signed) long	4		
unsigned int	4	0	+4294967295
unsigned long	4		
float	4	-3,402828E+38	3,402828E+38
double	8	-1,79769E+308	1,79769E+308
long double			

Diese Angaben gelten für den Visual Studio 2013 Compiler und der 32 bit Konfiguration. Für die Zeiger auf eine Speicheradresse, sogenannte Pointer, ist die Größe im 32 bit System 4 Byte lang und für das 64 bit System 8 Byte lang. Die atomaren Datentypen sind gleich groß.

Je nach Compiler können die Datentypen (wie double) eine andere Größe besitzen. Für weitere Informationen wird an dieser Stelle auf den **C++** Standard verwiesen.

Werden Datentypen mit einer bestimmten Größe benötigt, wie z.B. beim einlesen eines festgeschriebenen Dateiformats, dann kann auf Definition folgender Datentypen zurückgegriffen werden: *int8_t*, *uint8_t*, *int16_t*, *uint16_t* usw. siehe *stdint.h*.

Fließkommazahlen verwenden für die Nachkommastellen einen Punkt.

Im Code gibt es die Möglichkeit, sowohl Größe als auch Minimal- und Maximalwert eines Datentypes zu ermitteln.

Beispiele:

```
std::cout << "'double' hat die Größe "
          << sizeof( double ) << " Bytes"
          << std::endl;

std::cout << "Der Wertebereich von 'double' liegt bei: "
          << std::numeric_limits<double>::lowest()
          << " bis "
          << std::numeric_limits<double>::max() << std::endl;
```

Die Funktion *sizeof(T)* ermittelt die Größe eines Datentyps, der auch komplex sein kann. Für die Templatefunktion *numeric_limits< T >* wird die Headerdatei *limits* benötigt.

3.5 Variablen

Variablen werden nachfolgendem Schema deklariert:

[<Qualifier>] <Datentyp> <Bezeichnung>

Die Bezeichnung darf nicht mit einer Zahl anfangen. Bei den Bezeichner ist auf eine Groß- und Kleinschreibung zu achten.

Ein paar Beispiele:

```
// ** Deklaration von Variablen
// Fließkommazahl
float lengthFloat;
double lengthDouble;
// Ganzzahl
int numberOfEdges;
int numberOfTriangles;
int numberOfPoly;

// ** Wertzuweisung
// Für float wird hinter der Fließkommazahl ein f für float benötigt.
lengthFloat = 5.0f;
lengthDouble = 5.0;
// Wertzuweisung im Hexadezimalsystem (0x Prefix)
numberOfPoly = 0xFF;
// Mehrfachzuweisung
numberOfEdges = numberOfTriangles = 4;
```

```
// ** Deklaration von mehreren Variablen mit dem gleichen Typ
int numberOfObjects, numberOfDays;

// ** Deklaration und Initialisierung
double seconds = 0.0;
// Hier wird nur die Variable minutes initialisiert
double hours, minutes = 0.0;
```

Konstante können im weiteren Programmverlauf nicht verändert werden. Sie werden wie folgt definiert werden:

```
const int numberOfWeekdays = 7;
```

Konstante Ausdrücke können seit C++11 auch wie folgt markiert werden:

```
constexpr int sum(int a, int b) {
    return a + b;
}
```

Eine Variable sollte immer bei der Deklaration initialisiert werden. Wird die Variable nicht initialisiert, so wird der aktuelle Speicherinhalt als Wert verwendet. Dies kann zu einem unerwünschten Programmverhalten führen. Eine Mehrfachdeklaration sollte möglichst umgangen werden, um die Lesbarkeit im Code zu erhöhen.



3.6 Berechnungen

Für die Berechnungen stehen folgende Operatoren zur Verfügung:

- **Multiplikation:** *
- **Division:** /
- **Addition:** +
- **Subtraktion:** –
- **Modulo:** %

Bei den Berechnungen ist darauf zu achten, dass die Datentypen identisch sind oder miteinander kompatibel. Je nach CompilerEinstellungen kann eine indirekte Umwandlung (= cast) – ohne den Entwickler zu benachrichtigen – erfolgen und das gewünschte Ergebnis verfälschen.

```
double a = 1.0;
double b = 3.0;

double sum = a + b; // Ergebnis: 4.0
double diff = b - a; // Ergebnis: 2.0
double mul = a * b; // Ergebnis: 3.0
double div = a / b; // Ergebnis: 1.0/3.0
int mod = 3 % 5; // Ergebnis: 3
```

Folgende Kurzformen sind ebenso möglich:

```
// ** identisch mit: a = a + 5.0
double a = 10.0;
a += 5.0;

// ** identisch mit: b = b - 5.0
double b = 10.0;
b -= 5.0;

// ** identisch mit: c = c * 5.0
double c = 10.0;
c *= 5.0;

// ** identisch mit: d = d / 5.0
double d = 10.0;
d /= 5.0;

// ** identisch mit: e = e % 5:
int e = 3;
e %= 5;

// ** identisch mit: f = f + 1
int f = 0;
++f;
// oder
f++;

// identisch mit: g = g - 1
int g = 0;
--g;
// oder
g--;
```

Es sollte immer die Variante `--i`/`++i` (pre-increment) bevorzugt werden, wenn der aktuelle Wert beim Ausführen der Codezeile keine Rolle spielt. Bei `i++`/`i--` (post-increment) wird der aktuelle Wert zurückgegeben und dann erst `+1`/`-1` gerechnet. Im Fall von `++i`/`--i` wird die Rechenoperation direkt ausgeführt.



Weitere mathematische Funktionen, wie die Wurzel `sqrt`, `cos`, `sin` usw., sind in der Header-Datei `math.h` zu finden. In dieser Header-Datei sind ebenfalls mathematische Konstanten zu finden.

```
#define M_E 2.71828182845904523536
#define M_LOG2E 1.44269504088896340736
#define M_LOG10E 0.434294481903251827651
#define M_LN2 0.693147180559945309417
#define M_LN10 2.30258509299404568402
#define M_PI 3.14159265358979323846
```

```
#define M_PI_2 1.57079632679489661923
#define M_PI_4 0.785398163397448309616
#define M_1_PI 0.318309886183790671538
#define M_2_PI 0.636619772367581343076
#define M_2_SQRTPI 1.12837916709551257390
#define M_SQRT2 1.41421356237309504880
#define M_SQRT1_2 0.707106781186547524401
```

Eine Konstante sollte nie mit `#define` definiert werden. Ein `#define` kann mit dem Präprozessormakro `#undef` und `#define` überschrieben werden!



Hier ein Beispiel:

```
// ** math.h
#define M_PI 3.14159265358979323846

// ** im Eigenen Quellcode (Überschreiben):
#undef M_PI
#define M_PI 1234

// ** Ausgabe 1234
std::cout << "PI = " << M_PI << std::endl;
```

Folgende Varianten sind zu bevorzugen:

```
// ** Definitionen in einer Klasse als statische Methode
class Constants {

    static const double PI() { return 3.14159265358979323846; }

}

// -- ODER --

// ** Definition einer Konstante als Variable
const double PI = 3.14159265358979323846;
```

3.7 Enumerationen

Mit Hilfe von Enumerationen lassen sich Literale selbst definieren. Somit erhalten konkrete Werte einen sprechenden Namen. Hier ein Beispiel mit den Wochentagen. Als Alternative wäre auch ein `int` möglich gewesen, allerdings ohne sprechende Namen.

```
enum class Wochentag {
    Montag = 7,      // Standardwert für das erste Element ist 0,
    Dienstag,       // dies kann mit = Wert überschrieben werden.
    Mittwoch,        // Alle folgende Werte werden mit +1 hinterlegt.
    Donnerstag,      // Dienstag = 8, Mittwoch = 9, Donnerstag = 10 ...
    Freitag,
    Samstag,
    Sonntag
};
```

Die Enumeration kann wie folgt angewendet werden:

```
Wochentag w0 = Wochentag(3);  
Wochentag w1 = Wochentag::Mittwoch;  
Wochentag w2 = w0 + w1;
```

Der vorgestellte Code hier lehnt sich an den C++11 Standard. Mit kleinen Modifikationen ist dieser auch für C++98 anwendbar.

```
enum Wochentag { /*... */ }  
Wochentag w1 = Mittwoch;
```

Der Nachteil der C++98 Definition ergibt sich durch folgendes Beispiel:

```
enum Color {  
    Red  
};  
enum State {  
    Red  
};  
State s = Red;
```

Bei der Zuweisung von $s = \text{Red}$ ist nicht klar, welcher Datentyp verwendet werden soll – *Color* oder *State*?

3.8 Aufgabe: Grundrechenarten

In einem kurzen Programm sollen die Grundrechenarten unter C++ getestet werden. Deklarieren Sie dazu die Variable *result* als *float*. Überprüfen Sie jeweils, welche Resultate folgende Berechnungen liefern:

1. $\text{result} = 3 + 2;$
2. $\text{result} = 3 - 4;$
3. $\text{result} = 3 * 4;$
4. $\text{result} = 5 / 2;$
5. $\text{result} = 5 \% 2;$
6. $\text{result} = 5. / 2.;$

Die Ausgabe erfolgt jeweils über den *std::cout*-Stream.

Beantworten Sie folgende Fragen:

1. Welche Rechenarten liegen bei 4., 5. und 6. vor?
2. Wie sind die unterschiedlichen Ergebnisse aus 4. und 6. zu deuten?
3. Wie werden Zahlen potenziert?
4. Wo liegt der Unterschied zwischen ++i und i++?

3.9 Funktionen und Methoden

- **Funktionen:** Funktionen werden innerhalb eines Namensbereiches definiert.
- **Methoden:** Methoden, sind Funktionen, die innerhalb einer Klasse definiert sind.

```
int main(int argc, char* argv[])
```

Rückgabotyp: `int` (Ganzzahl mit Vorzeichen)

Name: `main`

Argumentenliste:

Argument 1:

Datentyp: `int`

Bezeichnung: `argc` (Akronym für argument count)

Argument 2:

Datentyp: `char**`

Bezeichnung: `argv` (Akronym für argument vector)

Eine Funktion/Methode besteht aus folgenden Bestandteilen:

1. Einen Rückgabotyp
2. Bezeichnung
3. Argumentenliste (Datentyp + Bezeichnung)

Ist der Rückgabotyp `void`, so gibt diese Funktion oder Methode keinen Wert zurück. Die Arten des Rückgabetyps sind die Gleichen wie bei der Argumentenliste.

Die Bezeichnung kann freigewählt werden.

Eine Bezeichnung sollte immer sinnvoll gewählt werden, um den Code besser zu verstehen. Akronyme sollten, wenn diese nicht allgemein Bekannt sind, vermieden werden. Eine stupide Durchzählung von Funktionen/Variablenamen ist nicht zu empfehlen.



Es gibt unterschiedliche Varianten, die Argumente an eine Funktion oder Methode zu definieren. Eine Funktion oder Methode kann keine Argumente besitzen.

- **arguments by value:** Wert wird kopiert.
- **arguments by reference:** Wert wird als Referenz übergeben.
- **arguments by address:** Pointer zu diesem Wert wird übergeben.

Wird der Argumenttyp mit `const` markiert, so kann die Implementierung der Funktion oder Methode den Wert des Arguments nicht ändern.

```
// ** value by value (a, b)
int add1( int a, int b ) {
    return a + b;
}
```

```
// ** value by reference (result)
void add2( int a, int b, int& result ) {
    result = a + b;
}

// ** value by address (result)
void add3( int a, int b, int* result ) {
    *result = a + b;
}

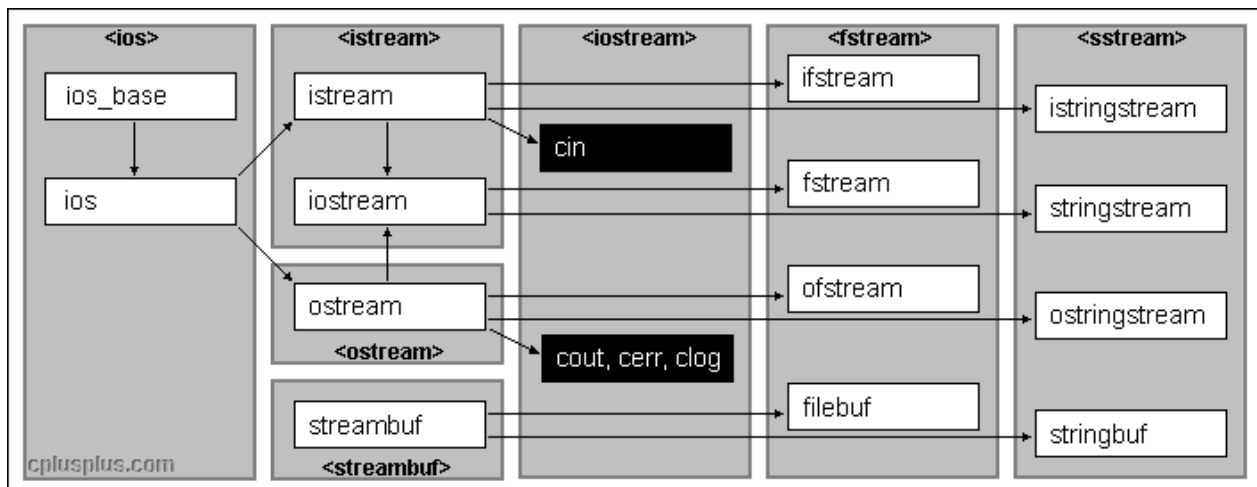
// ** const
void test( const int a ) {
    a = 4; // -> compiler error
}
```

Atomare Datentypen sollten immer, wenn dieser kein Rückgabewert ist, by value übergeben werden. Es ist schneller den Wert zu kopieren, als die Referenz auf die Variable aufzulösen.



3.10 Ein- und Ausgabe

In folgendem Diagramm von cplusplus.com <http://www.cplusplus.com/reference/iolibrary/> wird aufgezeigt, welche Streamtypen in der STL zur Verfügung stehen. o steht für output, i für input, f für file.



Mit Hilfe der Operatoren `<<` und `>>` kann von einem Stream gelesen oder auch geschrieben werden. Zusätzlich gibt es die Methoden `read` und `write` dafür.

Das Darstellungsformat von Fließkommazahlen kann bei den verfügbaren Streams eingestellt werden.

```
// Setzt Ausgabegenauigkeit auf 2 Kommastellen
std::cout.precision( 2 );

// Setzt das Ausgabeformat auf eine Festkommaausgabe
std::cout.setf( std::ios::fixed );

// Anzeige von Nullen vor bzw. hinter dem Komma
std::cout.setf( std::ios::showpoint );
```

Nähere Informationen zu diesen Funktionen sind u. a. auf der Homepage [cplusplus.com](http://www.cplusplus.com/reference/ios/ios/) <http://www.cplusplus.com/reference/ios/ios/> zu finden.

3.10.1 Konsole

Bereits am Beispiel des Hello World Programms war ersichtlich, dass eine Ausgabe an die Konsole mit Hilfe von `std::cout` und dem Operator `<<` erfolgt.

Mit Hilfe von `std::cin` und dem Operator `>>` können Eingaben von der Konsole gelesen werden.

Folgendes Beispiel zeigt eine Möglichkeit, um den Flächeninhalt eines Quadrats mit Eingabe vom Anwender zu bestimmen:

```
std::cout << "Seitenlänge in m: ";

double sideLength = 0.0;
std::cin >> sideLength;

double area = sideLength * sideLength;
std::cout << std::endl << "Der Flächeninhalt beträgt: " << area << "m²";
```

Es gibt die Möglichkeit mit Hilfe des Streams `std::cerr` Fehlermeldungen auf die Konsole zu schreiben. Wird eine Applikation von der Konsole aus gestartet, kann mit Hilfe von Pipelining `>` die Ausgabe z.B. in eine Datei umgelenkt werden. Mehr Informationen sind z.B. hier http://www.robvanderwoude.com/battech_redirection.php zu finden.



3.10.2 Datei

Mit Hilfe der STL-Klassen können auch Dateien gelesen und geschrieben werden. Es ist möglich, nicht nur Textdateien (ASCII), sondern auch binäre Dateiformate zu öffnen. Ein Dateistream kann zum Lesen, zum Schreiben oder zum Lesen und Schreiben geöffnet werden.

So könnte die Initialisierung der Dateistreams aussehen:

```
std::ifstream onlyRead( "test.in" );
std::ofstream onlyWrite( "test.out" );
std::fstream readWrite( "test.txt" );

std::fstream binaryReadWrite( "test.bin"
                             , std::ios_base::binary
```



```
| std::ios_base::in  
| std::ios_base::out );
```

Hier ein Beispiel um "Hello World" in eine Textdatei zu schreiben:

```
// ** Öffne Datei 'test.txt' zum schreiben  
std::ofstream onlyWrite( "test.txt" );  
  
// ** Prüfe ob die Datei geöffnet wurde  
if( !onlyWrite.is_open() ) {  
    // Im Fehlerfall, schreibe eine Fehlermeldung auf den Error-Stream  
    std::cerr << "test.txt konnte nicht geöffnet werden!" << std::endl;  
} else {  
    // Die Datei konnte erfolgreich geöffnet werden, schreibe den Dateiinhalt.  
    onlyWrite << "Hello World!" << std::endl;  
    // Schließe die Datei Ordnungsgemäß  
    // Wird die Variable 'onlyWrite' nach dem Verlassen des aktuellen Scope  
    // zerstört so wird der Dateistream automatisch geschlossen.  
    onlyWrite.close();  
}
```

3.11 Aufgabe: Variablen Initialisierung

Schreiben Sie ein Programm, welches mindestens 5 Variablen mit mindestens 3 unterschiedlichen Datentypen beinhaltet und diese in der Konsole ausgibt. Die Variablen sollen nur deklariert und nicht initialisiert werden. Beobachten Sie das Verhalten.

3.12 Aufgabe: Kreisberechnung

Schreiben Sie ein Programm zur Kreisberechnung. Nach der Eingabe des Radius soll der Kreisumfang sowie die Fläche mit entsprechenden Hinweisen am Bildschirm ausgegeben werden.

Folgende Formeln werden benötigt:

Kreisfläche:

$$A = \pi * r^2$$

Kreisumfang:

$$U = 2 * \pi * r$$

Konstante:

$$\pi = 3.1415926535897932$$

3.13 Aufgabe: Benzinverbrauch

Es soll ein Programm zur Berechnung des Benzinverbrauchs erstellt werden. Folgende Größen sind nach Bildschirmabfrage einzugeben:

- Kilometerstand beim vorletzten Tanken
- Kilometerstand beim letzten Tanken
- Getankte Spritmenge in Litern

Ausgegeben wird der errechnete Benzinverbrauch auf 100 km.

3.14 Aufgabe: Winkelfunktionen

In einem Programm sollen nach Eingabe des Winkels in Grad der Sinus, der Cosinus und der Tangens ausgegeben werden.

4 Kontrollstrukturen

Mit Hilfe von Kontrollstrukturen kann der Programmfluss verändert werden. Die Kontrollstrukturen können ineinander verschachtelt werden.

Ist die Anzahl der Folgeanweisung einer Kontrollstruktur 1, so kann auf die geschweifte Klammer verzichtet werden.

```
if( a == 0 )  
    std::cout << "a = 0!";
```

4.1 Bedingungen

Es stehen folgende Vergleichsoperatoren zur Verfügung (a, b und c können für absolute Werte stehen oder wiederum für eigene (verschachtelte) Bedingungen):

- a wird **negiert**: `!a` (true → false; false → true)
- a ist **gleich** b: `a == b`
- a ist **ungleich** b: `a != b`
- a ist **kleiner** als b: `a < b`
- a ist **größer** als b: `a > b`
- a ist **kleiner-gleich** b: `a <= b`
- a ist **größer-gleich** b: `a >= b`

Diese können logisch miteinander verbunden werden:

- a ist **gleich** b **und** b ist **größer** als c: `a == b && b > c`
- a ist **gleich** b **oder** b ist **größer** als c: `a == b || b > c`

Mit Hilfe von Klammerungen lassen sich Bedingungen schachteln.

- a ist **gleich** b **und** b ist **größer** als c **oder** die Bedingung ist erfüllt, wenn d **größer** 0 ist:
`(a == b && b > c) || d > 0`

4.2 Wenn-Dann (if-else)

Um Verzweigungen abhängig von einem zu bewertenden Ausdruck zu realisieren, wird in **C/C++** der Befehl *if* verwendet. Ist der Ausdruck falsch (*false*), wird hinter dem Block weitergearbeitet, ist er wahr (*true*), wird der folgende Block ausgeführt. **Jeder Wert ungleich 0 wird als wahr interpretiert!**

Eine Wurzelberechnung macht nur Sinn, wenn die Zahl größer gleich 0 ist. Daher hier ein Beispiel:

```
if( number >= 0.0 ) {  
    float result = sqrt( number );  
    std::cout << "Ergebnis lautet: " << result << std::endl;  
} else {  
    std::cerr << "Keine reelle Lösung vorhanden!" << std::endl;  
}
```

Der Dann/**Else**-Block einer if Anweisung ist **optional**. Es gibt auch die Möglichkeit *if-else-if-else...* Anweisungen zu bauen. Hier die Beispiele:

```
if( number >= 0.0 ) {  
    // mach was  
}  
  
// -----  
  
if( number < 10.0 ) {  
    // mach was  
} else if( number > 20.0 ) {  
    // mach was anderes  
} else {  
    // oh ganz toll  
}
```

Für die *if* Anweisung existiert eine Kurzschreibweise. Diese kann an einigen Stellen sinnvoll sein, es wird allerdings empfohlen, die ausgeschriebene Variante zu wählen, um die Lesbarkeit im Code nicht zu beeinträchtigen.



```
// ** Ausgeschrieben
{
    float number = 64.0f;
    float result = 0.0f;

    if( number >= 0 ) {
        result = sqrt( number );
    } else {
        result = 0.0f;
    }
}

// ** Kurzform
{
    float number = 64.0f;
    float result = number >= 0.0 ? sqrt( number ) : 0.0;
}
```

4.3 Aufgabe: Gasverbrauch

Gaszähler sind mit fünfstelligen Zählwerken ausgerüstet. Wenn der Gasverbrauch 99,999 m³ überschreitet, beginnt der Zähler wieder von 0 m³ an zu zählen.

Schreiben Sie ein Programm zur Berechnung des Gesamtgaspreises. Der alte abgelesene Zählerstand sowie der aktuelle Zählerstand bilden die Eingangsvariablen des Programms. Der Gaspreis soll im Programm mit einer Konstantendefinition zu 0,17 € pro m³ festgelegt werden. Berücksichtigen Sie im Programm auch den Fall, dass ein „Überlauf“ des Zählers stattfindet.

Schreiben Sie das Programm in **C++** und fertigen Sie ein Struktogramm an. Der Geldbetrag soll formatiert mit zwei Kommastellen ausgegeben werden (z.B. 46,48 €).

4.4 Behauptungen

Zur Compilezeit (wenn das Programm kompiliert wird) und zur Programmlaufzeit (wenn das Programm ausgeführt wird) gibt es die Möglichkeit Behauptungen aufzustellen. Behauptungen entsprechen den Annahmen die ein Entwickler annimmt in welchem Zustand zum Zeitpunkt X sein Programm sowohl zur Compilezeit respektive Programmlaufzeit hat.

Geht der Entwickler davon aus, dass ein Datentyp *Object* zum Compilezeit eine Größe von $2 * int$ sein muss, dass sein Programm wie erwartet funktioniert, so kann er dies wie folgt ausdrücken:

```
// ** Ohne eigene Fehlermeldung
static_assert( sizeof(Object) == sizeof(int)*2 );
// ** Mit eigene Fehlermeldung
static_assert( sizeof(Object) == sizeof(int)*2,
               "Invalid object size!");
```

Beim Kompilieren wird zwischen Debug und Release meist unterschieden. Je nach Projekt können noch zusätzliche weitere Konfigurationen für den Compiler bestehen. Hier die wesentlichen Unterschiede:

- **Debug:** Debugger zum Untersuchen des Programmablaufs ist ohne weiteres möglich. Es werden keine Optimierungen am Quellcode vorgenommen.
- **Release:** Optimierungen am Quellcode werden vorgenommen. Diese Konfiguration entspricht meist dem Ausliefergegenstand beim Kunden.

Die folgenden Behauptungen werden nur in der Debug Konfiguration ausgeführt. Dies bedeutet auch, dass auf keinem Fall Zuweisungen o.ä. die den Programmablauf durch deaktivieren der Behauptung verändern definiert werden dürfen.

Hier ein Beispiel:

```
int i = 0;
std::cin >> i;
// ** Ohne eigene Fehlermeldung
assert( i < 0 );
// ** Mit eigene Fehlermeldung
assert( i < 0 && "i is lower than 0!" );
```

4.5 Aufgabe: Parallelschaltung

Erstellen Sie ein Programm, welches den Gesamtwiderstand einer Parallelschaltung mit drei Widerständen berechnet. Es soll der Fall abgedeckt werden, dass ein Widerstand auch den Wert $0\ \Omega$ haben kann. Fangen Sie auch mögliche Fehleingaben ab. Die Fehleingaben sind mit Hilfe der Behauptungen abzufangen. Führen Sie Fehlerfälle in der Debug und Release Konfiguration aus!

Geben Sie den errechneten Wert in die Konsole aus, nach Eingabe der Widerstandswerte.

Die Formel für den Gesamtwiderstand für die vorliegende Parallelschaltung lautet:

$$\frac{1}{R_{ges}} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}$$

4.6 Aufgabe: Quadratische Gleichung

Die allgemeine Formel zur Lösung einer quadratischen Gleichung lautet:

$$a * x^2 + b * x + c = 0$$

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Erstellen Sie ein Programm, welches nach Tastatureingaben der Koeffizienten a, b und c die beiden Lösungen x1 und x2 für die Gleichung berechnet.

Berücksichtigen Sie auch die Sonderfälle $a = 0$ und einen möglichen negativen Wurzelausdruck.

- Wie lautet die Lösung für $a = 0$?
- Erstellen Sie ein Struktogramm.

4.7 Aufgabe: Tageszins

Nach der deutschen Zinsrechnung besteht der Zinsmonat aus 30 Tagen, d.h. das Jahr hat insgesamt $12 * 30 = 360$ Zinstage. Wenn Sie beispielsweise am 31.01. einen Betrag auf ihr Sparkonto einzahlen, werden die Zinsen ab dem 30.01. für den Zeitraum bis zum Jahresende berechnet. Der Jahreszinssatz sei mit 3 % konstant.

Schreiben Sie ein Programm zur Berechnung der Zinsen ab dem Einzahlungszeitpunkt eines Geldbetrages bis zum Jahresende.

Eingabedaten für das Programm sind

- der Geldbetrag
- der Einzahlungsmonat
- der Einzahlungstag

Die Zinsen errechnen sich wie folgt:

$$\text{Zinsen} = \frac{\text{Betrag} * \text{Zinstage} * \text{Zinssatz}}{3600}$$

Überlegen Sie zunächst, wie die effektiven Zinstage berechnet werden können. Verwenden Sie eine *if*-Abfrage, um den 31. des jeweiligen Monats abzudecken.

Zeichnen Sie ein Struktogramm und schreiben Sie das Programm!

4.8 Schalter (switch-case)

Die switch-case Anweisung ist eine weitere Kontrollstruktur. Jedes switch-case kann mit einer verschachtelten if-Anweisung ersetzt werden. Ein switch-case kann nur auf numerische ganzzahlige Werte angewendet werden. Enumerationen werden intern als numerische Ganzzahlen interpretiert.

```
int number = 0;
std::cout << "Geben Sie eine Nummer ein: ";
std::cin >> number;
std::cout << std::endl;

switch( number ) {
    case 0:
        std::cout << "Null";
        break;
    case 1:
        std::cout << "Eins";
        break;
    default:
        std::cout << "Fall für Nummer " << number << " nicht definiert.";
}
```

```
}
```

Wird in einem case-Block kein passender Wert gefunden, geht die Suche im nächsten case-Block weiter. Falls in keinem case-Block ein Wert gefunden wird, mit dem die Variable übereinstimmt, wird der default-Block ausgeführt. Der default-Block ist optional. Ein break kann auch gewollt weggelassen werden.

4.9 Aufgabe: Taschenrechner

In einem Programm werden zwei Zahlen sowie der Rechenoperator (+, -,) eingegeben. Je nach gewähltem Operator wird die entsprechende Rechenoperation ausgeführt. Die Eingabe eines ungültigen Operators soll zu einer Fehlermeldung führen.

Erstellen Sie ein Struktogramm und schreiben Sie das Programm!

4.10 Aufgabe: Mehrwertsteuer

Der Staat erhebt eine Mehrwertsteuer beim Verkauf von Waren. In der Regel beträgt sie 19% des Warenwertes. Für einige Warengruppen, wie z.B. Lebensmittel, werden nur 7% MwSt. erhoben. In einem Programm wird neben dem Nettopreis ein Buchstabe als Mehrwertsteuerkennzeichen eingegeben.

Lautet der Buchstabe „h“, sind 7% MwSt. zu berechnen. Wurde ein anderer Buchstabe eingegeben, so sind 19% MwSt. zu erheben. Ausgabedaten des Programmes sind die MwSt. und der Bruttopreis in Euro.

Erstellen Sie ein Struktogramm und schreiben Sie das Programm in **C++**.

4.11 Schleifen

Schleifen sind Kontrollstrukturen, die ein Set an Anweisungen wiederholen, bis die Bedingung nicht mehr gültig ist oder sie mit *break* oder *return* abgebrochen werden. Mit Hilfe des Schlüsselworts *continue* kann direkt in die nächste Wiederholung gesprungen werden.

Es ist darauf zu achten, dass keine Unendlich-Schleife bei Anwendungen entstehen! Ausnahmen bilden hierbei u.A. die Mikrocontroller Entwicklung.



4.11.1 Kopfgesteuerte-Schleife (while)

```
// ** Eingabe: Anzahl Wiederholungen
unsigned int numOfLoops = 0;
std::cout << "Anzahl Wiederholungen: ";
std::cin >> numOfLoops;

// ** Laufe Solange bis numOfLoops == 0 ist
while( numOfLoops > 0 ) {
    std::cout << numOfLoops << std::endl;
```



```
--numOfLoops;
}
```

Und die Unendlich-Schleifen-Variante:

```
while( true ) {
    // tbd
}
```

4.11.2 Fußgesteuerte-Schleife (do-while)

Die Fußgesteuerte-Schleife läuft mindestens einmal durch.

```
// ** Eingabe: Anzahl Replikate
int numOfReplications = 0;
std::cout << "Anzahl Replikate: ";
std::cin >> numOfReplications ;

// ** Laufe Solange bis numOfReplications == 0 ist
do {
    std::cout << numOfReplications << std::endl;
    --numOfReplications;
} while( numOfReplications >= 0 );
```

Und die Unendlich-Schleifen-Variante:

```
do {
    // tbd
} while( true );
```

4.11.3 Zählschleife (for)

Die Zählschleife läuft ebenfalls so lange, bis die Bedingung nicht mehr gültig ist.

```
// ** Ich zähle von 0 bis 9
for(unsigned int i=0; i < 10; ++i) {
    std::cout << i << std::endl;
}
```

Im ersten Bereich wird die Variable deklariert. Im zweiten Bereich wird die Bedingung formuliert. Im dritten Bereich wird festgelegt, wie sich die Variable nach einem Durchlauf verändern soll. Im dritten Bereich können auch mehrere Variablen verändert werden. Jeder Bereich kann leer gelassen werden.

```
int j = 100;
for(unsigned int k=0; k < 9; ++k, --j) {
    std::cout << k << " " << j << std::endl;
}
```

Und die Unendlich-Schleifen-Variante:

```
for(;;) {  
    // tbd  
}
```

Eine Unendlich-Schleife sollte immer mit *for* gebaut werden. Der Maschinencode, der bei einer *for* Schleife generiert wird, ist effizienter. Hierbei wird keine Bedingung mehr geprüft, wie bei der Implementierung mit einer *while(true)* Schleife. Der Wert *true* wird erst ausgewertet. Dieser Schritt entfällt bei *for(;;)*.

Die Verschachtelungstiefe von Kontrollstrukturen sollte möglichst flach gewählt werden.



5 Zeiger, Felder und Zeichenketten

5.1 Zeiger (Pointer)

Mit Hilfe eines Zeigers kann auf einen Speicherbereich gezeigt werden. Dies wird notwendig, wenn Datenstrukturen nicht mehr auf dem Stack (= lokaler Speicherbereich) allokiert werden können. Jeder Datentyp kann im Heap (= Haufen / Arbeitsspeicher) mit Hilfe des new Operator allokiert werden. Die Speicherfreigabe erfolgt über die delete Anweisung. Um hierbei einen Wert zuzuweisen, muss der Pointer dereferenziert werden. Dies geschieht mit Hilfe des * Operators. Hier ein Beispiel:

```
// Variable returnCode liegt auf dem Stack:
int main(int argc, char* argv[]) {
    int returnCode = 0;
    return returnCode;
}

// Variable returnCode liegt auf dem Heap:
int main(int argc, char* argv[]) {
    // Speicherreservierung im Heap
    int* returnCode = new int;
    // Mit Hilfe von * dereferenzieren für die Wertzuweisung:
    *returnCode = 0; // alternativ: returnCode[0] = 0;
    return *returnCode;
}
```

Von bestehenden (dereferenzierten) Objekten kann mit Hilfe von dem & Operator die Speicheradresse ermittelt werden.

```
// returnCode initialisieren
int returnCode = 0;

// Zeiger auf returnCode ermitteln
int* pReturnCode = &returnCode;
// Wertänderung
*pReturnCode = 3;

// Ausgabe von Wert 3
std::cout << returnCode << std::endl;
```

5.2 Felder (Array)

Mit Hilfe von Feldern kann ein zusammenhängender Speicher für den gleichen Datentyp reserviert werden.

5.2.1 Statische Felder

Ist zur Kompilierungszeit bereits klar, wie groß das Array ist, dann können statische Felder definiert werden. Die Felder können für jeden Datentyp definiert werden. Es ist zusätzlich möglich,

mehrdimensionale Felder zu definieren. Das erste Element in einem Feld wird mit dem Index 0 angesprochen. Daraus folgt, dass die Indizes einen Wertebereich von 0 bis Größe – 1 besitzen.

```
// ** 3D Vektor
double vec3a[3];
vec[0] = 0.0; // x
vec[1] = 1.0; // y
vec[2] = 2.0; // z
// oder
double vec3b[3] = { 0.0, 1.0, 2.0 };

// ** 3D Matrix
double mat3a[3][3];
for(unsigned int x=0; x < 3; ++x) {
    for(unsigned int y=0; y < 3; ++y) {
        mat3a[x][y] = 0.0;
    }
}
// oder
double mat3b[3][3] = { { 0.0, 0.0, 0.0 },
                      { 0.0, 0.0, 0.0 },
                      { 0.0, 0.0, 0.0 } };
```

5.2.2 Dynamische Felder

Wird die Größe eines Arrays erst zur Programmlaufzeit bestimmt, können dynamische Felder im Heap definiert werden. Mit Hilfe von `new` lässt sich ein Speicherblock reservieren und mit `delete[]` lässt sich ein Speicherblock freigeben. Die `[]` ist nur bei Feldern notwendig. Wird der Operator `[]` weggelassen, so wird nur das erste Feld freigegeben.

```
// ** 3D Vektor
unsigned int numOfElements = 3;

// Speicherreservierung
unsigned int* vec3a = new unsigned int[ numOfElements ];

// Wertzuweisung (Initialisierung)
vec3a[0] = 0.0;
vec3a[1] = 1.0;
vec3a[2] = 2.0;

// Speicherfreigabe
delete[] vec3a;

// ** 3D Matrix
// Speicherreservierung
unsigned int* mat3a = new unsigned int[ numOfElements * numOfElements ];

// Intialisierung
for(unsigned int x=0; x < numOfElements; ++x) {
```

```

    for(unsigned int y=0; y < numOfElements; ++y) {
        mat3a[ x + y * numOfElements ] = 0.0;
    }
}

// Speicherfreigabe
delete[] mat3a;

```

Die STL bietet bereits Container-Klassen an. Mit Hilfe dieser Container-Klassen lässt sich auch ein Vektor (=zusammenhängender Speicherblock) realisieren. Dadurch können Speicherlöcher, sogenannte memory leaks, vermieden werden. Nähere Informationen zur vector Klasse sind hier (<http://www.cplusplus.com/reference/vector/vector/>) zu finden.

```

// ** 3D Vektor (Header-Datei: vector)
// Anstelle von size_type kann auch andere Ganzzahl-Datentypen verwendet
werden.
// Dann wird u.U. ein impliziter cast erfolgen -> Wertebereiche beachten!
std::vector<double>::size_type numElements = 3;
// Instantiiere den Vektor mit Anzahl von numElements
// und setze die Werte auf 0.0
std::vector<double> vec3a( numElements, 0.0 );
// - oder -
std::vector<double> vec3a( numElements );
// - oder -
std::vector<double> vec3a;
vec3a.resize( numElements );

// ** 3D Matrix
std::vector< std::vector< double > > mat3a( 3, std::vector< double >( 3, 0.0 )
);

```

Es sollte, wenn möglich, auf die manuelle Speicherreservierung/-freigabe verzichtet werden. Dies führt u. U. zu Abstürzen bei Zugriff auf einen nicht gültigen Speicherbereich. Der allokierte Speicher wird nach Beenden der Applikation automatisch vom Betriebssystem freigegeben.



5.3 Aufgabe: Histogramm

Erstellen Sie ein Programm, welches auf Basis folgender 14 Datenpunkte ein Histogramm in der Konsole ausgibt. Die Kategorien des Histogramms sind: 1, 2, 3, 4, 5 und 6.

Klausur	Note
Mathe I	2
Mathe II	1

Physik I	3
Physik II	2
Informatik I	1
Informatik II	2
Biologie	4
Chemie	4
Deutsch	2
Englisch I	3
Englisch II	3
Hauswirtschaft	1
Kunst	6
Sport	5

5.4 Zeichenketten

Byte:	0	1	2	3	4	5	6	7	8	9	10	11
Wert:	H	e	l	l	o		W	o	r	l	d	\0

Die **C++** Zeichenkette entspricht einem Array vom Typ `char`. Eine **C++** Zeichenkette endet mit dem Kontrollzeichen `\0`. Somit ist die Größe des Arrays = Länge der Zeichenkette + 1.

```
const char* txt = "Hello World";
```

Die STL kapselt dieses Datenfeld und bietet Komfortfunktionen. Weitere Informationen zu der `string` Klasse ist hier (<http://www.cplusplus.com/reference/string/string/>) zu finden.

```
// Verwendung von std::string (Header-Datei: string)
std::string txt = "Hello World";
// char pointer
const char* pTxt = txt.c_str();

// Zeichenketten verbinden
std::string a = "Hello";
std::string b = "World";
std::string res = a + b;
```

Bei Verwendung von Zeichenketten sollte immer eine string Klasse verwendet werden, welche Komfortfunktionen bietet. Ein manuelles Bearbeiten (der Speicherblöcke) von Zeichenketten empfiehlt sich nicht.



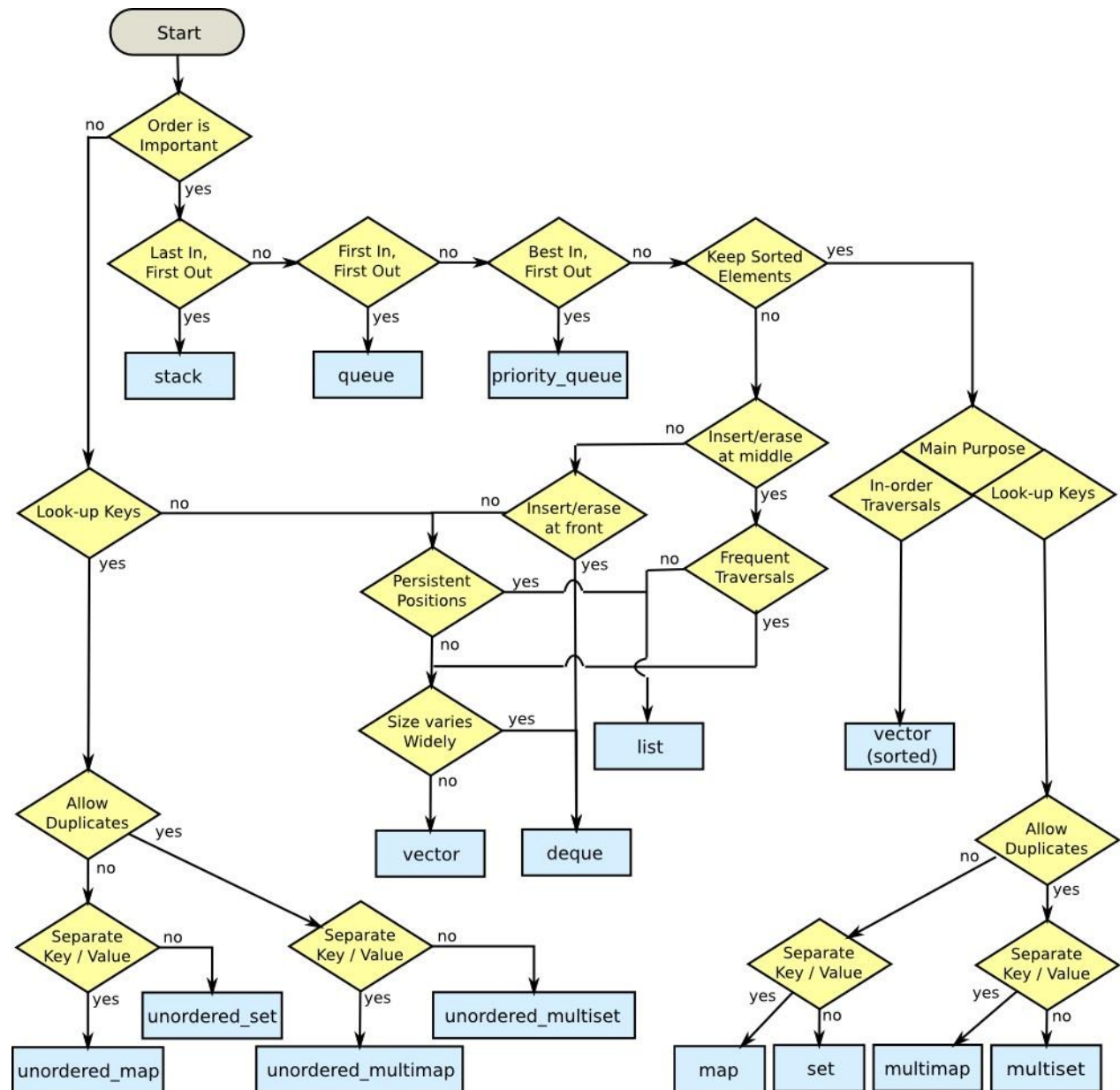
5.5 Aufgabe: Palindrom

Erstellen Sie ein Programm, welches ein Wort überprüft ob es sich um ein Palindrom handelt. Das Wort soll der Anwender über die Konsole eingeben können.

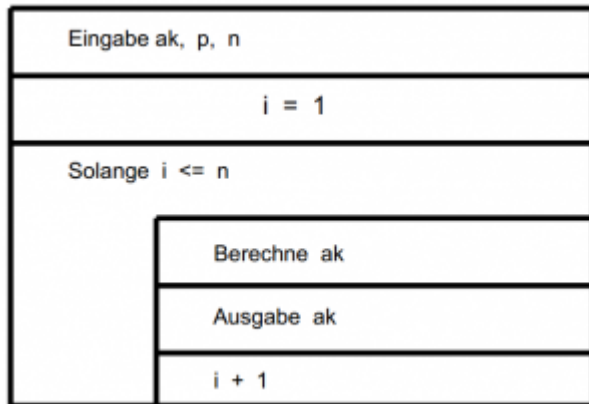
Die Erkennung soll sich auf eine einfache Überprüfung beschränken. Dies bedeutet, ein Wort ist ein Wortpalindrom, wenn das Wort Rückwärtsgelesen genau dasselbe Wort ergibt (z.B. Reittier).

5.6 Weitere Datenstrukturen

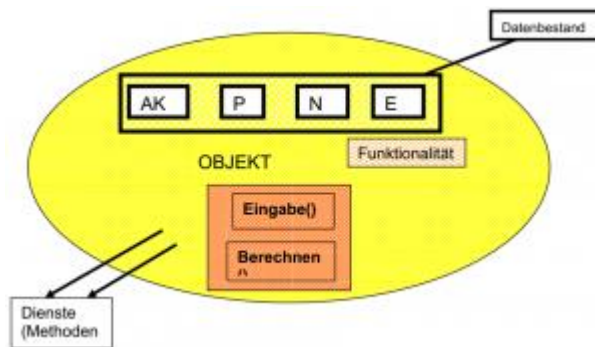
Jede Datenstruktur (statische, dynamische Felder...) hat sein eigenes Speicherlayout. Jedes dieser Layouts haben Vor- und Nachteile. Daher ist je nach Anwendungsfall zu unterscheiden welches Layout am geeignetstem ist. Dazu kann folgender Entscheidungsbaum (<http://i.stack.imgur.com/G70oT.png>) für die STL Datenstrukturen helfen:



6 Objektorientiertes Programmieren



Bisher stand der Programmfluss im Vordergrund. Dies nennt sich strukturiertes Programmieren.



Objektorientierung bedeutet Funktionen thematisch – und sinnvoll – in eine Klasse (= Konstruktionsplan) zu gruppieren. Mit Hilfe von Objektorientierung kann eine Hierarchie der Klassen abgebildet werden (Vererbung). Eine Klasse besteht aus Daten (= Datentypen) und Vorschriften (= Methoden), wie die Daten zu verändern sind. Ein Objekt ist eine Instanz einer Klasse.

Änderung eines Objektzustands erfolgt über Methodenaufrufe. Der direkte Zugriff von außen auf Datenelemente eines Objekts ist größtenteils nicht notwendig und in der Regel auch nicht erlaubt. Bestimmte Informationen sind daher nur lokal. Sie verteilen sich nicht über ein ganzes Softwaresystem, welches somit veränderbar und übersichtlich bleibt.

6.1 Klassen definieren

Eine Klasse wird mit dem Schlüsselwort `class` definiert. Ein `struct` ist technisch gesehen das Selbe. In folgendem Beispiel wird eine 3D Vektorklasse beschrieben, die unterschiedliche Methoden zur Bearbeitung des Inhaltes aufweist. Typischerweise steht in der Header-Datei die Klassendefinition und in der **C++** Datei die Implementierung der definierten Methoden. Der Methodenzusatz `const` gibt das Versprechen, dass mit dieser Methode keine Veränderung des Objektzustandes herbeigeführt wird. In einer Klasse gibt es drei unterschiedliche Sichtbarkeiten:

- **private:** Alle Methoden/Variablen sind nur innerhalb der Klasse diesen Typs sichtbar.
- **protected:** Alle Methoden/Variablen sind von abgeleiteten Klassen sichtbar.
- **public:** Alle Methoden/Variablen sind für alle sichtbar.

```
// **
// Header-Datei
// **
class Vector3 {
public:
    // Konstruktor - Aufruf bei der Instantiierung eines Objekts
    Vector3( double x, double y, double z );

    // Destruktor - Aufruf, wenn das Objekt gelöscht wird
    virtual ~Vector3();

    // Setter
    void set( double x, double y, double z );

    // Getter
    double getX() const;

    double getY() const;

    double getZ() const;

private:
    // Variablendefinition
    double m_x;
    double m_y;
    double m_z;
}

// **
// CPP-Datei
// **
Vector3::Vector3( double x, double y, double z )
// Konstruktorliste
: m_x( x )
, m_y( y )
, m_z( z )
{
}

Vector3::~~Vector3() {
}

void Vector3::set( double x, double y, double z ) {
    // Werte anpassen
```

```
m_x = x;
m_y = y;
m_z = z;
}

double Vector3::getX() const {
    return m_x;
}

double Vector3::getY() const {
    return m_y;
}

double Vector3::getZ() const {
    return m_z;
}

// **
// Verwendung
// **
// Objekt instantiieren
Vector3 a( 1.0, 2.0, 3.0 );
// Methode set Aufrufen
a.set( 4.0, 3.0, 2.0 );
// Ausgabe des Y Werts -> 3.0
std::cout << a.getY();
```

- Es empfiehlt sich alle Klassenvariablen mit einem Prefix zu versehen. Im angeführten Beispiel ist die Klassenvariable mit dem Prefix m_ (für Member) versehen.
- Für einfache Setter und Getter sollten die Methoden inline implementiert werden, um eine schnellere Abarbeitung zu ermöglichen.



```
class Vector2 {
public:
    // Konstruktor - Aufruf bei der Instanziierung eines
    Objekts
    Vector2 ( double x, double y );

    // Destruktor - Aufruf, wenn das Objekt gelöscht wird
    virtual ~Vector2 ();

    // Setter
    void set( double x, double y );

    // Getter
    double getX() const { return m_x; }

    double getY() const { return m_y; }

private:
    double m_x;
    double m_y;
}
```

Hier nochmal zum Vergleich die prozedurale und objektorientierte Variante:

```
// ** Prozedural
double a[2] = { 0.0, 0.0 };
double b[2] = { 0.0, 0.0 };

// ** Objektorientiert
Vector2 a( 0.0, 0.0 );
Vector2 b( 0.0, 0.0 );
```

Werden jetzt vektorspezifische Funktionen benötigt, so muss in der prozeduralen Variante jeweils eine Funktion implementiert werden. In der objektorientierten Variante kann die Klasse um die jeweiligen Fähigkeiten erweitert werden. Somit steht diese überall im Programm zur Verfügung, auch in den abgeleiteten Klassen.

```
// ** Prozedural
double vec2_length(const double* vec2) {
    return sqrt( vec2[0] * vec2[0] + vec2[1] * vec2[1] );
}

// ** Objektorientiert
class Vector2 {
    // siehe Code-Ausschnitt oben
```

```

double length() const {
    return sqrt( m_x * m_x + m_y * m_y );
}

// ** Aufrufe
// -- Prozedural
double a[2] = { 0.0, 0.0 }
double a_length = vec2_length( &a );

// -- Objektorientiert
Vector2 a( 0.0, 0.0 );
double a_length = a.length();

```

6.2 Vererbung

Mit Hilfe von Vererbung lassen sich Eigenschaften/Methoden in eine Ableitungshierarchie abbilden. Virtuelle Methoden können von abgeleiteten Klassen überschrieben werden. Ist die virtuelle Methode pure-virtual, dann müssen die abgeleiteten Klassen diese implementieren – außer sie selbst ist wieder ein Interface. Ist eine Methode virtuell, bleibt sie in den abgeleiteten Klassen virtuell – auch wenn das Schlüsselwort virtual nicht mehr geschrieben wird. Die Sichtbarkeiten der Super-Klassen können auch mit public, protected und private geändert werden.

```

// Interface Definition für ein Lebewesen
class LivingEntity {
public:
    // pure-virtual:
    //   Interface Methode, jede abgeleitete Klasse muss diese implementieren
    virtual bool isHungry() const = 0;
};

// Implementierung eines Menschen
class Human : public LivingEntity {
public:
    // Überschreiben der Interface Methode
    virtual bool isHungry();

    // Setter/Getter für Anschrift
    void setAddress(const std::string& address);
    const std::string getAddress() const;
};

// Implementierung eines Tiers
class Animal : public LivingEntity {
public:
    // pure-virtual:
    //   Eine weitere Interface Methode, jede abgeleitete Klasse muss diese
    //   implementieren
    virtual unsigned int getNumberOfPaws() const = 0;
};

```

```
};

class Dog : public Animal {
public:
    virtual bool isHungry() const;
    virtual unsigned int getNumberOfPaws() const;
};

class Cat : public Animal {
public:
    virtual bool isHungry() const;
    virtual unsigned int getNumberOfPaws() const;
};
```

Wird eine Interface-Klasse verwendet, so ist es notwendig, dass das Objekt mit Hilfe des new Operators erzeugt wird. Dies ist notwendig, damit die vtable für die virtuellen Methoden korrekt initialisiert wird.

```
// Instanziierung des Objekts vom Typ Cat
Animal* animal = new Cat();

// Mit Hilfe des Pfeil-Operators, die Methode isHungry aufrufen
if( animal->isHungry() ) {
    // Ausgabe
    std::cout << "Ah, das Tier ist hungrig!" << std::endl;
}

// Ausgabe Anzahl Pfoten
std::cout << "Das Tier hat "
           << animal->getNumberOfPaws()
           << " Pfoten." << std::endl;

// Objekt löschen
delete animal;
```

C++ unterstützt Mehrfachvererbung. Dies bedeutet, dass eine Klasse von N verschiedenen Klassen gleichzeitig erben kann.

```
// * Definiert ein Interface, um ein Objekt in eine Datei zu serialisieren.
class Serializable {
public:
    virtual void writeTo(const std::string& fn) = 0;

    virtual void readFrom(const std::string& fn) = 0;
};

// * Beschreibt ein allgemeines Objekt in der Applikation.
class Object {
public:
```

```

void setId(unsigned int id) { m_id = id; }

unsigned int getId() const { return m_id; }

private:
    unsigned int m_id;
};

// * Ein Tier, welches ein Objekt darstellt
// * und zusätzlich noch serialisierbar ist.
class Animal : public Object, public Serializable {
    // tbd
};

```

Das manuelle Reservieren und Freigeben von Speicher sollte auch hier vermieden werden. Dafür gibt es die sogenannten smart pointer. Diese werden im Rahmen dieser Vorlesung nicht näher beleuchtet. Die verfügbaren smart pointer der STL sind hier (<http://www.cplusplus.com/reference/memory/>) zu finden.



6.3 Aufgabe: Rechtschreibung

Schreiben Sie ein Programm, dass die folgende These bestätigt!

„An einer englischen Universität wurde herausgefunden, dass man beim Lesen eines Textes nur auf das gesamte Wort achtet und nicht auf die Anordnung der einzelnen Buchstaben. Entscheidend sind lediglich der erste und der letzte Buchstabe des Wortes...“

Dabei müssen Sie beachten:

- Die Ein- und Ausgabe soll mit Hilfe von .txt-Dateien ausgeführt werden.
- Programmieren Sie möglichst objektorientiert!
- Berücksichtigen Sie Sonderzeichen!

Hier ein kleines Beispiel eines solchen Textes:

Afugrnud enier Sduite an enier Elingshcen Unvirestiät ist es eagl, in wleher Rienhnelfoge die Bcuhtsbaen in eniem Wrot sethen, das enizg wcihitge dbaei ist, dsas der estre und lzete Bcuhtsbae am rcihgiten Paltz snid. Der Rset knan ttolaer Bölsdinn sien, und Sie kenönn es torztedm onhe Porbelme lseen. Das ghet dseahlb, wiel wir nchit Bcuhtsbae für Bcuhtsbae enizlen lseen, snodren Wröetr als Gnaezs. Smtimt's?

6.4 Aufgabe: Würfel

Erstellen Sie einen Zufallsgenerator, der es ermöglicht für den Datentyp int eine Zufallszahl zwischen einem vom Benutzer gewählten Minimum und Maximum zu ziehen.

Die generierten Zufallszahlen sind keine echten Zufallszahlen, da sie durch einen Computer generiert werden. Mathematisch korrekt werden Sie daher als Pseudozufallszahlen bezeichnet.

Der Befehl `srand ((unsigned) time(NULL));` initialisiert eine neue Zufallszahlenfolge abhängig vom Typ und der Zeit seit 1970. Bei der Verwendung von `time` wird nur ca. jede Sekunde eine neue Zufallszahl erzeugt, daher wird im praktischen Gebrauch eher `srand ((unsigned) clock());` verwendet. Bei der Initialisierung mit `clock()` wird die Summe der bisher durchgeführten Prozesse zur Ermittlung der Zufallszahl mit einbezogen.

Um die Reihenfolge der gezogenen Zufallszahlen wiederherstellen zu können, muss der Initialisierungswert (= seed) des Zufalls gespeichert werden und der Programmablauf darf sich nicht verändern.

- Entwerfen Sie eine Klasse, die die Fähigkeit eines Zufallszahlengenerators kapselt.
- Es soll folgender Datentyp unterstützt werden: `int`
- Der Anwender soll das Minimum und das Maximum festlegen können
- Der Anwender soll die Zufallszahl auf der Konsole dargestellt bekommen
- Ungültige Eingaben sollen abgefangen werden

7 Vorlagen (Templates)

Mit Hilfe von Templates lassen sich für jeden beliebigen Datentyp Funktion-/Methoden-/Klassenvorlagen erstellen. Angenommen ein 2D Punkt soll in einem Ganzzahlensystem und in einem Fließkommasystem beschrieben werden, so können diese zwei Spezialisierungen ausprogrammiert werden.

```
// * 2D Punkt für ein Ganzzahlensystem
class IntPoint2D {
public:
    IntPoint2D(int x, int y) : m_x( x ), m_y( y ) {}
    virtual ~IntPoint2D() {}

    int getX() const { return m_x; }
    int getY() const { return m_y; }
private:
    int m_x;
    int m_y;
};

// * 2D Punkt für ein Fließkommasystem
class FloatPoint2D {
public:
    FloatPoint2D(float x, float y) : m_x( x ), m_y( y ) {}
    virtual ~FloatPoint2D() {}

    float getX() const { return m_x; }
    float getY() const { return m_y; }
private:
    float m_x;
    float m_y;
};
```

Dies lässt sich mit Hilfe von Templates wie folgt lösen:

```
// * Definition Templateklasse für ein 2D Punkt.
template< class T >
class Point2D {
public:
    Point2D(T x, T y) : m_x( x ), m_y( y ) {}
    virtual ~Point2D() {}

    T getX() const { return m_x; }
    T getY() const { return m_y; }
private:
    T m_x;
    T m_y;
};

// * Anwendungsbeispiele
```

```
Point2D<int> intP2D( 3, 5 );
Point2D<float> floatP2D( 2f, 6f );
Point2D<double> double2D( 2.4, 4.5 );
```

Die Anzahl der Templateargumente ist nicht beschränkt. Es gibt die Möglichkeit, Templateargumente Standardwerte zuzuweisen.

```
// * Definition Templateklasse für ein 2D Punkt.
// * Standard für den Datentyp float.
template< class T = float >
class Point2D {
public:
    Point2D(T x, T y) : m_x( x ), m_y( y ) {}
    virtual ~Point2D() {}

    T getX() const { return m_x; }
    T getY() const { return m_y; }
private:
    T m_x;
    T m_y;
};

// * Anwendungsbeispiele
Point2D floatP2D( 2f, 6f );
```

8 Umwandeln (Casts)

Mit Hilfe eines Casts kann ein Typ A auf einen Typ B umgewandelt werden. Die in **C++** zur Verfügung stehenden Casts sind im Folgenden aufgelistet.

8.1 static_cast

Mit Hilfe des static_cast kann hart von einem Typ A auf ein Typ B gecastet werden.

```
// * Ausgeschriebene Form
double a = 3.5;
int b = static_cast<int>( a ); // Ergebnis: 3
double c = static_cast<double>( b ); // Ergebnis: 3.0

// * Kurzform(en)
double a = 3.5;
int b = int( a ); // Ergebnis: 3
double c = (double)b; // Ergebnis: 3.0
```

8.2 dynamic_cast

Mit Hilfe des dynamic_cast kann geprüft werden, ob ein Objekt von Typ A auch von Typ B ist.

```
// * Eine einfache Klassenhierarchie (A <- B <- C)
class A {
};

class B : public A {
};

class C : public B {
};

// * Erstellung der zu testenden Objekte
A* b = new B();
A* c = new C();

// * Wandle Objekt c nach Typ B um.
B* bc = dynamic_cast<B*>( c ); // Ergebnis: gültiger Pointer

// * Wandle Objekt b nach Typ C um.
C* cb = dynamic_cast<C*>( b ); // Ergebnis: nullptr
```

8.3 const_cast

Wird ein Parameter als const übergeben, dann ist das ein Versprechen, dass die aufzurufende Funktion an dem übergebenen Objekt nichts verändert. Versprechen können jedoch nicht gehalten werden. In seltenen Fällen ist es notwendig, einen const Zeiger in einen normalen Zeiger zu wandeln.

```

class A {
public:
    A(int id) : m_id(id) {}

    void setId(int id) { m_id = id; }

    int getId() const { return m_id; }
private:
    int m_id;
};

// * Erzeuge Objekt mit einer Id = 5
const A* a = new A( 5 );
// * Ausgabe der Id -> 5
std::cout << a->getId() << std::endl;

// * Folgender Aufruf würde zu einem Compilerfehler führen.
// a->setId( 123 );

// * Umwandlung in ein normalen Pointer
A* nonConstA = const_cast< A* >( a );
nonConstA->setId( 123 );

// * Ausgabe der Id -> 123
std::cout << a->getId() << std::endl;

```

8.4 reinterpret_cast

Mit Hilfe des `reinterpret_cast` wird der Datenbereich auf Bitebene als gewünschter Typ interpretiert.

```

// * Initialisiere Datenbereich
char data[4] = { 0xFF, 0xFF, 0xFF, 0xFF };

// * Interpretiere den Datenbereich als ein unsigned int
// -> std::numeric_limits<unsigned int>::max() = 4294967295
unsigned int* v = reinterpret_cast<unsigned int*>(&data[0]);

```

Eine Richtlinie besagt, sobald ein Cast in der Softwarearchitektur benötigt wird, ist die Softwarearchitektur von niedriger Qualität. Kommen in einem Programmablauf sehr viele Casts vom Typ `dynamic_cast` vor, sollte überlegt werden, ob die Fähigkeiten nicht stärker in Objekte und Ableitungshierarchien verschoben werden können.



9 Operatoren

Es besteht die Möglichkeit Operatoren zu überschreiben.

```
class Vector2D {
public:
    Vector2D(int x, int y)
        : m_x(x)
        , m_y(y)
    {}

    int getX() const { return m_x; }

    int getY() const { return m_y; }

    // + Operator überschreiben
    friend Vector2D operator+(const Vector2D& lhs, const Vector2D& rhs) {
        return Vector2D(lhs.m_x + rhs.m_x, lhs.m_y + rhs.m_y);
    }

private:
    int m_x;
    int m_y;
};

// * Anwendung
Vector2D a(2, 2);
Vector2D b(1, 3);
// Ergebnis: c = (3/5)
Vector2D c = a + b;
```

Das Überschreiben von Operatoren kann auf jeden beliebigen Operator angewendet werden. Eine ausführliche Beschreibung ist hier (<http://en.cppreference.com/w/cpp/language/operators>) zu finden.

10 (Fehler)quellen

In diesem Kapitel werden die möglichen Fehlerquellen während der Programmierung beleuchtet.

Es kann in bestimmten Fällen sinnvoll sein, die folgenden Verhalten in der Implementierung auszunutzen. Daher ist es wichtig zu wissen, welcher Code zu welchem Verhalten führt!



Ganzzahl-Division

```
double resultA = 5 / 2; // Ergebnis: 2
double resultB = 9 / 2; // Ergebnis: 4

// ** Mehrwertsteuerberechnung
double mwsA = 2 / 100 * 19; // Falsch! (Ergebnis: 0.0)
double mwsB = 2 / 100. * 19; // Richtig! (Ergebnis: 0.38)
```

Sind die Operanden Ganzzahlen, wird eine Ganzzahl-Division durchgeführt, d.h. die Kommastellen werden abgeschnitten.

Überlauf

```
char result = 200 + 500; // Ergebnis: -62
```

Wird für eine Berechnung der falsche Datentyp gewählt, kann es zu einem Überlauf kommen.

Abbruch nicht definiert

```
switch( day ) {
    // Mo
    case 0:
        std::cout << "Pommes" << std::endl;
    // Di
    case 1:
        std::cout << "Nudeln" << std::endl;
    // Mi
    default:
        std::cout << "Geschlossen" << std::endl;
}
```

Im angeführten Beispiel wurde in jedem case Block das break vergessen. Somit erscheint als Ausgabe des aktuellen Menüplans, dass Pommes und Nudeln auf der Karte stehen und zudem, dass die Kantine geschlossen hat.

11 Aufgabenpool

11.1 Aufgabe: Fibonacci Folge

Die Fibonacci-Folge ist die unendliche Folge von natürlichen Zahlen, die (ursprünglich) mit zweimal der Zahl 1 beginnt oder (häufig, in moderner Schreibweise) zusätzlich mit einer führenden Zahl 0 versehen ist. Im Anschluss ergibt jeweils die Summe zweier aufeinanderfolgender Zahlen die unmittelbar danach folgende Zahl:

$$0, 1, 1, 2, 3, 5, 8, 13, \dots$$

(optional) $0+1$ " $1+1$ " $1+2$ " $2+3$ " $3+5$ " $5+8$ "

Die Fibonacci-Folge f_1, f_2, f_3, \dots ist durch das rekursive Bildungsgesetz

$$f_n = f_{n-1} + f_{n-2} \text{ für } n > 2$$

mit den Anfangswerten

$$f_1 = f_2 = 1$$

definiert. Das bedeutet in Worten:

Für die beiden ersten Zahlen wird der Wert *eins* vorgegeben.

Jede weitere Zahl ist die Summe ihrer beiden Vorgänger in der Folge.

Daraus ergibt sich:

$f_1 = 1$	$f_6 = 8$	$f_{11} = 89$
$f_2 = 1$	$f_7 = 13$	$f_{12} = 144$
$f_3 = 2$	$f_8 = 21$	$f_{13} = 233$
$f_4 = 3$	$f_9 = 34$...
$f_5 = 5$	$f_{10} = 55$	$f_{20} = 6765$

Anforderungen:

Grundgerüst

- (1) Implementieren Sie eine Funktion zur rekursiven Berechnung der Fibonacci Folge.
- (2) Der Anwender soll den Wert von n über die Konsole bestimmen können.

Komfort

- (3) Es sollen ungültige Benutzereingaben abgefangen werden.

Herausfordernd

- (4) Lösen Sie die rekursive Implementierung auf.

11.2 Aufgabe: Datenkapselung

Implementieren Sie eine Klasse `Speed` die folgende Anforderungen erfüllt.

- Interne Datenhaltung erfolgt in m/s.
- Eine implizite Konvertierung von `double` zum Datentyp `Speed` muss unterbunden werden.
- Die Klasse besitzt die Fähigkeit die Geschwindigkeit in folgenden Maßeinheiten zurückzugeben: km/h, mph
- Es soll möglich sein bei vorhanden m/s, km/h oder mph Werten direkt eine Objektinstanz zu erzeugen.
- Überladen Sie alle sinnvollen Operatoren.
- Testen Sie die Implementierung mit Hilfe von **assert**. (Alternative: Visual Studio Test Framework oder freie Unit Test Bibliotheken.)

11.3 Aufgabe: Übergabearten

Erstellen Sie eine Klasse `Helper` und implementieren Sie folgende Methoden mit einer jeweiligen aussagekräftigen Konsolenausgabe:

- Default-Konstruktor
- Copy-Konstruktor
- Destruktor

Beantworten Sie folgende Fragen:

- (1) Wann wird welcher Konstruktor aufgerufen (Bedingung)?
- (2) Wo liegt der Unterschied zwischen den Übergabearten?

by-value, by-ref und by-address

- (3) Wann sollten Sie welche Übergabeart wählen und warum?
- (4) Was passiert wenn Sie vor dem Übergabetyp ein `const` schreiben?

`func(const Helper object)`

- (5) Worin unterscheidet sich `const Helper*` und `Helper const*`?

```
void func_1(Helper object_by_value) {
    std::cout << "func_1 was called, object by value" << std::endl;
}

void func_2(Helper& object_by_ref) {
    std::cout << "func_1 was called, object by reference" << std::endl;
}

void func_3(Helper* object_by_address) {
    std::cout << "func_1 was called, object by address" << std::endl;
}
```



```
void main() {  
    std::cout << "Test 1 - started" << std::endl;  
    {  
        Helper a;  
  
        std::cout << "func_1 will be calling..." << std::endl;  
        func_1(a);  
  
        std::cout << "func_2 will be calling..." << std::endl;  
        func_2(a);  
  
        std::cout << "func_3 will be calling..." << std::endl;  
        func_3(&a);  
    }  
    std::cout << "Test 1 - finished" << std::endl;  
  
    std::cout << "Test 2 - started" << std::endl;  
    Helper global;  
    {  
        Helper a;  
        global = a;  
    }  
    std::cout << "Test 2 - finished" << std::endl;  
}
```

11.4 Aufgabe: Blaue Donau

Erstellen Sie ein Programm zur einfachen Verschlüsselung kompletter Zeichenketten mit der Geheimschrift Blaue Donau.

	B	L	A	U	E
D	A	B	C	D	E
O	F	G	H	I	K
N	L	M	N	O	P
A	Q	R	S	T	U
U	V	W	X	Y	Z

Hiernach würde das A im Klartext in der Chiffre als BD erscheinen. Hinweis: J wird durch Y ersetzt, in der Chiffre also zu UU!

- Skizzieren Sie mit Hilfe eines Struktogramms o. ä. das Programm.
- Verwenden Sie zur Lösung dieser Aufgabe die objektorientierte Programmierung.
- Berücksichtigen Sie im Entwurf, dass weitere Verschlüsselungsverfahren implementiert werden können.

11.5 Aufgabe: Vektor

11.5.1 Einfacher Vektor mit dem Datentyp `int`

Implementieren Sie eine Klasse (*IntVector*) zur Verwaltung eines zusammenhängenden Speicherblocks für den atomaren Datentyp `int`. Der Speicherblock soll **dynamisch** veränderbar sein. Diese Klasse soll folgende Anforderungen erfüllen:

Grundgerüst

- (1) Es soll möglich sein, einen leeren Vektor zu erzeugen (-> Keine Speicherallokation).
- (2) Es soll möglich sein, beim konstruieren des Objektes die Anzahl der zu allozierenden Elemente und optional dessen Initialisierungswert zu übergeben.
- (3) Es soll möglich sein, ein Element am Anfang (*push_front*) des Speicherblocks hinzuzufügen.
- (4) Es soll möglich sein, ein Element am Anfang (*pop_front*) des Speicherblocks zu löschen.
- (5) Es soll möglich sein, ein Element am Ende (*push_back*) des Speicherblocks hinzuzufügen.
- (6) Es soll möglich sein, ein Element am Ende (*pop_back*) des Speicherblocks zu löschen.
- (7) Es soll möglich sein, die Anzahl der Elemente mit einer Methode zu ändern. Es soll optional möglich sein die neuhinzugefügten Elementen einen Initialisierungswert zuzuweisen (*resize*).
- (8) Es soll möglich sein, die aktuelle Größe des Vektors zu bestimmen (*size*).
- (9) Es soll möglich sein, zu prüfen ob der Vektor leer ist (*empty*).
- (10) Es soll möglich sein auf einzelne Werte zuzugreifen (*at*).
- (11) Es soll möglich sein einzelne Werte zu ändern (*set*).
- (12) Es soll möglich sein eine echte Kopie (vgl. Deep-Copy) des Vektors anzulegen.
- (13) Es soll kein Speicherloch entstehen (-> Korrekte Speicherfreigabe, auch beim zerstören des Objektes.)

Komfort

- (14) Es soll möglich sein, mit Hilfe des `[]`-Operators auf einzelne Werte zuzugreifen.
- (15) Es soll möglich sein, mit Hilfe des `==`-Operators die Gleichheit zweier Vektoren zu testen.
- (16) Es soll möglich sein, mit Hilfe des `!=`-Operators die Ungleichheit zweier Vektoren zu testen.

Beispiel

Code	Anforderung
<code>IntVector myEmptyVector;</code>	(1)
<code>IntVector myVector(5, 10); // 5 Elemente - 10 = Standardwert</code>	(2)
<code>IntVector myVector8(8); // 8 Elemente</code>	(2)
<code>myVector.push_front(123); // [123] [...]</code>	(3)
<code>myVector.push_back(321); // [...] [321]</code>	(5)
<code>std::cout << "size = " << myVector.size() << std::endl;</code>	(8)
<code>std::cout << "empty = " << (myVector.empty() ? "true" : "false")</code>	(9)
<code><< std::endl;</code>	
<code>std::cout << "element[0] = " << myVector.at(0) << std::endl;</code>	(10)
<code>myVector.set(0, 333);</code>	(11)
<code>IntVector otherVector(myVector);</code>	(12)
<code>std::cout << "element[0] = " << myVector[0] << std::endl;</code>	(13)
<code>myVector[0] = 345;</code>	
<code>std::cout << "otherVector == myVector = " << (otherVector ==</code>	(15)
<code>myVector ? "true" : "false") << std::endl;</code>	

11.5.2 Copy-On-Write Mechanismus

Der Mechanismus Copy-On-Write (COW) dient dazu, eine Speicherallokation erst bei einer anstehenden Änderung eines geteilten Speicherinhalts (=über mehrere Objekte) zu kopieren und dann erst die Änderung durchzuführen.

Kopieren Sie die Vektor Implementierung und implementieren Sie den Copy-On-Write Mechanismus in dieser neuen Klasse (*COWIntVector*).

Code
<pre> COWIntVector cowVector; // 1. interne Speicherallokation mit 5 Elementen cowVector(5); // Keine interne Speicherallokation! // -> Daten werden geteilt. COWIntVector tmpVector(cowVector); // Keine interne Speicherallokation! bool empty = tmpVector.empty(); // 2. interne Speicherallokation // -> Speicherblock wird kopiert. tmpVector.set(3, 4); // Keine interne Speicherallokation // -> Daten werden nicht geteilt. tmpVector.set(2, 1); </pre>

11.5.3 Vektor für beliebige atomare Datentypen

Kopieren und erweitern Sie die Implementierung aus der Aufgabe Vektor (*IntVector*) oder der Zusatzaufgabe (*COWIntVector*), dass diese Implementierung jeden beliebigen (atomaren) Datentyp unterstützt.

```
template<class T>
class Vector {
public:
    void push_back(T value) {
        // to be define
    }

    void push_front(T value) {
        // to be define
    }

    // to be define

private:
    // to be define
};
```

Beispiel

Code

```
Vector<int> myIntVector;
myIntVector.push_back(1);
myIntVector.push_back(2);
myIntVector.push_back(3);

Vector<double> myDoubleVector;
myDoubleVector.push_front(2.5);
myDoubleVector.push_front(4.7);
myDoubleVector.push_front(8.5);
```

11.6 Aufgabe: Bildbearbeitung

Lesen Sie folgendes 24 bpp Bitmap ein und verändern Sie die Farbwerte.



Die Beschreibung des Dateiformats ist u. A. auf wikipedia zu finden.

- Entwerfen Sie eine Programmskizze zur Lösung des dargestellten Problems unter Berücksichtigung der objektorientierten Programmierung.
- Lesen Sie das angegebene Bild (24 bit) ein.
- Invertieren Sie die Farbwerte und speichern Sie das Bild unter einem neuen Namen.
- Entfernen Sie jeweils den Rot-, Grün- und Blau-Kanal und speichern Sie jeweils das Bild unter einen neuen (eindeutigen) Namen.

11.7 Aufgabe: Sudoku

			8		1			
9		5				1		4
	1	7		5		9	2	
6	4		3		9		8	7
	9		4		5		1	
5	3		1		7		4	9
	2	6		1		8	7	
1		9				6		2
			2		6			

Bereits vor etwa fünfzehn Jahren in Japan entwickelt, eroberten die Sudoku-Rätsel schlagartig den Rest der Welt. Das Prinzip hinter Sudoku ist denkbar einfach: Es geht um die Anordnung von Zahlen in einer bestimmten Abfolge, und das hat nichts mit Mathematik oder Rätselraten zu tun, sondern ausschließlich mit logischem Denken.

Jedes Sudoku Gitter ist einzigartig und besteht aus neun Blöcken, die jeweils neun Felder enthalten. Ziel des Rätsels ist es, das Gitter so auszufüllen, dass in jeder Reihe, in jeder Spalte und jedem Block von 3 x 3 Feldern die Ziffern 1 bis 9 jeweils nur ein Mal erscheinen. Wird eine Ziffer im falschen Feld eingetragen, lässt sich das Rätsel nicht lösen. [...] – Super-Sudoku, Yukio Suzuki, ISBN-13: 978-3-442-16861-3

- Entwerfen Sie eine Programmskizze zur Lösung des dargestellten Problems unter Berücksichtigung der objektorientierten Programmierung.
- Schreiben Sie ein Programm, welches jedes beliebige Sudoku löst (siehe Beispiel oben).
- Beachten Sie bei der Entwicklung, dass eine Erweiterung zum Hexadoku möglich ist. Hexadoku ist eine Erweiterung des Sudokus und bildet 4 x 4 Felder je Block ab und verwendet die Ziffern 1 bis 16 (oder in Hex-System ausgedrückt: 1 bis FF).
- Das Programm soll das eingegebene Sudoku in der Konsole ausgegeben.
- Das gelöste Sudoku soll in der Konsole ausgegeben werden.
- (Optional) Das Programm soll das zu lösende Sudoku aus einer CSV Datei lesen und das gelöste Sudoku wieder abspeichern können.
- (Optional) Erweitern Sie das Programm, sodass neue Sudokus generiert werden können.

11.8 Aufgabe: Prüfsumme

Erstellen Sie eine Konsolenanwendung zur Berechnung von Prüfsummen. Implementieren Sie dafür das MD5-Verfahren und berücksichtigen Sie in der Software-Architektur, dass weitere Prüfsummenverfahren implementiert werden können.

Die Eingaben sollen über Programmaufrufargumente gesteuert werden. Sind die Eingaben ungültig, so erscheint eine passende Fehlermeldung mit Hilfestellung für die korrekte Eingabe.

Beispiel:

```
checksum.exe -algorithm MD5 -file myFile.txt
```

Ausgabe:

```
7815696ecbf1c96e6894b779456d330e
```

Es sollen folgende Argumente erlaubt sein:

-algorithm <Impl> = Auswahl des Prüfsummenverfahren, Optional, Default: MD5

-path <File or Dir> = Datei oder Ordner, Pflichteingabe

Programmieren Sie möglichst Objektorientiert. Verwenden Sie bei Möglichkeit das „Factory“-Entwurfsmuster in Verbindung des Sprachmerkmals „Function Pointer“ für die Verwaltung der Prüfsummenimplementierungen.

Quellen:

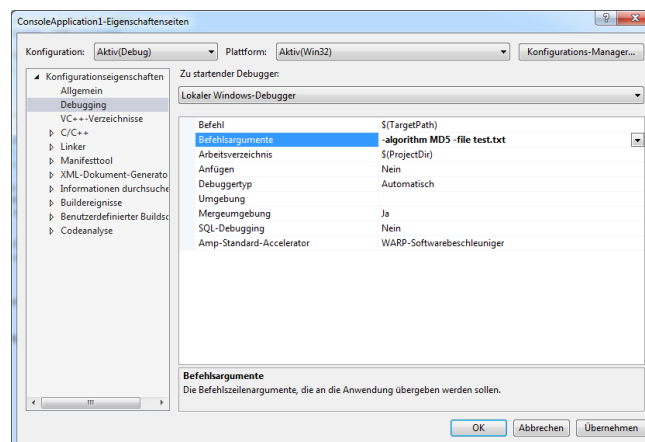
The MD5 Message-Digest Algorithm (<https://www.ietf.org/rfc/rfc1321.txt>)

Fabrikmethode (<https://de.wikipedia.org/wiki/Fabrikmethode>)

Function Pointer (<http://www.cprogramming.com/tutorial/function-pointers.html>)

Tipp:

Über die Projekteinstellungen in Visual Studio können Aufrufargumente festgelegt werden.



11.9 Aufgabe: Scalable Vector Graphic



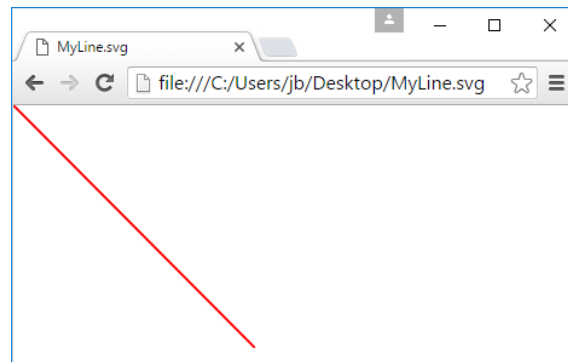
<https://dev.w3.org/SVG/tools/svgweb/samples/svg-files/car.svg>, 19.02.16 18:30

Erstellen Sie eine Bibliothek zur Verwaltung (Canvas) von Grafikobjekten (GraphicsItem). Es soll möglich sein folgende Elemente anzulegen:

- Linie (LineItem)
- Kreis (CircleItem)
- Rechteck (RectItem)

Jedes dieser Elemente sollen alle notwendige Eigenschaften besitzen, um ein abspeichern in das Scalable Vector Graphics (SVG) Format zu ermöglichen. Die Dateierweiterung dieses Formats ist SVG (Beispiel: MyLine.svg). Die SVG Dateien können direkt im Browser (Internet Explorer, Firefox...) dargestellt werden. Im Ersten Schritt reicht es aus, dass nur die Positionen und Ausdehnungen der Objekte veränderbar sind.

Ein einfaches Beispiel (MyLine.svg):



```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg xmlns="http://www.w3.org/2000/svg"
    version="1.1"
    width="200" height="200" >

    <line x1="0" y1="0" x2="200" y2="200" style="stroke:rgb(255,0,0);stroke-
width:2" />

</svg>
```

11.9.1 Anforderungen


- (17) Es soll möglichst Objektorientiert programmiert werden.
- (18) Die Verwaltungsstruktur (Canvas) übernimmt die Speicherverwaltung. Dies bedeutet, dass nur diese die Grafikelemente erstellt und auch löscht.
- (19) Die Verwaltungsstruktur nutzt ausschließlich die Basisklasse von den Grafikelementen (GraphicsItem) für die Verwaltung (keine separate Verwaltung für LineItem, CircleItem und RectItem).
- (20) Es soll möglich sein, neue Grafikelemente der Verwaltung (Canvas) hinzuzufügen (addLine, addCircle, ...).
- (21) Es soll möglich sein, jede Eigenschaft des konkreten Grafikelements über eine Methode zu setzen und auszulesen.
- (22) Es soll möglich sein, bestehende Grafikelemente aus der Verwaltung (Canvas) zu löschen (remove).
- (23) Es soll möglich sein, den aktuellen Zustand als Datei im SVG Format zu speichern (save).
- (24) Es soll möglich sein, jedes beliebige Grafikelement, um eine Differenz (dx, dy) zu verschieben.
- (25) Es soll möglich sein, von jedem Grafikelemente den Begrenzungsrahmen (BoundingBox) zu bestimmen.
- (26) Es soll möglich sein, von jedem Grafikelement die (relative) Ausdehnung (Extends) zu bestimmen.
- (27) Es soll möglich sein, jedem Grafikelemente eine Z-Ebene zuzuweisen. Die Z-Ebene bestimmt, welches Element zuerst gezeichnet wird. Dadurch ist ein überzeichnen von bereits gezeichneten Elementen möglich.
- (28) Die Größe des SVG (siehe Attribut width und height in dem Tag svg) soll automatisch durch die Verwaltung bestimmt werden.
- (29) Es soll möglich sein, zu prüfen ob ein Grafikelement mit einem anderen kollidiert (collide) auf Basis des Begrenzungsrahmens (BoundingBox).
- (30) Stellen Sie sicher, dass die komplette Implementierung Fehlerfrei ist.

Folgender Code soll das oben angeführte Beispiel erzeugen:

Code
<pre>Canvas canvas; LineItem* lineItem = canvas.addLine(0, 0, 100, 100); lineItem->setEnd(200, 200); canvas.save("MyLine.svg");</pre>

11.9.2 Kurzreferenz

Linie

 <pre><line x1="0" y1="0" x2="200" y2="200" style="stroke:rgb(255,0,0);stroke-width:2" /></pre>
--

- x1: Startposition auf der X-Achse
- y1: Startposition auf der Y-Achse

- x2: Endposition auf der X-Achse
- y2: Endposition auf der Y-Achse
- style: Darstellungsart (vgl. CSS)
 - stroke: Linienfarbe
 - stroke-width: Linienstärke

→ http://www.w3schools.com/svg/svg_line.asp

Kreis



```
<circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" />
```

- cx: Mittelpunktposition auf der X-Achse
- cy: Mittelpunktposition auf der Y-Achse
- r: Radius
- stroke: Linienfarbe
- stroke-width: Linienstärke
- fill: Füllfarbe

→ http://www.w3schools.com/svg/svg_circle.asp

Rechteck



```
<rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)" />
```

- width: Breite
- height: Höhe
- style: Darstellungsart
 - fill: Füllfarbe
 - stroke-width: Linienstärke
 - stroke: Linienfarbe

→ http://www.w3schools.com/svg/svg_rect.asp

Für weitere Informationen siehe:

- https://de.wikipedia.org/wiki/Scalable_Vector_Graphics
- <http://www.w3schools.com/svg/default.asp>