C++ Crashkurs

Prof. Dr. Andreas Plaß

Department Medientechnik

HAW Hamburg

Inhaltsverzeichnis

1 C++ und Java

1.1 Geschichte

Die Programmiersprache C stammt aus dem Jahr 1972 und war für damalige Computer entworfen:

- Schnell
- Geringer Speicherplatzbedarf
- direkter Zugriff auf Hardware
- wenig overhead

Für Desktop-Computer ist das heute nicht mehr relevant, aber zum Beispiel Micro-Controller werden häufig in C programmiert.

Die Programmiersprache C++ (Bjarne Stroustrup, ab 1979) ist Nachfolger von C:

- abwärtskompatibel zu C
- objektorientiert
- genauso effizient wie C

Die Programmiersprache Java wurde 1995 als rein objektorientierte Programmiersprache konzipiert. Von ihrer Syntax her sind Java und C++ verwandt. Sie verfolgen allerdings unterschiedliche Philosophien.

1.2 Plattformabhängigkeit

C++ ist plattform-abhängig. Auch Code, der von unterschiedlichen Compilern (z.B. MingGW vs. Visual Studio) oder unterschiedlichen Compiler-Versionen erstellt wurde, funktioniert häufig nicht. Dafür benötigen C++ Programme keine virtuelle Maschine und laufen direkt auf dem Zielsystem.

1.3 Geschwindigkeit vs. Sicherheit

„C++'s main priority is getting correct programs to run as fast as they can; incorrect programs are on their own. Java's main priority is not allowing incorrect programs to run; hopefully correct programs run reasonably fast, and the language makes it easier to generate correct programs by restricting some bad programming constructs" (Mark Allen Weiss, C++ for Java Programmers)

C++ Hauptfocus ist Geschwindigkeit, auf Kosten eventueller Sicherheitsüberprüfungen, z.B.

- Speicher muss explizit freigegeben werden (C++ hat keine garbage collection)
- beliebige Speicheradressen können verändert werden (auch versehentlich)
- keine Überprüfung von Array-Indizes
- keine automatische Initialisierung von Variablen

1.4 Threads, APIs

C++ bietet kaum APIs – mit Ausnahme der *Standard Template Library (STL).* Insbesondere gibt es keine standardisierte Thread-API.

1.5 Gründe für C++

- C++ ist schnell, daher für Medien-Anwendungen (Signalverarbeitung) gut geeignet
- C++ ist weit verbreitet
- C++ Programme sind kleiner

2 C++ Überblick

Als Einstieg dient ein *Hello World* Programm, das die Grundstruktur von C++ erläutert. Variablen und Kontrollstrukturen sind ähnlich zu denen von Java. Jedoch fehlen einige wichtige Sicherheitsüberprüfungen des Compilers.

2.1 Hello World

Folgendes Beispiel main.cpp zeigt ein einfaches *Hello World* Programm in C++:

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5    cout << "Hello World" << endl;
6    return 0;
7 }
```

Die wichtigsten Eigenschaften – auch im Vergleich zu Java sind:

- Einstiegspunkt ist die **Funktion** `main()`
- Name der Datei ist beliebig, in der Regel mit Endung `.cpp`
- Zeile 1: externe Dateien werden mit #include eingebunden
- Zeile 2: C++ kennt *namespaces,* diese sind in etwa mit Java-Paketen vergleichbar – aber flexibler einseztbar.
- Zeile 4: Einstiegspunkt ist die Funktion main().
- Zeile 5: Ausgabe eines Strings an cout (Standard-Ausgabe) mit dem Shift-Operator <<

Aufgaben

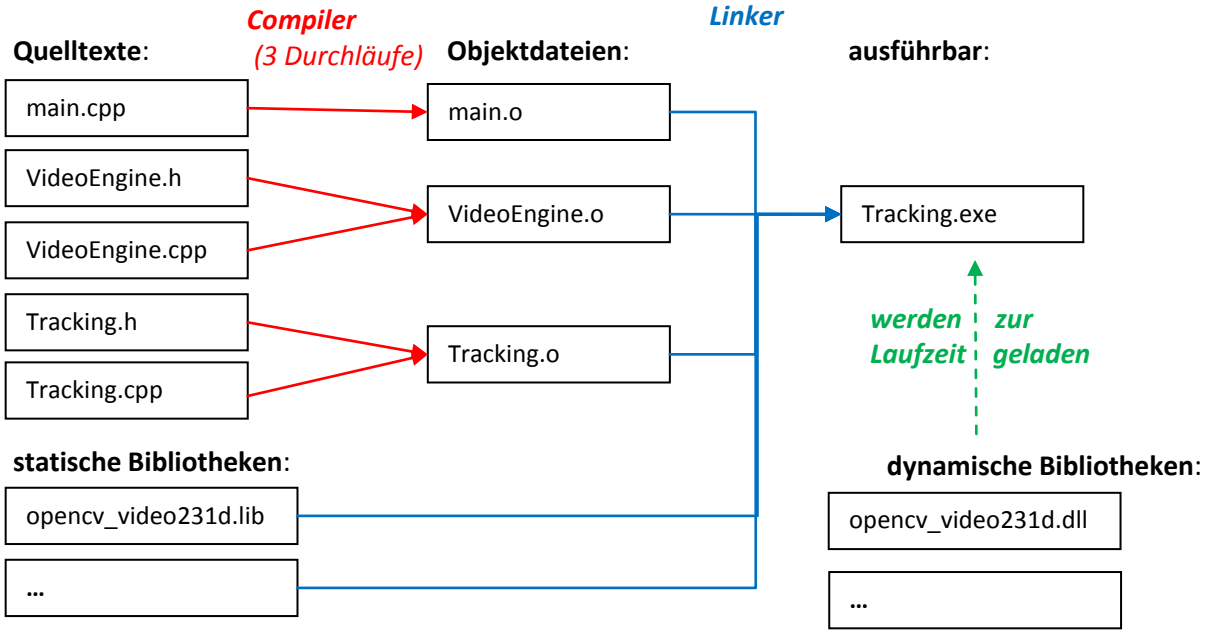
1. Schreiben Sie ein Programm *HierBinIch,* mit dem Sie sich vorstellen: Name, Vorname, Matrikelnummer, jeweils in getrennten Zeilen.
2. Suchen Sie die ausführbare Datei im Projektordner und starten Sie sie dort mit Doppelklick.
3. Schreiben Sie ein Programm, das zwei Zahlen addiert. Die Zahlen werden über die Tastatur eingelesen, das Ergebnis wird auf der Konsole ausgegeben.

2.2 Linker

Beim Erstellen eines C oder eines C++ Programms sind drei Werkzeuge im Einsatz:

- Präprozessor: Einbindung externer Dateien, bedingter Quelltext, Makros
- Compiler: compiliert Quelltext und erzeugt Objektdatei (*.o oder *.obj)
- Linker: verbindet einzelne Objektdateien und externe Bibliotheken.

Das folgende Beispiel zeigt den Erstellungsprozess eines einfachen Programms, das die *opencv* Bibliotheken verwendet. Die fertige Software *tracking.exe* lädt zur Laufzeit die externen dynamischen Bibliotheken (z.B. Endung **.dll* oder **.so*)



2.3 Elementare Datentypen und Variablen

Die Programmiersprachen C/C++ verwenden die gleichen elementaren Datentypen wie Java – bis auf einige kleine feine Unterschiede. So ist die Bitbreite der Datentypen nicht standardisiert. Sie kann plattform- oder Compiler-abhängig sein. Bei einigen Datentypen können die Eigenschaften sogar in den Compiler-Einstellungen variiert werden (char als ASCII oder UNICODE). Folgende Tabelle zeigt die elementaren Zahlen-Datentypen und ihre Eigenschaften des *Microsoft* Visual-C++ Compilers:

Datentyp	Wortbreite	kleinste Zahl	größte Zahl
`char`	8	-128	127
`short`	16	-32768	32767
`int`	32	-2^{31} = -2147483648	2^{31}-1 = 2147483647
`long`	32	-2^{31} = -2147483648	2^{31}-1 = 2147483647
`unsigned char`	8	0	255
`unsigned short`	16	0	65535
`unsigned int`	32	0	2^{32-1} = 4294967295

Unterschiede zu Java

bool: Für Wahrheitswerte definiert C++ den Datentyp bool mit den Werten false, true. (In C ist bool nicht definiert, stattdessen wird int verwendet).
char: Für Zeichen verwendet C++ den Datentyp char. Die meisten Compiler definieren ihn als 8-Bit Datentyp, d.h. er repräsentiert nur den ASCII-Zeichensatz. Für UNICODE-Zeichen kann dann (Compiler-abhängig) der 16-Bit-Datentyp wchar_t verwendet werden.
long: Viele C++-Compiler definieren long als 32-Bit Ganzzahl-Datentyp – *im Unterschied zu Java!* Als 64-Bit Ganzzahl-Datentyp definiert Visual Studio den Typ `__int64`, andere Compiler definieren den Datentyp long long.
Signed/Unsigned: Bei den Ganzzahl-Datentypen gibt es die *signed* und die *unsigned* Variante, sie werden durch Voranstellung des entsprechenden Schlüsselworts definiert. Die Variable im folgenden Beispiel repräsentiert einen Farbkanal mit Wertebereich 0...255:

```
unsigned char colorChannel;
```

Typedef

Das typedef-Schlüsselwort erlaubt die einfache Definition neuer Datentypen, dabei wird ein *Alias* erzeugt. Hier einige typische Anwendungsbeispiele:
byte: ein Datentyp für positive 8-Bit Zahlen (0 ... 255)

```
typedef unsigned char byte;
```

uint32: viele Softwarebibliotheken definieren eigene elementare Datentypen, um so das Problem Compiler-abhängiger Bitbreiten zu lösen:

```
#if WIN32
typedef unsigned int uint32;
#endif
```

size_t : Die Standardbibliothek verwendet den Datentyp `size_t` für Indizes oder für Anzahl der Elemente eines Vectors oder eines anderen Containers. Er ist in der Regel definiert als

```
typedef unsigned int size_t;
```

Sizeof()-Operator

Der sizeof()-Operator liefert die Bitbreite eines Datentyps, Beispiel:

```
cout << sizeof(bool) << endl;     // Ausgabe: 8
```

Typumwandlungen

In C und C++ werden bei Wertzuweisungen oder Ausdrücken die Datentypen automatisch in den Ziel-Typ umgewandelt. Zum Beispiel liefert folgender Code wahrscheinlich Compiler-Warnungen, aber er lässt sich compilieren (in Java jedoch nicht!)

```
double x = 6.0;
int i;
i = x;
```

Hier ist es empfehlenswert, einen expliziten Typecast zu verwenden. Dies geht mit der Java-Syntax: `i = (int) x` oder auch so: `i = int(x)`. Besser ist jedoch folgende modernere Syntax

```
double x = 6.0;
int i;
i = static_cast<int>(x);
```

Die Verwendung des Operators `static_cast<…>()` hat den Vorteil, dass sie beim *Suchen & Ersetzen* leichter zu finden ist und sie sollte daher auch verwendet werden.

Initialisierung von Variablen

In C++ werden Variablen **nicht automatisch initialisiert**. Das ist eine häufige Fehlerquelle.
Beispiel: folgender Quelltext lässt sich compilieren, liefert aber ein undefiniertes Ergebnis wenn die Argumente a und b den gleichen Wert haben:

```
int findeMaximum (int a, int b) {
   int dasMaximum;
   if (a > b) {
      dasMaximum = a;
   }
   else if (b > a) {
      dasMaximum = b;
   }
   return dasMaximum;
}
```

2.4 Kontrollstrukturen

In C und C++ existieren die gleichen Kontrollstrukturen wie in Java:
Verzweigungen: if()-else if() – else
Schleifen: while(), do-while(), for()

Ein wichtiger Unterschied ist jedoch zu beachten. Er führt häufig zu Fehlern. Das Argument der `if()`–Bedingung darf in C++ ein `bool` **oder ein** `int` sein. Das kann leicht zu folgendem **fehlerhaftem Code** führen:

```
if (x = 0) {                  // <-- Fehler: x wird immer auf Null
                              //     gesetzt und ausgewertet
   machEtwasGanzWichtiges(); // <-- wird nie aufgerufen!!!
}
```

Häufig findet man auch Code der folgenden - schwerer lesbaren - Form:

```
if (x) {
   ...
}
```

stattdessen sollte man die ausführliche Form vorziehen

```
if (x != 0) {
   ...
}
```

Regel: Bei Bedingungen `if(), while(), for()` sollte immer die Java-Syntax verwendet werden!

2.5 Präprozessoranweisungen

Präprozessor-Anweisungen werden ***vor*** dem Compilieren ausgeführt, d.h. sie modifizieren den Quelltext vor dem Compiler-Durchlauf.

#include

Zum Einbinden externer Header-Dateien (im Prinzip sind auch andere Dateiarten möglich) dient die #include-Anweisung. Systemdateien werden in spitzen Klammern angegeben:

```
#include <iostream>
```

Eigene Headerdateien werden in Anführungsstrichen angegeben:

```
#include "MeineKlasse.h"
```

#if, #ifdef

Plattform-spezifischer Code kann mit den #if, #ifdef Anweisungen definiert werden. Der Code wird nur dann compiliert, wenn eine entsprechende Konstante definiert ist , oder einen bestimmten Wert hat:

```
#ifdef WIN32
  // diesen Code "sieht" nur der Windows-Compiler
#else
  // diesen Code "sieht" nur der Mac-Compiler
#endif
```

Entsprechende Konstanten können mit der #define Anweisung definiert werden, z.B.

```
#define MEDIA_SYSTEMS 1
```

3 Arrays

Arrays sind in C oder C++ Adressen von bestimmten reservierten Speicherbereichen. Es fehlen komfortablere Features wie das `length`-Attribut (in Java die Länge eines Arrays) oder Sicherheitsmechanismen gegen Überschreiben eines Arrays (in Java `ArrayIndexOutOfBoundsException`).

C++ definiert in der *Standard Template Library* eine eigene Klasse `vector`, die in den meisten Anwendungen ein Array ersetzen kann.

3.1 Zeiger

Eine **Zeigervariable** - kurz Zeiger (pointer) - ist eine Variable, deren Wert eine Speicheradresse ist. Eine Zeigervariable zeigt auf einen Datentyp, dies ist ein Wert oder ein Objekt einer Klasse.

Deklaration: Zeigervariablen werden als Zeiger auf einen Datentyp durch hintangestellten Stern deklariert. Im folgenden Beispiel wird ein Zeiger auf eine int-Variable deklariert und mit 0 initialisiert.

```
int * zeigerAufA = 0;
```

Adress-Operator *: Das & Zeichen dient als Adressoperator, z.B.

```
int a = 5;
zeigerAufA = &a;
```

Hier wird der Variablen `zeigerAufA` die Adresse der Variablen a zugewiesen.
De-Referenzierungsoperator *: Der *-Operator liefert den Wert der an einer bestimmten Adresse liegt. Im Beispiel wird der int-Wert geholt, auf den der Zeiger `zeigerAufA` zeigt.

```
int  b = *zeigerAufA;
```

Anwendungen

Zeiger auf Arrays: C-Arrays werden als Zeiger auf das erste Arrayelement deklariert. Funktionen, die ein Array als Parameter erwarten, deklarieren ein solches Argument als Zeiger. Folgende Funktion erwartet einen String `filename` als Array von `char`-Elementen:

```
IplImage* cvLoadImage(const char* filename, int iscolor)
```

Zeiger auf Objekte: Variablen, die Objekte einer Klasse oder Datenstruktur repräsentieren, können als Zeigervariablen deklariert werden. Die oben gezeigte Funktion `cvLoadImage`() liefert als Rückgabewert einen Zeiger auf ein Objekt der Klasse `IplImage`. Im folgenden Beispiel wird eine Variable image deklariert, sie ist ein Zeiger auf ein Objekt vom Typ QImage:

```
QImage* image;
```

Soll eine Methode aufgerufen werden, z.B. load() von QImage, wird zunächst das Objekt de-referenziert, und dann die Methode aufgerufen:

```
(*image).load();
```

Hierfür gibt es alternativ den Pfeil-Operator ->

```
image->load();
```

Zeigerarithmetik: C erlaubt arithmetische Operationen mit Zeigervariablen. So sieht man häufig folgenden Code, der die Elemente von Array b in Array a kopiert.

```
for(int i = 0; i < 10; i++){
   *a++ = *b++;
}
```

Dieser Code ist **schwer lesbar** und **sehr fehlerträchtig.** Deshalb sollte Zeigerarithmetik möglichst vermieden werden!

3.2 Arrays

Arrays enthalten mehrere Elemente gleichen Datentyps in einer durchnummerierten Liste. Arrays in C sehen zunächst sehr ähnlich den Java-Arrays. Mit C-Arrays hat der Programmierer jedoch einen direkten Zugriff auf den Speicher. Dies führt häufig zu fatalen Fehlern (*Buffer overflow*), so dass entsprechende Sorgfalt notwendig ist.

Arrays fester Größe

Die einfachste Form von Arrays sind Arrays fester Größe. Hier wird die Arraygröße zur *Compilierzeit* festgelegt:

```
int zahlen[1000];             // direkte Initialisierung,
...                           // delete ist nicht notwendig

int zahlen[] = {1, 5, 4, 3}; // direkte Initialisierung
                              // delete ist nicht erlaubt
```

Diese Art von Arrays können genauso verwendet werden wie Java-Arrays. Insbesondere muss der Speicher nicht freigegeben werden.

Dynamische Arrays

Häufiger verwendet sind dynamsiche Arrays, deren Größe erst zur Laufzeit festgelegt wird. Ein Array mit Elementen vom Typ double kann auf folgende Weise definiert werden:

```
double *daten;
daten = new double[1000];
```

Die Variable daten wird als Zeiger auf das erste Element deklariert.

Achtung: Nach der Initialisierung sind die Elemente eines Arrays ***nicht initialisiert!***

Der Zugriff auf die Elemente erfolgt mit den Index-Operator []:

```
for (int i = 0; i < 1000; i++){
   daten[i] = rand();
}
```

Nach der Verwendung muss der reservierte Speicherbereich mit delete[] explizit wieder freigegeben werden:

```
delete[] daten;
```

Funktionen mit Arrays

Wenn eine Funktion ein Array als Argument erwartet, muss in der Regel die Länge des Arrays als zusätzliches Argument an die Funktion übergeben werden. Beispiel: Die Funktion `berechneMittelwert()` berechnet den Mittelwert der Zahlen eines ihr übergebenen Arrays:

```
int berechneMittelwert(int *data, int numberOfElements){
   int mittelwert = 0;
   for (int i = 0; i < numberOfElements; i++){
      mittelwert += data[i];
   }
   return mittelwert/numberOfElements;
}
```

Im Gegensatz zu Java sollten Funktionen jedoch **keine Arrays zurückgeben**. Der Grund dafür ist, dass später die Freigabe des Speicherbereichs leicht vergessen wird. Stattdessen wird als Argument ein Array übergeben, in das die Funktion die Ausgangsdaten schreibt. Beispiel: Folgende Funktion kopiert die Daten vom Array `input` in das Array `output`:

```
void kopieren(float *input, float *output, int numberOfElements){
   for (int i = 0; i < numberOfElements; i++){
      output[i] = input[i];
   }
}
```

3.3 Die Klasse vector

C++ definiert in der *standard template library (STL)* als elegante Alternative zu C-Arrays die Klasse `vector`, die in selbstgeschriebenen Programmen möglichst an deren Stelle verwendet werden sollte. Sie ist der Klasse ArrayList aus Java vergleichbar.
Die Klasse `vector` ist ein Template, d.h. ein parametrisierter Datentyp. Der Datentyp der Elemente wird in spitzen Klammern angegeben. Die Länge des Arrays wird im Konstruktor übergeben.
Folgender Code erzeugt ein `vector`-Objekt und initialisiert seine Elemente mit dem Wert 0:

```
#include <vector>
...
vector<int> data(arraySize);
```

Alternativ kann das `vector`-Objekt mit der Methode `resize`() erzeugt bzw. geändert werden:

```
#include <vector>
...
vector<int> data;
data.resize(arraySize);
```

Die Methode `size`() liefert die Anzahl der Arrayelemente:

```
int numberOfElements = data.size();
```

Auf die Elemente wird mit dem Indexoperator [] zugegriffen:

```
int summe = 0;
for (int i = 0; i < data.size(); i++){
   summe += data[i];
}
```

Hinweis: In diesem Beispiel ist die Variable `data` ein Objekt von `vector<int>` - nicht ein Zeiger auf ein solches Objekt. Daher ist ein `delete` nicht notwendig und **darf auch nicht verwendet werden.**

Verwendung von vector mit C-Funktionen

Erwartet eine C-Funktion ein Array, so kann diese Funktion auch zusammen mit `vector`-Objekten verwendet werden.
Beispiel: Folgende C-Funktion soll den Mittelwert der Arrayelemente berechnen:

```
int berechneMittelwert(int *data, int numberOfElements){
   int mittelwert = 0;
   for (int i = 0; i < numberOfElements; i++){
      mittelwert += data[i];
   }
   return mittelwert/numberOfElements;
}
```

Folgendes Codefragment verwendet diese Funktion:

```
vector<int> myData;
myData.resize(blockSize);
....
int mittelwert = berechneMittelwert(&myData[0], myData.size());
```

Es wird mit `&myData[0]` die Adresse des ersten Elements des Vektors übergeben. Dies ist vom Typ int* und damit kompatibel mit der Funktionssignatur.

Nachteile von C-Arrays
C-Arrays erlauben einen direkten Speicherzugriff. Dies wird jedoch mit gravierenden Nachteilen in Kauf genommen, die immer wieder zu schweren Programmfehlern führen:

- kein Schutz vor unerlaubten Indizes: dies kann zu ungültigen Speicherzugriffen führen (Java kennt dafür `ArrayIndexOutOfBoundsException`)
- fehlendes delete führt zu Speicherlöchern
- keine Information über Arraylänge: C-Arrays haben kein `length`-Attribut. Beim Aufruf von Funktionen muss die Länge eines Arrays immer als zusätzlicher Parameter übergeben werden
- keine Default-Initialisierung: C-Arrays haben nach ihrer Erzeugung mit `new` einen undefinierten Inhalt, was zu Programfehlern führen kann

3.4 Strings

Strings (Zeichenketten) sind in C++ Objekte vom Datentyp string.

```
string begruessung = "Hello World";
```

Mit dem + Operator können Strings aneinandergehängt werden:

```
begruessung += "!";
```

Die Klasse `string` hat eine Reihe nützlicher Methoden, wie z.B. `size`(), `substr`(), usw.

Ein- und Ausgabe von Strings

Für die Ein- und Ausgabe verwendet C++ *stream*-Objekte. Mit den Shift-Operatoren >> und << können Strings, Zahlen und Objekte an einen Datenstrom gesendet, bzw. von dort gelesen werden. Dies wurde bereits im *Helloworld* Programm genutzt:

```
cout << "Hello World";
```

Die Variable `cout` repräsentiert die Konsolenausgabe, sie ist in der Standardbibliothek vordefiniert. Das Gegenstück zu `cout` ist das Objekt `cin`. Folgender Code liest eine Ganzzahl von der Tastatur ein:

```
int eingabe;
cin >> eingabe;
```

Umwandlung von Zahlen in String

Für die Umwandlung von Zahlen in Strings verwendet C++ die gleiche Strategie wie bei der Konsolenausgabe von Zahlen. Dazu wird ein `stringstream`-Objekt erzeugt, in das mit dem Shift-Operator Zahlen geschrieben werden können.

```
int number = 5;
ostringstream os;
os << number;
string result = os.str();
```

C-Strings

In der Programmiersprache C werden Strings als Arrays vom Typ `char` behandelt, deren letztes Element die Zahl 0 ist. Beispiel:

```
char * text = "Hello";
```

C-Strings kommen zum Beispiel bei der Verwendung externer Bibliotheken zum Einsatz. So hat *opencv* beispielsweise die Funktion

```
int cvNamedWindow( const char* name, int flags);
```

Die Klasse `string` hat mit ihrer Memberfunktion c`_str`() eine Brücke zu C-Strings.

```
string windowName = "openCV Window";
int window = cvNamedWindow(windowName.c_str(), 0);
```

Hinweis: C++ Programme sollten für Zeichenkette immer die Klasse `string` verwenden. Sie können mit c_`str`() in C-Strings gewandelt werden.

Aufgaben

1. Schreiben Sie ein Programm, das die Zahlen von 1 bis 100 hochzählt und ausgibt mit je 10 Zahlen in einer Zeile.

2. Definieren Sie mit Hilfe des `typedef` Befehls einen neuen Datentyp `SehrGrosseZahl` als ein 64-Bit Ganzzahl-Datentyp.

3. Schreiben Sie ein Programm, das die Fakultät einer von der Konsole eingegebenen Zahl berechnet. Verwenden Sie dazu den in Teil 2 definierten Datentyp.

4. Schreiben Sie ein Programm, das eine Zahl N von der Tastatur einliest, dann N Zahlen von der Tastatur einliest und diese anschließend wieder ausgibt.

5. Schreiben Sie ein Programm, das alle Primzahlen bis zu einer Zahl N (von Tastatur eingelesen) ausgibt. Verwenden Sie den Algorithmus *Das Sieb des Erastothenes.*

4 Funktionen

C++ erlaubt Funktionen, die nicht Member einer Klasse sind, die werden einfach *Funktionen* genannt. Sie sind etwa den statischen Java Methoden vergleichbar. In der Regel

4.1 Headerdateien

Die Definition einer Funktion geschieht in der Regel aufgeteilt in zwei Dateien:

- Headerdatei: **Funktionsdeklaration** (auch **Funktionsprototyp** genannt)
- Implementierungsdatei: **Funktionsimplementation**

Als Beispiel soll die Funktion `max2()` in der Headerdatei `max2.h` deklariert werden.

```
#ifndef MAX2_H
#define MAX2_H

int max2(int a, int b);

#endif
```

Headerdateien in C++ sollten sogenannte *include guards* (#ifndef … #endif) enthalten, die verhindern, dass eine Headerdatei mehrfach eingebunden wird.

Die eigentliche Funktoin wird in der Implementierungsdatei `max2.cpp` definiert.

```
#include "max2.h"

int max2(int a, int b) {
   if (a > b) {
      return a;
   }
   else {
      return b;
   }
}
```

4.2 Inline Funktion

C++ erlaubt es aber auch, Deklaration und Implementierung zusammenzufassen. Dann muss die Funktion inline deklariert sein. Dies ist darüber hinaus ein Hinweis an den Funktionsrumpf an die benötigte Stelle direkt "hineinzucompilieren". Als Beispiel die Funktion `min2()`

```
#ifndef MIN2_H
#define MIN2_H

inline int min2(int a, int b) {
   if (a > b) {
      return a;
   }
   else {
      return b;
   }
}
#endif
```

Solch *inline* Funktionen werden meistens aus Performancegründen verwendet. Ausserdem kommen Sie bei Template-Libraries zum Einsatz, um so unabhängig von unterschiedlichen Binärformaten zu werden (z.B. die *opencv* C++ Wrapper)

4.3 Parameterübergabe

In Java gibt es zwei Arten der Parameterübergabe:

- *call by value*: bei elementaren Datentypen wird der Wert des Arguments an die Funktionsparameter kopiert. Die Methode kann den Wert der ursprünglichen Variable nicht verändern.
- *call by reference*: bei Objekten und Arrays wird eine Referenz auf das Objekt übergeben. Die Methode kann den Zustand des Objekts verändern.

C++ ermöglicht für elementare Datentypen **und** Objekte/Arrays drei Arten der Parameterübergabe:
- *call by value*: übergebenes Argument (darf auch ein Objekt sein) wird kopiert
- *call by reference*: Referenz auf das Argument wird übergeben. Die Funktion kann den Wert des Arguments verändern
- *call by const reference*: Referenz auf das Argument wird übergeben. Die Funktion kann den Wert des Arguments **nicht** verändern

Call by Value: im Default-Modus der Parameterübergabe wird der Funktion eine Kopie des Arguments übergeben. Der Originalwert bleibt unangetastet.
Beispiel: Es sei diese Funktion definiert:

```
int berechneQuadrat(int zahl) {
   zahl = zahl * zahl;
   return zahl;
}
```

Beim Aufruf:

```
int a = 5;
int b = berechneQuadrat(a);
```

ist der Wert von a durch den Funktionsaufruf unverändert.
Dies ist in C++ auch mit Objekten möglich. Dort kann es aber zu Performanceproblemen führen.

```
// Funktionsdefinition
int findeMaximum(vector<int> daten){
   ...
}

// Aufruf
vector<int> zahlen;
zahlen.push_back(5);
...
zahlen.push_back(7);

int gefundenesMaximum = findeMaximum(zahlen);
```

Hier wird in der letzten Zeile der ganze Inhalt des Vektors zahlen kopiert. Das kann im ungünstigen Fall sehr lange dauern!

Call by Reference: C++ erlaubt *call by reference* bei allen Datentypen, auch bei elementaren Datentypen. Dazu wird hinter den Datentyp ein &-Zeichen gesetzt.
Beispiel: Die Funktion `swap(a,b)` soll die Werte der beiden Variablen a,b vertauschen:

```
void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
Aufruf:
int a = 5;
int b = 7;
swap(a, b);  // a: 7, b: 5
```

Call by const Reference: Diese dritte Variante der Parameterübergabe existiert in Java nicht. Das vor den Datentyp gestellte `const` garantiert, dass die Funktion die übergebene Referenz nicht verändert (sonst gibt es einen Compiler-Fehler).
Beispiel: Eine schnellere Variante der Funktion `max`().

```
int max3(const int &a, const int &b) {
   if (a > b) {
      return a;
   }
   else {
      return b;
   }
}
```

Beim Aufruf:

```
int a = 5;
int b = 7;
int c = max3(a, b);
```

entfällt die Kopie der Argumente.
Dies ist gerade bei Objekten vorteilhaft. So kann die oben beschriebene Funktion `findeMaximum`() wie folgt umgeschrieben werden:

```
int findeMaximum(const vector<int> & daten){
   ...
}
```

Bei ihrem Aufruf werden die Daten des Vektors nicht mehr kopiert.

4.4 Funktionen in C

In C gibt es keine Referenzvariablen. Stattdessen müssen Zeiger verwendet werden. Damit gibt es die Arten der Parameterübergabe:

Call by Pointer: der Funktion wird ein Pointer auf eine Variable übergeben, die Funktion kann den Wert verändern, auf den der Pointer zeigt, z.B.

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

Beim Aufruf werden die Zeiger auf entsprechende Variablen übergeben:

```
int zahla = 5;
int zahlb = 7;
swap (&zahla, &zahlb);
cout << zahla;
```

Call by const pointer: der Funktion wird ein Pointer auf eine Variable/Objekt übergeben, die Funktion **darf** den Wert **nicht verändern**, z.B.

```
int findeMaximum(const int *array){
...
}
```

4.5 Defaultparameter

C/C++ erlaubt es, Parametern einen Defaultwert zu geben. Beim Aufruf braucht dieser Parameter dann nicht mehr angegeben zu werden. Beispiel:

```
void fülleVector(vector<int> &v, int wert = 1) {
   for (int i = 0; i < v.size(); i++) {
      v[i] = wert;
```

```
    }
}
```

- `fülleVector`() erhält als Argument eine Referenz auf ein `vector<int>`-Objekt
- der Parameter `wert` hat einen Defaultwert, der implizit übergeben wird, wenn dieser Parameter nicht angegeben wurde

Folgende Aufrufe sind möglich:

```
// leerer Vektor der Groesse 10
vector <int> daten;
daten.resize(10);

// Aufruf mit einem Parameter
fülleVector(daten);
cout << daten[0];       // 1

// Aufruf mit zwei Paramtern
fülleVector(daten, 7);
cout << daten[0];       // 7
```

5 Klassen

Klassen in C++ sind wie in Java auch Datentypen. Bei ihrer Definition und Verwendung gibt es jedoch wesentliche Unterschiede. Bei C++ Klassen sind die Begrifflichkeiten etwas anders als in Java:

C++	Java	Beschreibung
Membervariable	Instanzvariable	Variable, die dem Objekt einer Klasse zugeordnet ist
Memberfunktion	Methode	Funktion, die zum Objekt einer Klasse gehört
Basisklasse	Superklasse	Klasse von der eine andere Klasse erbt
abgeleitete Klasse	Subklasse	Klasse, die von einer anderen Klasse erbt
rein virtuelle Methoden	abstrakte Methoden	Methode, die überschrieben werden muss
rein virtuelle Klassen	Interface	Klasse, die nur rein virtuelle Methoden hat

5.1 Headerdatei: Deklaration der Klasse

Als Beispiel soll eine Klasse `Image` definiert werden, die ein digitales Bild repräsentiert. Als Instanzvariablen enthält sie Felder für die eigentlichen Bilddaten (ein Array vom Typ `int`), sowie für die Breite und Höhe des Bildes. Die Memberfunktion `show`() stellt das Bild dar.

Die Definition der Klasse wird in der Regel aufgeteilt in Headerdatei und Implementierungsdatei. Anders als in Java gibt es keine Vorschriften für die Dateinamen. Es folgt die Headerdatei dieser Klasse:

```
#ifndef IMAGE_H
#define IMAGE_H

#include <vector>
class Image
{
public:
               Image(int width, int height);
               ~Image();
               void show();
private:
               int* data;
               int width;
               int height;
};
#endif
```

- **Wichtig:** die Klassendefinition endet mit einem **Semikolon**! Dies ist ein häufiger schwer zu findender Anfängerfehler.
- in C++ werden die Zugangsmodifier (`public`, `protected`, `private`) als Label mit Doppelpunkt vor einen Block mit Funktionen gesetzt

- eine Headerdatei sollte **immer** Include Guards der folgenden Form enthalten:

```
#ifndef DATEINAME_H
#define DATEINAME_H
...
#endif
```

- die Headerdatei enthält nur die Signatur der Methoden, gefolgt von einem Semikolon.
- die Headerdatei kann weitere Headerdateien einbinden
- eine Headerdatei **soll keine** `using namespace …;` Anweisung enthalten. Stattdessen wird der vollständige Name verwendet, z.B. `std::vector`.

Die Headerdatei ist die sichtbare Schnittstelle für Code, der diese Klasse verwenden soll. Bei Softwarebibliotheken ist sie häufig die wichtigste Dokumentation der enthaltenen Klassen.

5.2 Implementierungsdatei: Definition der Klasse

Die Implementierungsdatei ist eine eigene Quelldatei, in der die Klasse definiert wird. Sie enthält die Definitionen von Konstruktor, Destruktor und den Memberfunktionen. Dabei wird jeweils mit Doppelpunkt der Name der Klasse vorangestellt, z.B. für `show`()

```
void Image::show() {…}
```

Die Definition in diesem Beispiel beginnt mit dem Einbinden der Headerdatei und der Auswahl eventueller Namespaces:

```
#include "image.h"
using  namespace std;
```

Es folgt die Definition des **Konstruktors**. Hier werden die Instanzvariablen initialisiert und das Array `data` wird allokiert und initialisiert.

```
Image::Image(int w, int h)
: width(w)
, height(h)
{
   data = new int[width * height];
   for (int i = 0; i < width * height; i++){
      data[i] = 0;
   }
}
```

Hier wurden die Instanzvariablen `width` und `height` über die Initialisierungsliste initialisiert. Dieser Weg sollte bevorzugt werden und ist bei eingebundenen Member-Klassen auch der einzig mögliche.

In der Regel haben C++ Klassen einen **Destruktor.** Er wird aufgerufen, wenn ein Objekt der Klasse mit delete zerstört wird. Er soll z.B. Speicher freigeben oder eventuell erzeugte Objekte zerstören.

```
Image::~Image(void){
   delete[] data;
}
```

Und schließlich die Implementierung der Methode `show`() (hier nur als Dummy-Code):

```
void Image::show(){
   for (int i = 0; i < data.size(); i++){
      cout << data[i];
   }
}
```

5.3 Verwendung einer Klasse

Die oben beispielhaft definierte Klasse Image kann auf zwei Arten verwendet werden:

- eine Variable ist Zeiger auf ein Objekt der Klasse Image
- eine Variable ist ein Objekt der Klasse Image

Beide Varianten sind gebräuchlich. *Qt* und die *VST-SDK* verwenden hauptsächlich die erste Variante, während *opencv* die zweite Variante verwendet.

Zeiger auf Objekte

Folgendes Codefragment verwendet die Klasse Image:

```
int main(){
   Image *image;
   image = new Image(10, 10);
   image->show();
   delete image;
}
```

Zunächst wird ein Zeiger auf ein Objekt der Klasse Image deklariert:

```
Image *image
```

Dieser wird dann durch Aufruf des Konstruktors allokiert und initialisiert. Dabei wird der Zeigervariablen die Adresse des Image-Objekts zugewiesen:

```
image = new Image(10, 10)
```

Zum Aufruf der Methode show() wird der Pfeil-Operator verwendet:

```
image->show()
```

im Detail: der Pfeiloperator dereferenziert einen Zeiger und wendet dann den Punkt-Operator an, d.h. verwendet eine Memberfunktion oder Membervariable. Folgende Codezeile sind äquivalent

```
(*zeigerAufObjekt).methode();

zeigerAufObjekt->methode();
```

Mit `delete` wird der Destruktor aufgerufen. Aber **Vorsicht**: die Variable `image` zeigt danach noch immer auf eine gültige Adresse, die aber nicht mehr ein Image-Objekt repräsentiert. Sie darf nicht mehr verwendet werden!

```
delete image;
```

Grundregel für guten Code: Eine Klasse, Methode, Modul, das ein Objekt mit `new` erzeugt, soll dieses auch an entsprechender Stelle mit `delete` wieder zerstören.

Objektvariablen

Folgender Code verwendet direkt ein Image-Objekt, er ist zu obigem Beispiel äquivalent.

```
Image image(10, 10);
image.show();
```

Die **Initialisierung**-Parameter werden in Klammern hinter die Variable geschrieben – (ohne `new`). Für den **Aufruf** einer Memberfunktion wird der Punkt-Operator verwendet. Der **Destruktor** wird automatisch dann aufgerufen, wenn die Variable `image` ihre Gültigkeit verliert.

5.4 Objekte als Member einer Klasse

Klassen können Objekte anderer Klassen sowohl direkt als Objekte oder als Zeiger auf Objekte enthalten.

Variante 1: Member ist Zeiger auf Objekt

Als Beispiel soll eine Klasse `VideoEngine` ein Objekt der Klasse `Image` als Instanzvariable enthalten. Im folgenden Beispiel ist die Instanzvariable `image` ein Zeiger auf ein Image-Objekt.

```
class VideoEngine{
...
private:
   Image *image;
};
```

Sie muss an geeigneter Stelle initialisiert werden, zumindest wird sie im Konstruktor auf 0 gesetzt. (0 kann in C++ einem Objektzeiger zugewiesen werden und entspricht dem `null`-Objekt von Java).

```
VideoEngine::VideoEngine()
: image(0)
{…}
```

Später, wenn Breite und Höhe des `Image`-Objekts bekannt sind, kann initialisiert werden, z.B. in einer Methode `open`()

```
void VideoEngine::open(){
   image = new Image(width, height);
}
```

In jedem Falle muss der Destruktor das Objekt zerstören:

```
VideoEngine::~VideoEngine(){
   delete image;
}
```

Hinweis: die Anweisung `delete` ist auch dann erlaubt, wenn `image` den Wert 0 hat.

Variante 2: Member ist Objekt

In diesem Alternativ-Beispiel soll die Instanzvariable image ein Objekt von Image sein.

```
class VideoEngine{
...
private:
   Image image;
};
```

Dieses Objekt **muss** im Konstruktor initialisiert werden:

```
VideoEngine::VideoEngine(int width, int height)
:image(width, height)
{...}
```

Das Objekt **darf nicht** zerstört werden.

Bemerkung: Diese Variante ist insofern sicherer, als der Aufruf des Destruktors automatisch geschieht und nicht vergessen werden kann. Sie hat allerdings den Nachteil, dass das Objekt im Konstruktor initialisiert werden muss. In unserem Fall muss also die Breite und Höhe des Image-Objekts bekannt sein.

5.5 const Member

C++ bietet optional die Möglichkeit, Member einer Klasse *read-only* zu definieren. Sie sind dann von außen nicht veränderbar. Dies kann die Code-Qualität verbessern.

const Membervariablen

Membervariablen, die sich nach ihrer Initialisierung nicht verändern, sollten `const` deklariert werden. Zum Beispiel sollen die Breite und die Höhe eines Image-Objekts nicht nachträglich veränderbar sein, z.B. in `Image.h`

```
...
class Image {
...
private:
   const int width;
   const int height;
}
```

const-Membervariablen **müssen** im Konstruktor mit der Initialisierungsliste initialisiert werden, z.B. in `Image.cpp`:

```
Image::Image(int w, int h)
: width(w)
, height(h)
{...}
```

const Memberfunktionen

Um die Breite und die Höhe eines Image-Objekts abzufragen, kann man Getter-Funktionen definieren. Getter-Funktionen werden in C++ als **const-Memberfunktionen** definiert:

```
Image.h
...
class Image {
public:
   int getWidth() const;
   int getHeight() const;
...
}
Image.cpp
int Image::getWidth() const {
   return width;
}
```

Das Schlüsselwort `const` hinter der Methode `getWidth`() bedeutet hier, dass der Aufruf von `getWidth`() den Zustand des Objekts **nicht verändert**. In unserem Beispiel ist es nicht sinnvoll, die Breite oder die Höhe eines Bildes *nachträglich* zu verändern.

Methoden, die den Zustand ihres Objekts nicht verändern, sollten `const` deklariert werden!

5.6 struct

Das Schlüsselwort `struct` ermöglicht in C und in C++ die einfache Definition von Datenstrukturen. Kurz gesagt ist ein `struct`-Objekt eine Klasse mit `public` Membervariablen. So kann eine Klasse Point wie folgt definiert werden:

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

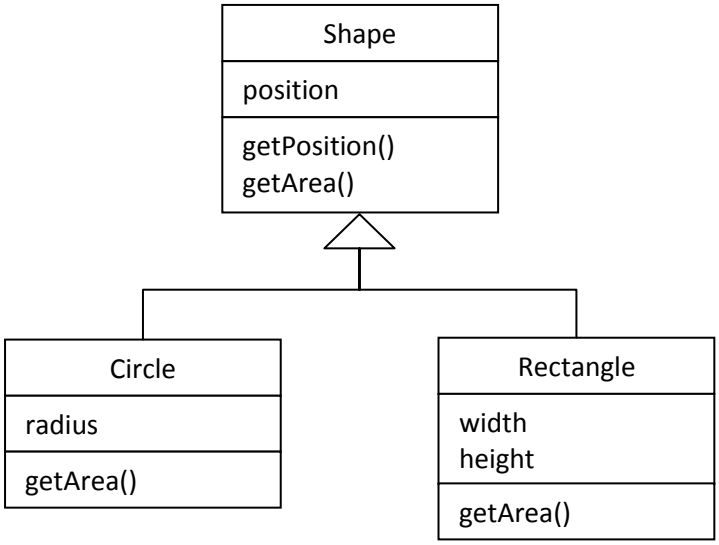
Die Verwendung von `struct`-Objekten ist identisch mit der von Klassen.

6 Vererbung

C++ unterstützt wie Java Vererbung. Dabei gibt es für die meisten Konzepte der Vererbung in Java auch entsprechende Konzepte in C++. C++ verwendet jedoch eine andere Syntax.

6.1 Grundsyntax

Als Beispiel verwenden wir das bekannte Beispiel aus der Java-Vorlesung:



Die Klasse `Shape` ist **Basisklasse** eines Vektor-Zeichenprogramms. Sie kapselt die Variable `position`. Die **abgeleiteten Klassen** `Circle` und `Rectangle` erben die Member `position` und `getPosition`(), überschreiben `getArea`() und fügen neue Member hinzu.

Die **Basisklasse** Shape hat folgende Deklaration

```
class Shape {
public:
   Shape(Point position);
   virtual ~Shape();
   virtual Point getPosition() const;
   virtual float getArea() const = 0;
protected:
   Point position;
};
```

Regeln für Basisklassen

1. Alle Methoden, die überschrieben werden sollen, werden `virtual` deklariert.
2. Basisklassen sollen **immer** einen **`virtual` deklarierten Destruktor** haben
3. rein virtuelle Methoden (in Java: abstrakte Methoden) werden `virtual` deklariert und `=0` gesetzt.

In diesem Beispiel ist die Methode `getArea`() rein virtuell, weil keine Aussage über die Fläche eines Shape-Objekts gemacht werden kann. Die Methode `getPosition`() wurde ϖιρτυαλ deklariert, um zu ermöglichen, dass abgeleitete Klassen diese Methode überschreiben.

Die **abgeleitete Klasse** Rectangle hat folgende Deklaration

```
class Rectangle: public Shape {
public:
   Rectangle(Position position, int width, int height);
   float getArea();
private:
   int width;
```

```
    int height;
};
```

Zu beachten ist nun die Implementation des Konstruktors. Hier wird zunächst der Konstruktor der Basisklasse in der Initialisierungsliste aufgerufen:

```
Rectangle::Rectangle(Position position, int width, int height)
: Shape(position), width(width), height(height)
{}
```

6.2 Rein virtuelle Klassen = Interfaces

In C++ existieren nicht die Schlüsselworte abstract bzw. interface. Das Konzept eines Interface wird in C++ mit einer Klasse realisiert, die nur rein virtuelle Funktionen enthält. Eine solche Klasse heißt **rein virtuelle Klasse**.

Als Beispiel soll eine mögliche Implementation des Observer-Entwurfsmusters gezeigt werden. Die rein virtuelle Klasse Observer definiert eine rein virtuelle Methode update():

```
class Observer{
public:
    virtual ~Observer(){}
    virtual void update() = 0;
};
```

Die Klasse hat einen virtual deklarierten Destruktor, der in der Headerdatei leer implementiert wird. Alle Methoden einer rein virtuellen Klasse sind rein virtuell. Eine solche Klasse hat keine Membervariablen.

6.3 Mehrfachvererbung

Java unterstützt keine Mehrfachvererbung, jedoch Mehrfachimplementierung. C++ unterscheidet nicht zwischen Vererbung und Implementierung, d.h. C++ Klassen können von mehreren Klassen erben. Da dies jedoch sehr fehlerträchtig ist, sollte es möglichst nie verwendet werden.

7 Sicheres Programmieren

Für Java-Programmierer, die zu C++ wechseln, stellen sich zunächst einige Fallstricke und schwer zu findende Fehlerquellen. Die hier erläuterten Punkte zeigen einige potentielle Fehlerquellen bei der C++-Programmierung. Fehlerhafter Java-Code kann gültiger C++ Code sein. Um dies zu verhindern sollte man

- Code so schreiben, dass er auch vom Java-Compiler "genehmigt" würde
- Warning-Level des C++ Compilers möglichst hoch setzen

Folgende **sehr empfehlenswerten** Bücher beschreiben viele C++-Programmiertechniken, die zu sicherem C++-Code führen:

- Scott Meyer, *Effective C++, 55 Specific Ways To Improve Your Programs And Designs*
- Scott Meyer, *More Effective C++, 35 New Ways to Improve Your Programs and Designs*
- Herb Sutter, Andrei Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*
- Scott Meyer, *Effective Stl, 50 Specific Ways To Improve The Use Of The Standard Template Library*

Fehler	Ursache	Abhilfe
Variablen		

nicht initialisierte lokale Variablen	der C++ Compiler zwingt nicht zur Initialisierung von Variablen. Im folgenden Beispiel wird die Variable a verwendet, ohne dass sie initialisiert wurde. Das Ergebnis ist unvorhersagbar:
	`int a;`
	`int b = a * a;`
		Variablen sollten möglichst immer initialisiert werden.

nicht initialisierte Membervariablen	Membervariablen werden in C++ nicht automatisch initialisiert. Dies wird auch nicht vom Compiler überprüft.	Der Konstruktor muss alle Membervariablen initialisieren. Wenn kein Wert gegeben ist, können sie mit 0 initialisiert werden.
Bedingungen		

Zuweisung statt Vergleichsoperator	wenn der C++ Compiler in einer Bedingung, z.B. if() eine Zuweisung erhält, prüft er implizit, ob der Wert ungleich 0 ist. Ein häufiger Fehler ist folgender Code:
	`if (x = 0) …`
	Hier wird x der Wert 0 zugewiesen. Anschließend prüft der Compiler, ob der Ausdruck ungleich 0 ist. In diesem Fall wird der if-Zweig nie ausgeführt.
	Der Fehler kann sich auch in einer Schleife verstecken, hier wird die Schleife nie ausgeführt:
	`for(int x = 100; x = 0; x--)`
		Guter Stil verlässt sich nicht auf die implizite Überprüfung, ob ein Wert ungleich 0 ist.

Arrays		
nicht initialisierte Array-Elemente	Nach der Allokation haben die	Nach der Allokation sollen

	Elemente eines Arrays einen undefinierten Wert. Im folgenden Beispiel wird ein undefinierter Wert ausgegeben:
	`int * data = new int [5];`
	`cout << data[0];`
		die Werte immer initialisiert werden.
		
		Alternativ kann die Klasse vector verwendet werden.

Zugriff auf ungültige Array-Elemente	C++ macht keine Überprüfung des Array-Index. So ist es möglich, auf beliebige Speicheradressen zuzugreifen:
	`int *data = new int[10];`
	`...`
	`data[1000] = -1;`
	Dieser Fehler kann sich auch in falschen Schleifen „verstecken“:
	`int *data = new int[10];`
	`for (int i = 0; i <= 10; i++){`
	`  data[i] = 0;`
	`}`
		Hier helfen oft nur Debugging-Tools.
		
		Alternativ kann die Klasse vector verwendet werden.

delete statt delete[] bei Arrays	der C++ Compiler erwartet die eckigen Klammern beim Zerstören von Arrays.	Debugging-Tools helfen Speicherlecks zu finden

fehlendes delete[]	C++ hat keine Garbage Collection wie Java. Wenn delete[] vergessen wird, bleibt der Speicher reserviert. Beispiel: hier wird 100 mal ein Array von 10000 int – Werten allokiert, aber nirgends freigegeben:
	`int *data;`
	`for (int i = 0; i < 100, i++){`
	`  data = new int[10000];`
	`}`
		Debugging-Tools helfen Speicherlecks zu finden

mehrfaches delete[]	delete prüft nicht, ob ein Speicherbereich bereits freigegeben wurde. Mehrfaches delete einer Adresse führt zu Speicherfehlern.	Debugging-Tools helfen Speicherlecks zu finden
Klassen		

Übergabe von Objekten	Unterschied zwischen *call by reference* bzw. *call by pointer* und *call by value.*
	Beispielsweise wird hier eine Referenz auf ein string-Objekt übergeben (es wird lediglich die Adresse kopiert)
	`void open(string & file)…`
	Im folgenden Beispiel wird das gesamte string-Objekt kopiert:
	`void open(string file) …`
	In **Java** werden elementare	

	Datentypen *call by value* übergeben und Objekte als *call by reference.*	
Methoden sollen möglichst keine Objekte erzeugen	Methoden, die Objekte erzeugen, ohne sie zu zerstören sind sehr fehlerträchtig. In **Java** ist dies dank *garbage collection* kein Problem.	
nicht virtuelle Methoden	C++ erlaubt auch die nicht-virtuelle Vererbung. Dies führt aber zu unerwünschten Nebeneffekten und ermöglicht keinen Polymorphismus. Dieser Fehler tritt auf, wenn virtual vergessen wird.	alle überschreibbaren Methoden müssen virtual deklariert werden
fehlender virtueller Destruktor einer Basisklasse	ist der Destruktor einer Basisklasse nicht virtuell kann es zu Speicherfehlern kommen.	Basisklassen haben immer einen rein virtuellen Destruktor

8 Ausblick

Dieses Skript ist eine Anleitung für die ersten Schritte in der Programmiersprache C++. Hier seien weitere fortgeschrittene Themen erwähnt, die ein guter C++ Programmierer beherrschen sollte.

8.1 Operator überladen

C++ erlaubt es, das Klassen Operatoren (arithmetische, Zuweisung, Typecast usw.) überschreiben. Dies tut z.B. die Klasse Mat von *opencv* und ermöglicht folgenden Code, der ein Differenzbild berechnet:

```
Mat image1;
Mat image2;
…
Mat differenceImage = image1 – image2;
```

8.2 Big Three

Im Kapitel zu Klassen wurde der virtuelle Destruktor besprochen. Die *rule of big three* besagt, dass immer folgende drei Methoden zusammen definiert sein sollen:

- Destruktor
- Copy Constructor
- Zuweisungsoperator

Eine Erläuterung lässt sich z.B. hier finden: http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400

8.3 Templates

Templates sind parametrisierte Datentypen, vergleichbar den Generics in Java. Zum Beispiel ist die Klasse vector eine Template-Klasse:

```
vector<int> tabelle(10);
vector<Image>images;
```

C++ hat auch die Möglichkeit, Funktionen als Template-Funktionen zu parametrisieren. Ein Beispiel ist die Methode at() der Klasse Mat von *opencv:*

```
class Mat{
   …
   template<typename _Tp> _Tp& at(int y, int x);
};
```

Die komplizierte Syntax wird bei der Verwendung dieser Methode deutlich. Folgender Aufruf fragt den Wert des Pixels an den Koordinaten x,y eines Farbbildes (Pixel haben Werte vom Typ `Vec3b`) ab.

```
Mat img;
Vec3b pixel = img.at<Vec3b>(y, x);
```

Für die Pixelabfrage eines Graustufenbildes (Pixel haben Werte vom Typ uchar) wird folgende Anweisung verwendet:

```
Mat grayImage;
uchar pixel = grayImage.at<uchar>(y,x);
```

8.4 Standard Template Library (STL)

C++ hat mit der Standard Template Library eine sehr mächtige Bibliothek von Datenstrukturen, Algorithmen, und Funktionen. Beispiele sind die Klassen `vector, string, iostream.` Bei der Implementierung wurde auf Performance geachtet. So kann die Klasse vector in der Regel immer anstelle eines C-Arrays verwendet werden.

8.5 Smart Pointer

Neuere C++ Versionen bieten sogenannte *smart pointer.* Diese Template-Klassen bieten eine automatische Speicherfreigabe, wenn ein Objekt nicht mehr benötigt wird. Dies erlaubt sehr komfortable Resourcenverwaltung und vermeidet Speicherleck-Fehler.

8.6 boost Library

Die *boost*-Library ist eine *open source* Bibliothek, die als Erweiterung der STL gedacht ist. Einige Klassen der *boost*-Library sind mittlerweile C++ Standard geworden.