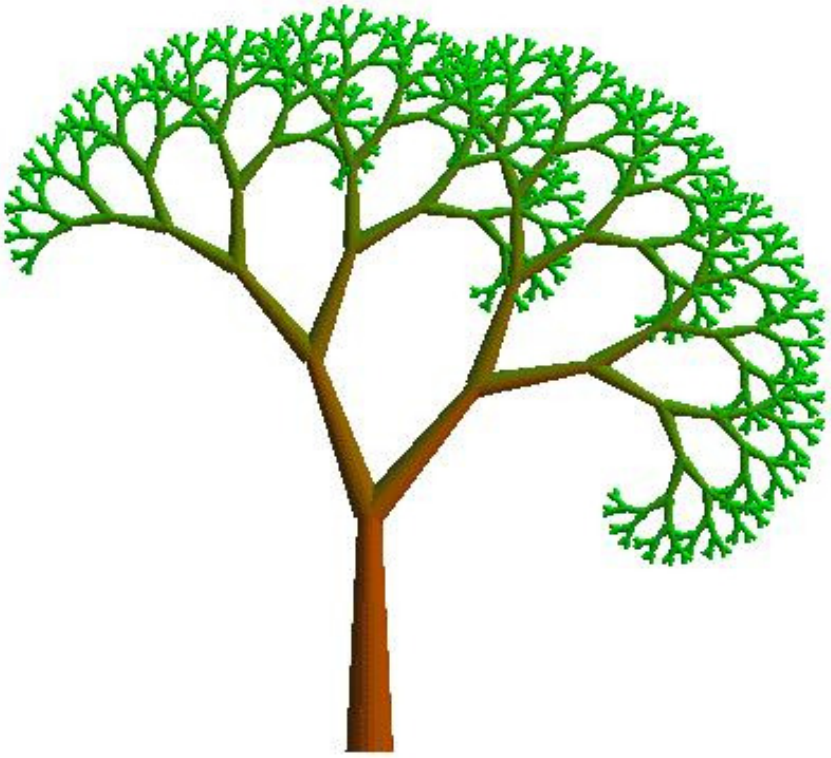
C++ Study Guide

Einführung in die Programmiersprache C++



by David Herzig

Version 2.4, Februar 2010

Inhaltsverzeichnis

0.1 Vorwort

Im folgenden Script habe ich meine Erfahrungen in der Informatik, welche ich aus mehrjähriger Projekt- und Schulungsarbeit gewonnen habe festgehalten.

Es ist hauptsächlich für die Studenten der FHNW und der KTSI geschrieben worden. Zusammen mit dem Unterricht soll es Ihnen zu einem erfolgreichen Abschluss auf diesem Gebiet verhelfen.

Da ich weder Gott noch dessen Vikar auf der Erde bin, kommen auch in diesem Script Fehler vor. Daher bin ich allen aufmerksamen Lesern dankbar, mir eventuelle Unklarheiten oder Fehler mitzuteilen.
`dave@kitware.net`

0.2 Copyright

Kapitel 1

Einführung

Wenn Wissenschaftler etwas als möglich darstellen, liegt er fast sicher richtig. Wenn er etwas als unmöglich hinstellt, liegt er sicher wahrscheinlich falsch. Die einzige Möglichkeit, die Grenzen des Möglichen zu erkunden, liegt darin, sich dicht an ihm vorbei ins Unmögliche zu wagen. Jede einigermassen moderne Technik ist von Magie nicht zu unterscheiden.
Gesetze der Technik von Arthur C. Clarke

1.1 Präambel

Als Ingenieure/Techniker werden Sie ihr tägliches Brot mit der Lösung einer bestimmten Klasse von Problemen verdienen, die ganz allgemein als mathematische Probleme bezeichnet werden können. Sie werden sich häufig fragen, ob für die Lösung spezielle Methoden vorhanden seien. Wenn damit ein Kochrezept gemeint ist, kann die Antwort nur *Nein* lauten. Wenn Sie dagegen damit die Existenz von heuristischen Wegen postulieren wollen, die der Lösung auf systematischer Weise näher kommen, dann lautet die Antwort *Ja*.

Heuristik ist die Untersuchung der Mittel und Methoden des Aufgabenlösens. Die heuristischen Verfahren sind der praktischen Arbeit abgeschaut, der Art und Weise, in der Mathematik als Prozess stattfindet und von Menschen gemacht wird (zu unterscheiden von der standardisierten und zumeist stark verkürzten Mitteilung fertiger Mathematik in Ergebnisform). Zumindest diejenigen, die selber gerne mathematische Dinge tun, erhalten damit die Chance, ihre Problemlösefähigkeiten zu steigern.

Ihre Hauptwerkzeuge sind:

1. Induktion
2. Verallgemeinerung
3. Spezialisierung
4. Analogie

1.1.1 Beispiel

Induktion fängt meistens mit einer Beobachtung an. Zum Bespiel:

$$3 + 7 = 10$$
$$3 + 17 = 20$$
$$13 + 17 = 30$$

Sehen Sie eine Gesetzmässigkeit in diesen Operationen?

1. Vermutung: Jede gerade Zahl, die grösser als 4 ist, ist die Summe von zwei ungeraden Primzahlen. (Warum muss die Zahl grösser als 4 sein?) Experiment:

$$8 = 3 + 5$$
$$10 = 3 + 7 = 5 + 5$$
$$12 = 5 + 7$$
$$14 = 3 + 11 = 7 + 7$$
$$16 = 3 + 13 = 5 + 11$$

2. Vermutung: Jede gerade Zahl, die weder selbst eine Primzahl noch das Quadrat einer Primzahl ist, ist die Summe von zwei ungeraden Primzahlen.

Dies ist eine gewaltige Verallgemeinerung! Können Sie diese Behauptung beweisen? Wenn nein, muss Ihr Ego nicht zuviel darunter leiden: Die besten Köpfe der Menschheit versuchen nämlich seit 200 Jahren vergebens diese Vermutung zu beweisen. (Dieses Problem ist in der Geschichte der Mathematik als Goldbachsche Vermutung eingegangen.)

Bemerkung:
Wie schon Sherlock Holmes merkte, es gibt gewaltige Unterschiede zwischen Sehen und Beobachten! Die meisten Menschen sehen viel, aber beobachten wenig. Beobachten verlangt Neugier und die zielstrebige Suche nach logischen Zusammenhängen.

1.1.2 Die Gefahren der Induktion

Auszug aus: G.Polya: Mathematik und plausibles Schliessen, Band 1, S.32, Birkhäuser, Basel 1988.

Der Logiker, der Mathematiker, der Physiker und der Ingenieur. "Seht euch doch diese Mathematiker an " sagt der Logiker, "er bemerkt, dass die ersten neunundzwanzig Zahlen kleiner als hundert sind und schliesst daraus auf Grund von etwas, das er Induktion nennt, dass alle Zahlen kleiner als hundert sind."
"Ein Physiker glaubt", sagt der Mathematiker, "dass 60 durch alle Zahlen teilbar ist. Er bemerkt, dass 60 durch 1, 2, 3, 4, 5 und 6 teilbar ist. Er unterscheidet noch ein paar Fälle wie 10, 20 und 30, die, wie er sagt, aufs Geratewohl herausgegriffen sind. Da 60 auch durch diese teilbar ist, betrachtet er seine Vermutung als hinreichend durch den experimentellen Befund bestätigt."
"Ja, aber seht Euch doch die Ingenieure an", sagt der Physiker. "Ein Ingenieur hatte den Verdacht, dass alle ungeraden Zahlen Primzahlen sind. Jedenfalls so argumentierte er, kann 1 als Primzahl betrachtet werden. Dann kommen 3, 5 und 7, alle zweifellos Primzahlen. Dann kommt 9; ein peinlicher Fall, wir scheinen hier keine Primzahl zu haben. Aber 11 und 13 sind unbestreitbar Primzahlen." "Auf die 9 zurückkommend", sagte er, "schliesse ich, dass 9 ein Fehler im Experiment sein muss."

Es liegt nur allzu sehr auf der Hand, dass Induktion zu Irrtum führen kann. Um so bemerkenswerter ist es, da die Irrtumsmöglichkeiten so überwältigend erscheinen, dass Induktion manchmal zur Wahrheit führt. Sollen wir mit der Untersuchung der auf der Hand liegenden Fälle beginnen, in denen Induktion zu Erfolg führt?
Es ist begreiflicherweise viel reizvoller, Forschungen über Edelsteine anzustellen als über Kieselsteine. Auch sind die Mineralogen viel eher durch Edelsteine als durch Kieselsteine zu der wunderbaren Wissenschaft der Kristallographie geführt worden.

1.2 Eine Sammlung hübscher Probleme

1. Die Verwaltung einer grossen technischen Zeitung empfängt mehrere tausende Briefe pro Tag. Die Briefe gehöhren mehreren Kategorien an: Zahlungen, neue Abonnenten, Reklamation, Werbeanfragen, u.s.w. Diese Briefflut muss sortiert werden, bevor der entsprechende Sachbearbeiter seine Arbeit anfangen kann. Besprechen Sie Lösungen, die rasch zu implementieren und zusätzlich auch günstig sind.

2. Ein Bär läuft ab einem Punkt P 1km Richtung Süden. Dann wechselt er die Richtung und läuft wieder 1km nach Osten. Er biegt links ab und läuft für einen weiteren Kilometer Richtung Norden und kommt genau im Punkt P an. Was für eine Farbe hat der Bär?

   Lösung:

   - F: Welche ist die Unbekannte des Problems?
   - A: Die Farbe des Bären. Wieviele Bärenfarben gibt es? Mindestens braun und weiss.
   - F: Welche Daten stehen Ihnen zur Verfügung?

- A: Eine geometrische Lage und die Tatsache, dass der Bär nach einem 3km langen Spaziergang wieder im Punkt P angelangt ist.
- F: Welche Bedingung muss erfüllt sein?
- A: Eben nach 3km sitzt der Bär wieder am Startpunkt P
- F: Kennen Sie eine geometrische Fläche, die diese Bedingung erfüllt?
- A: Ja, die Oberfläche einer Kugel. Die Erdoberfläche ist annähernd eine Kugeloberfläche.

AHA-Erlebnis

- Wo leben weisse Bären?
- Nur am Nordpol. Diese Tatsache (=Datum) ist nicht im Problem enthalten. Die Rolle der Erfahrung und der Vorkenntnisse ist bei der Lösung von Problemen entscheidend. (Frei gestaltet nach G.Polya, *How to solve it, op. cit.*)

3. (T.A.Edison) Berechnen Sie das Volumen einer Glühbirne in 60 Sekunden.

4. Berechnen Sie das gesamte Volumen, das wegfallen würde, wenn Sie ein Loch mit der Höhe von 6cm in eine Kugel beliebiges Radius bohren würden.

5. Die Wasserquellen, die ein Tier, dass in der Wüste lebt, benutzen kann, sind weit entfernt. Wie hängt deren Entfernung von der Grösse L des Tieres ab?

6. Wie hängt die Geschwindigkeit eines Tieres von seiner Grösse L ab, wenn es geradeaus bzw. bergaufwärts läuft?

7. Wie hängt die Sprunghöhe eines Tieres von seiner Grösse L ab?

8. Eine Kaffeebüchse enthält eine nicht näher definierte Menge von weissen und schwarzen Kaffeebohnen. Wenden Sie folgenden Algorithmus an, bis nur eine Kaffeebohne übrig bleibt:

   - Picken Sie zwei Kaffeebohnen aus der Büchse heraus.
   - Wenn die Farbe gleich ist, werfen Sie beide Kaffeebohnen weg und werfen eine schwarze Kaffeebohne in die Büchse (es gibt genügend schwarze Kaffeebohnen).
   - Wenn die Farbe ungleich ist, werfen Sie die schwarze Kaffeebohne weg und werfen die weisse in die Büchse zurück.

   Welche Farbe hat die letzte Kaffeebohne in Abhängigkeit von der Anzahl weisser und schwarzer Kaffeebohnen, die ursprünglich in der Kaffeebüchse liegen?

1.3 Zeit

Die nächste Tabelle versucht Ihnen ein konkretes Bild der verschiedenen Grössenordnungen, die in der Informatik vorkommen, zu geben. Merken Sie bitte, dass π Sekunden ungefähr ein Nanojahrhundert sind.

Meters per second	*Example*
10^{-11}	Stalacites growing
10^{-10}	Slow continent drifting
10^{-9}	Fingernails growing
10^{-8}	Hair growing
10^{-7}	Weeds growing
10^{-6}	Glacier
10^{-5}	Minute hand of a watch
10^{-4}	Gastro-intestinal tract
10^{-3}	Snail
10^{-2}	Ant
10^{-1}	Giat Tortoise
10^{0}	Human walk
10^{1}	Human sprint
10^{2}	Propeller airplane
10^{3}	Fastest jet airplane
10^{4}	Space shuttle
10^{5}	Meteor impacting earth
10^{6}	Earth in galactic orbit
10^{7}	LA to satellite to NY
10^{8}	One-third speed of light

Es gibt viele Probleme, welche auch von einem Computer nicht in akzeptabler Zeit gelöst werden können. Beispiel: Rucksackproblem, Clique, ... Diese Probleme werden Probleme der Klasse NP genannt.

1.4 Was ist ein Algorithmus?

Die Probleme, die wir im früheren Abschnitt gelöst haben, hatten alle eine Gemeinsamkeit: Die Lösung konnte in einer endlichen Anzahl logischer Schritte erreicht werden. Nachdem wir den schwierigen Lösungsweg gefunden hatten, waren die Einzelschritte, die zum Resultat führten, einfach, sogar trivial einfach. Diese Eigenschaft wird von allen Algorithmen in der Informatik geteilt: Einen guten Algorithmus zu finden ist eine höllisch schwierige Aufgabe; die einzelnen Anweisungen eines guten Algorithmus auszuführen, eine denkbar triviale Angelegenheit, die einer Maschine delegiert werden darf.

Einführungskapitel nach *C.U.Nicola, Mikrorechner, FH-Aargau*

1.5 Zukunftsprognosen

Während dem Schreiben dieses Scriptes lass ich das Buch "Homo Sapiens" von Ray Kurzweil. Dort traf ich auf ein paar interessante Zukunftsprognosen im Bereich der Informatik.

640000 Bytes Speicherkapazität solltem jedem genügen.

Bill Gates, 1981

Flugzeuge haben keinen militärischen Nutzen.

Professor Marshal Foch, 1912

Computer der Zukunft dürfen nicht mehr als 1.5 Tonnen wiegen.

Popular Mechanics, 1949

Das Telefon hat zu viele Mängel, als dass es ernsthaft als Kommunikationsmittel in Betracht kommen könnte.

Manager der Western Union, 1876

Es gibt keinen Grund, warum Menschen zu Hause einen Computer haben sollten.

Ken Olson, 1977

Kapitel 2

Mathematische Grundlagen

A little knowledge is a dangerous thing.
J.Weizenbaum

2.1 Zahlensysteme

2.1.1 Umwandlung Dezimalsystem nach ...

Umwandlung der Zahl x vom Dezimalsystem in ein Zahlensystem mit Basis b.

$x : b = a_0 \quad REST \quad b_0$
$a_0 : b = a_1 \quad REST \quad b_1$
$a_1 : b = a_2 \quad REST \quad b_2$
$\cdots$
$a_{n-1} : b = 0 \quad REST \quad b_n$

Lösung: $b_n \cdots b_2 b_1 b_0$

2.1.2 Umwandlung ... nach Dezimalsystem

Umwandlung der Zahl x von einem Zahlensystem mit Basis b in das Dezimalsystem.

$$Dezimalwert = Ziffer\, n \cdot b^n + \cdots + Ziffer\, 2 \cdot b^2 + Ziffer\, 1 \cdot b^1 + Ziffer\, 0 \cdot b^0$$

Ziffer n ist die Ziffer ganz links.

2.1.3 Umwanldung Binärsystem - Hexadezimalsystem

Die Ziffern der binären Zahl werden von rechts in 4er Gruppen eingeteilt:

$1111001001101 \rightarrow 1 \quad 1110 \quad 0100 \quad 1101$

Jede Gruppe wird nun in den Dezimalwert umgerechnet.

$$\underbrace{1}_{1} \quad \underbrace{1110}_{E} \quad \underbrace{0100}_{4} \quad \underbrace{1101}_{D}$$

Lösung: $1E4D$

Diese Umwandlung funktioniert auch in umgekehrter Richtung, d.h. für die Umwandlung vom Hexadezimalsystem ins Binärsystem.

2.2 Zahlenmengen

- $\mathcal{N} = 1, 2, 3, ...$
  Menge der natürlichen Zahlen
- $\mathcal{N}_0 = 0, 1, 2, ...$
  Menge der natürlichen Zahlen inklusive der Zahl 0
- $\mathcal{R}$
  Menge der reelen Zahlen

2.3 Intervalle

Endliche Intervalle

$a \leq x \leq b$	abgeschlossenes Intervall
$a \leq x < b$	halboffenes Intervall
$a < x \leq b$	halboffenes Intervall
$a < x < b$	offenes Intervall

Unendliche Intervalle

$a \leq x < \infty$
$a < x < \infty$
$-\infty < x \leq b$
$-\infty < x < b$

2.4 Logarithmen

$$x = log_a r$$

r: Numerus ($r > 0$)
a: Basis ($a > 0; a \neq 1$)

Rechenregeln:

1. $log_a(u \cdot v) = log_a u + log_a v$
2. $log_a(\frac{u}{v}) = log_a u - log_a v$
3. $log_a(u^n) = n \cdot log_a u$
4. $log_a \sqrt[n]{u} = (\frac{1}{n}) \cdot log_a u$

2.5 Fakultät

$$n! = 1 \cdot 2 \cdot 3 ... (n-1) \cdot n \qquad (n \in \mathcal{N}_0)$$

Ergänzung: $0! = 1$

2.6 Summen

$$\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

$$\sum_{k=1}^{n} k^2 = \frac{n \cdot (n+1)(2n+1)}{6}$$

Kapitel 3

Grundlagen C++

We should recognize that the art of programming is the art of organizing complexity.
E.W.Dijkstra

3.1 Struktur eines C++ Programms

Einer der besten Wege eine neue Programmiersprache zu lernen, ist ein erstes Programm zu schreiben. Dieses könnte folgendermassen aussehen:

```
1 // my first program in C++
2
3 #include <iostream>
4 using namespace std;
5
6 int main(){
7   cout << "HELLO WORLD" << endl;
8   return 0;
9 }
```

Beschreibung:

```
1 // my first program in C++
```

Dies ist eine Kommentar Zeile. Alle Zeilen die mit // beginnen sind Kommentare und haben keine Auswirkung auf das Programm. Sie werden genutzt um den Code mit Erklärungen zu ergänzen.

In C++ existieren zwei Möglichkeiten für Kommentare:

```
1 // Dieser Kommentar geht bis ans Ende der Zeile
2 /* Dieser Kommentar endet
3 beim Erscheinen der Zeichenfolge */
```

```
1 #include <iostream>
2 using namespace std;
```

Zeilen, die mit einem # beginnen, sind Anweisungen für den Präprozessor. In diesem Fall wird dem Präprozessor mitgeteilt, das Headerfile iostream in den Code einzubinden. Die in diesem Script am meisten verwendeten Headerfiles sind: iostream, cmath und string.
Wird die Anweisung `using namespace std` *weggelassen, so muss das Kommando cout mit std::cout verwendet werden.*

```
1 int main()
```

Diese Zeile definiert den Startpunkt des Programms. Es spielt keine Rolle wo im Code sie sich befindet. Wird das Programm ausgeführt, so wird an dieser Stelle begonnen. Alle Programme müssen dies besitzen. Mit geschweiften Klammern wird definiert, was alles zum main Teil gehört.

```
1 cout << "Hello World" << endl;
```

Mit dieser Anweisung wird die Zeichenkette Hello World auf den Bildschirm geschrieben. Die Ausgabe Funktion cout ist im Headerfile iostream definiert.

```
1 return 0;
```

Die return Anweisung beendet die main Anweisung. Diese Anweisung benötigt auch den return Wert. Im Beispiel ist es 0. Dies bedeutet, dass das Programm ohne Fehler beendet wurde. Die meisten Programme enden auf diese Art.

3.2 Variablen, Datentypen und Konstanten

3.2.1 Variable

Damit wir bessere Programme als das vorherige Entwickeln können, müssen wir Variablen einführen. Eine Variable ist ein Speicherplatz, in welchem ein Wert gespeichert werden kann. Welche Art von Werten in einer Variablen gespeichert werden kann, wird durch den Datentyp angegeben. Jede Variable muss deklariert werden, bevor sie benutzt werden kann.

```
1 Datatype name;
```

Beispiele:

```
1 float x; // float Variable mit dem Namen x
2 int m,n; // Zwei int Variablen m und n
3 double pi = 3.141, g; //Mit Zuweisung
```

3.2.2 Datentyp

Die folgenden Datentypen existieren in C++:

Datentyp	*Anzahl Bytes*	*Werte*
`char`	1	einzelnes Zeichen
`int`	2	ganze Zahlen
`long int`	4	ganze Zahlen
`short int`	2	ganze Zahlen
`float`	4	reelle Zahlen
`double`	8	reelle Zahlen
`bool`	1	Wahrheitswerte `true` oder `false`

ACHTUNG: Welche Grösse eine Variable tatsächlich benötigt, ist je nach Plattform verschieden. Eine eindeutige Grösse wurde nie festgelegt. Die Werte in der Tabelle müssen nicht unbedingt zutreffen.

ACHTUNG: In C/C++ besitzen Variablen keinen Default Wert.

Neben den oben aufgeführten Integer Typen mit Vorzeichen stehen die folgenden Vorzeichenlosen Integer Typen zur Verfügung:

Datentyp	*Anzahl Bytes*	*Wertebereich*
`unsigned char`	1	0..255
`unsigned int`	2	0..65535
`unsigned long int`	4	$0..2^{32}-1$
`unsigned short int`	2	0..65535

3.2.3 Konstanten

Mit dem Schlüsselwort `const` kann eine Variable so deklariert werden, dass sich ihr Wert nicht ändern kann.

Beipiele:

```
1 const int m = 5;
2 const int n; // Falsche Anweisung
```

3.3 Operatoren

3.3.1 Zuweisung

Mit dem Zuweisungsoperator = können wir Werte zu Variablen zuweisen.

```
1 a = 5;
```

Diese Anweisung speichert den Wert 5 in der Variablen a. Die linke Seite des Zuweisungsoperators muss immer eine Variable sein. Auf der rechten Seite können sich Konstanten, Variablen, Resultate einer Berechnung oder Kombinationen davon befinden.
Eine Zuweisung findet von rechts nach links statt. Eine Zuweisung kann auf der rechten Seite weitere Zuweisungen enthalten:

```
1 a = 2 + (b = 5);
```

Diese Anweisung ist äquivalent mit

```
1 b = 5;
2 a = 2 + b
```

3.3.2 Arithmetische Operatoren

In C++ existieren die folgenden arithmetischen Operatoren:

- `+` Addition
- `-` Subtraktion
- `*` Multiplikation
- `/` Division
- `%` Modulo (nur ganzzahlige Datentypen)
- `+=` `i+=7` ist identisch mit `i=i+7`

- `-=` `i-=7` ist identisch mit `i=i-7`
- `*=` `i*=7` ist identisch mit `i=i*7`
- `/=` `i/=7` ist identisch mit `i=i/7`
- `%=` `i%=7` ist identisch mit `i=i%7` (nur ganzzahlige Datentypen)

3.3.3 Relationale Operatoren

In C++ existieren die folgenden relationalen Operatoren:

- `==` Gleich
- `!=` Ungleich
- `>` Grösser als
- `<` Kleiner als
- `>=` Grösser gleich
- `<=` Kleiner gleich

ACHTUNG: Der Operator = ist nicht identisch mit ==! Häufig wird ein Vergleich als Zuweisung programmiert.

3.3.4 Inkrement und Dekrement

In C++ existieren ein Inkremetoperator `++` und ein Dekrementoperator `--`.

`i++` ist äquivalent mit `i = i + 1`
`i--` ist äquivalent mit `i = i - 1`

Der Inkrement- und Dekrementoperator können vor (präfix) oder nach einer (suffix) Variablen verwendet werden.

Präfix:

```
1 b = 3;
2 a = ++b; // a=4, b=4
```

Suffix:

```
1 b = 3;
2 a = b++; // a=3, b=4
```

3.3.5 Logische Operatoren

In C++ existieren die folgenden logischen Operatoren:

- `!` Nicht
- `&&` Und
- `||` Oder

a	*b*	*a oder b*	*a und b*
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	false

3.3.6 sizeof

Mit Hilfe des sizeof Operators kann der verwendete Speicherplatz einer Variablen bestimmt werden.

```
1 int a,b;
2
3 // In a werden die Anzahl Bytes welche b
4 // verwendet gespeichert.
5 a = sizeof(b);
6
7 // In a werden die Anzahl Bytes einer
8 // double Variablen gespeichert.
9 a = sizeof(double);
```

Beispiel:

Gegeben ist die Variable `int n;`. Nun soll der Wertebereich dieser Variablen bestimmt werden.

```
1 int m = sizeof(n);
```

Nun sind in der Variablen `m` die Anzahl der Bytes abgespeichert, welche die Variable `n` benötigt.

Wertebereich:

$$\textit{Signed}: \qquad -2^{m\cdot 8-1} \ldots 2^{m\cdot 8-1}-1$$

$$\textit{Unsigned}: \qquad 0 \ldots 2^{m\cdot 8}-1$$

3.3.7 Casting

In manchen Fällen ist es notwendig, dass eine Variable in einen anderen Datentyp umgewandelt werden muss (`int` in `float`). Oder auch das in einer Berechnung eine `int` Variable als `float` Variable verwendet werden soll. Um eine Umwandlung vorzunehmen wird einfach der gewünschte Datentyp vor die Variable in Klammern angegeben.

```
int n = 3, m = 10;
float x;
x = m / n; // x=3, da Integerdivision
x = (float) m / n; // x=3.333, m wurde als float verwendet
x = 3.14159;
n = (int) x; // n=3, die Kommastellen werden abgeschnitten
```

3.4 Ein- und Ausgabe auf der Konsole

3.4.1 Output

Eine Ausgabe erfolgt mit dem Befehl `cout`. `cout` ist ein Stream, in welchen ein Wert geschrieben werden kann, der auf dem Bildschirm erscheinen soll.

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

Mit dieser Anweisung wird die Zeichenkette Hello World in den `cout` Stream geschrieben. Der Einfügungsoperator << kann dabei mehrmals verwendet werden.

```
cout << "C++" << " is so great" << endl;
```

Das `endl` steht für einen Zeilenumbruch.
Um den Wert einer Variablen auszugeben, wird lediglich die Variable nach dem Einfügungsoperator angegeben.

```
cout << a << endl; // Gibt den Inhalt von a aus
cout << "a" << endl; // Gibt das Zeichen a aus
```

3.4.2 Input

Eine Eingabe erfolgt mit dem Befehl `cin`. `cin` ist ein Stream, von welchem gelesen werden kann. Um einen Wert von der Konsole einzulesen, kann folgender Code verwendet werden:

```
// Einlesen einer Variable
int a;
cin >> a;
```

Beispiel:

Entwicklung eines Programmes, welches einen ganzzahligen Wert einliest, und diesen gleich wieder auf dem Bildschirm ausgibt.

```
#include <iostream>
using namespace std;

int main(){
  int i;
  cout << "i= ";
  cin >> i;
  cout << "The value you entered is " << i << endl;
  return 0;
}
```

Beispiel:

Für die Berechnung der Umlaufzeit eines Satelliten auf der Kreisbahn um die Erde kann die folgende Formel verwendet werden:

$$T[sec] = \frac{2 \cdot \pi}{R_E} \cdot \sqrt{\frac{(R_E + h)^3}{g}}$$

g ist die Erdbeschleunigung: $g = 9.80665 m/s^2$
R_E ist der Erdradius: $R_E = 6371 km$
$\pi = 3.14159$

```
#include <iostream>
#include <cmath>
using namespace std;

int main(){
  const double g = 9.80665;
  const double re = 6371;
  const double pi = 3.14159;

  double h;

  cout << "h = ";
  cin >> h;

  double result = 2*pi/(re*1000) *
    sqrt(pow(((re*1000)+h),3)/g);

  // Ausgabe und Umwandlung in Stunden
  cout << result / 3600 << endl;

  return 0;
}
```

3.5 Kommandozeilenparameter

In C++ können dem Programm Parameter über die Kommandozeile mitgegeben werden. Dazu wird die Deklaration der `main` Funktion wiefolgt geändert:

```
1 int main(int argc, char **argv}
```

- Die Variable `argc` beinhaltet die Anzahl Parameter der Kommandozeile einschliesslich des Kommandonamens. Bei dem folgenden Aufruf `prog 1 2 3` wird `argc` auf den Wert 4 gesetzt.
- `argv` ist ein Zeiger welcher die Adresse des ersten Elementes eines Adressvektors enthält.

Wird ein Programm mit Parametern gestartet, so werden diese wiefolgt zugewiesen:

```
testing       1        2        3
argv[0]     argv[1]  argv[2]  argv[3]

argc = 4
```

Der Umgang mit Arrays und Pointers wird zu einem späteren Zeitpunkt erklärt.

3.6 Aufgaben

3.6.1 Mathematische Ausdrücke

Formulieren Sie jeden der folgenden mathematischen Ausdrücke als C-Ausdruck. Verwenden Sie nur die minimale Anzahl Klammern zur Gruppierung von Unterausdrücken. Nutzen Sie Assoziativitäten und Prioritäten der Operatoren soweit möglich.

- $\frac{a}{b} - \frac{x}{y}$
- $\frac{a+b}{a-b} - \frac{x-y}{x+y}$
- $ax^3 + bx^2 + cx + d$
- $\frac{1}{x} + \frac{2}{x^2} + \frac{3}{x^3} + \frac{4}{x^4}$

3.6.2 Grössen von Variablen

Schreiben Sie ein Programm, welches die Speichergrösse von Variablen der folgenden Datentypen ausgibt:

- float
- double
- int
- long
- char

Geben Sie auch die Wertebereiche der einzelnen Datentypen an.

3.6.3 Freier Fall

Beim freien Fall befindet sich ein Körper zunächst in Ruhe und bewegt sich unter dem Einfluss der Erdanziehung aus einer bestimmten Höhe h_0 nach unten. Die Dauer eines Falles kann mit der folgenden Formel berechnet werden:

$$T[s] = \sqrt{\frac{2h_0}{g}}$$

g ist die Erdbeschleuningungskonstante. Ihr Wert beträgt:
$g = 9.807$

Schreiben Sie ein Programm, welches die Höhe h_0 einliest und den Wert der Dauer des Falles ausgibt. Von welcher Höhe muss ein Körper fallen, damit er sich 2s im freien Fall befindet?

3.6.4 Parallelschaltung von Widerständen

Der Gesamtwiderstand von zwei parallel geschalteten Widerständen lässt sich durch die folgende Formel ermitteln:

$$R_{Total} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$$

Schreiben Sie ein Programm, welches die beiden Werte R_1 und R_2 einliest und den Wert R_{Total} berechnet.

Kapitel 4

Kontrollstrukturen und Funktionen

Mit den Händen kannst du
nur eine bestimmte Menge bewältigen,
aber mit deinem Geist unendlich viel.
Kai Seinfeld

4.1 Kontrollstrukturen

C++ enthält die grundlegenden Kontroll-Anweisungen für strukturierte Programme:

- Fallunterscheidungen
- Schleifen mit Test der Abbruch Bedingung bei Beginn der Schleife oder am Ende.

4.1.1 Sequenz

Eine Sequenz ist eine Aneinanderreihung von einfachen Anweisungen, welche sequentiell ausgeführt werden. Für den Compiler gilt das ganze als Sequenz, wenn die Anweisungen in geschweiften Klammern sind.

```
1 // Beginn Sequenz
2 {
3   statement1;
4   statement2;
5   ...
6   statementN;
7 }
8 // Ende Sequenz
```

4.1.2 if Selektion

Die if Selektion wird verwendet, wenn ein Code nur dann ausgeführt werden soll, wenn eine bestimmte Bedingung wahr ist.

if - Selektion mit jeweils einem Statement:

```
if (condition)
  statement;
else
  statement;
```

if - Selektion mit mehreren Statements und mehreren Abfragen:

```
if (condition){
  statement1;
  statement2;
  ...
}
else if (condition){
  statement1;
  statement2;
  ...
}
else{
  ...
}
```

Der `else` Teil kann auch wegfallen.

Aufbau von Bedingungen

Bedingungen können jeweils wahr `(true)` oder falsch `(false)` sein.
Bedingungen können mit Vergleichsoperatoren und Logischen Operatoren aufgebaut werden.

Beispiele von Bedingungen:

```
n>=10 && n<=100
(n==4 || n==5) && (x<100.0)
!(x==0 || y==0)
```

Beispiel:

Implementierung einer Mehrfach Abfrage mit dem `if` Statement für Wochentage.

```
int day;
...
if (day == 0) cout << "Sunday" << endl;
else if (day == 1) cout << "Monday" << endl;
...
else if (day == 6) cout << "Saturday" << endl;
else cout << "not a valid day" << endl;
```

4.1.3 switch Selektion

Die `switch` Selektion implementiert eine Sprungtabelle. Es wird eine Variable vom Typ Integer abgefragt und danach der entsprechende Fall ausgeführt. Die Ausführung ist nun sequentiell, bis eine `break` Anweisung kommt.

```
int day;

switch(day){   // Abfragen der Integer Variable day

  case 0:  cout << "Sunday" << endl;
           break;

  case 1:  cout << "Monday" << endl;
           break;

  ...

  default: cout << "Not a valid day" << endl;
}
```

Bemerkungen:

1. Die `break` Anweisungen sind unbedingt erforderlich. Vergisst man sie, so werden ab dem ersten Fall der zutrifft, alle Befehle von diesem und den nachfolgenden Fällen ausgeführt.
2. Der `default` Zweig kann weggelassen werden.
3. Man beachte, dass die Statements der einzelnen Fälle nicht in Klammern eingeschlossen werden müssen.

4.1.4 for Schleife

Die `for` Schleife besteht aus den folgenden drei Teilen:

- Inital Statement (Bsp: `int i=0`)
- Bedingung (Bsp: `i<10`)
- Regel (Bsp: `int i++`)

```
for (init; condition; rule){
  // code
}
```

Ausführungsschritte:

1. Ausführung des Initial Statements (init)
2. Überprüfen der Schleifenbedingung (condition)
3. Ausführen der Schleife
4. Bedingung prüfen (rule)
   Ist die Bedingung wahr, wird mit Schritt 3 weitergefahren. Ist die Bedingung falsch, wird die Schleife beendet.

Beispiel:

Berechnung der harmonischen Reihe mit einer for Schleife.

Harmonische Reihe:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ...$$

```
#include <iostream>
using namespace std;

int main(int argc, char **argv){
  int numberOfElements = 30;
  float sum = 1;
  for (int i=2; i<=numberOfElements; i++){
    sum = sum + 1.0/i;
  }
  cout << sum << endl;
  return 0;
}
```

4.1.5 while Schleife

Bei `while` Schleifen wird eine bestimmte Verarbeitung wiederholt durchgeführt, solange eine Bedingung erfüllt ist. Die Bedingung wird jeweils vor der Durchführung der Schleife geprüft. Ist die Bedingung bereits beim ersten Mal falsch, wird die Schleife nicht ausgeführt.

```
while (condition){
  // code
}
```

Beispiel:

Berechnung der harmonischen Reihe mit einer while Schleife.

Harmonische Reihe:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ...$$

```
#include <iostream>
using namespace std;

int main(int argc, char **argv){
  int numberOfElements = 30;
  float sum = 1;
  int i = 2;
  while (i <= numberOfElements){
    sum = sum + 1.0/i;
    i++;
  }
  cout << sum << endl;
  return 0;
}
```

4.1.6 do while Schleife

Die `do while` Schleife ist von der Funktion identisch mit der `while` Schleife. Der einzige Unterschied ist, dass die Bedinung jeweils nach Ausführung der Schleife geprüft wird. Das hat zur Folge, dass die Schleife mindestens einmal ausgeführt wird.

```
do {
  // code
} while (condition);
```

Beispiel:

Berechnung der harmonischen Reihe mit einer do while Schleife.

Harmonische Reihe:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ...$$

```
#include <iostream>
using namespace std;

int main(int argc, char **argv){
  int numberOfElements = 30;
  float sum = 1;
  int i = 1;
  do {
    i++;
    sum = sum + 1.0/i;
  } while (i <= numberOfElements);
  cout << sum << endl;
  return 0;
}
```

4.1.7 break und continue

In den oben aufgelisteten Schleifen können auch noch zwei besondere Anweisungen stehen:

- `break`
  dient neben dem Abbruch eines Falls in einer Switch - Anweisung auch zum sofortigen Abbruch einer Schleife. Das ist im Prinzip ein Sprung hinter die Schleife und sollte nur in Sonderfällen eingesetzt werden.
- `continue`
  dient zum sofortigen Neueintritt in eine Schleife. Die aktuelle Iteration wird sofort abgebrochen und mit der nächsten fortgefahren. Auch diese Anweisung sollte nur in Sonderfällen eingesetzt werden.

4.2 Zufallszahlen

C++ stellt die Möglichkeit zur Verfügung, Zufallszahlen zu erzeugen. Die Zufallszahlen werden mit der Funktion `rand()` generiert.

```
// number liegt zwischen 0 und RAND_MAX
int number = rand();
```

Der Wert der Zufallszahl liegt zwischen 0 und der Konstanten `RAND_MAX`.

Sollen Zufallszahlen zwischen 0 und 1 generiert werden, so kann die Funktion `rand()` wie folgt verwendet werden:

```
// number liegt zwischen 0 und 1
float number = (float)rand() / RAND_MAX;
```

Sollen ganze Zufallszahlen zwischen 0 und `n` generiert werden, so kann die Funktion `rand()` wie folgt verwendet werden:

```
// number liegt zwischen 0 und n (100)
int n = 100;
int number = rand() % n;
```

Damit nicht immer die gleichen Zufallszahlen generiert werden, kann der Zufallszahlengenerator mit der Funktion `srand(n)` initialisiert werden:

```
srand(1000); // Initialisierung
int number = rand();
```

Für den Parameter der Funktion `srand(n)` kann die aktuelle Systemzeit verwendet werden:

```
// Initialisierung mit Systemzeit (#include <ctime>)
srand((unsigned) time(0));
int number = rand();
```

4.3 Funktionen

Eine Funktion in C++ ist folgendermassen aufgebaut:

```
resultType functionName (parameters){

  // Code

  return returnValue;

}
```

Beispiel:

Berechnung der harmonischen Reihe mit einer Funktion.

```
#include <iostream>
using namespace std;

float harmonicSerie(int numberOfElements){
  float sum = 1;
  for (int i=2; i<=numberOfElements; i++){
    sum = sum + 1.0/i;
  }
  return sum;
}
```

4.3.1 Aufruf einer Funktion

Eine Funktion wird durch Angabe des Namens und der Parameter aufgerufen. Um das Resultat der Funktion zu verwenden, kann eine Zuweisung verwendet werden.

Aufruf der Funktion für die harmonische Reihe:

```
// Aufruf mit Verwendung des Resultates
float value = harmonicSerie(1000);
```

```
// Aufruf ohne Verwendung des Resultates
harmonicSerie(1000);
```

4.3.2 Lokale Variablen

Variablen, die in einer Funktion definert sind, heissen lokale Variablen. Sie werden beim Aufruf der Funktion im Speicher angelegt und beim Ende der Funktion wieder gelöscht. Es können selbstverständlich auch lokale Konstanten angelegt werden.

4.3.3 Return Statement

Das `return` Statement setzt das Resultat einer Funktion und beendet die Funktion.

```
return returnValue;
```

Der angegebene Ausdruck wird, wenn nötig, automatisch in den Resultattyp konvertiert.
In einer Funktion können meherere `return` Statements vorkommen.

4.3.4 Statische lokale Variablen

Wird eine Funktion aufgerufen, so werden die lokalen Variablen im Speicher angelegt und am Schluss wieder gelöscht. Wird das Schlüsselwort `static` vor die Variable geschrieben, so wird die lokale Variable am Schluss der Funtkion nicht gelöscht, sondern bleibt im Speicher und kann beim nächsten Aufruf wieder verwendet werden. Sie ist aber trotzdem nur in der Funktion bekannt.

```
static int i;
```

4.3.5 Funktionen ohne Resultat

Besitzt eine Funktion kein Resultat, so kann als Resultattyp `void` verwendet und das `return` Statement weggelassen werden.

```
void functionName (parameter){

  // Code

}
```

Beispiel:

Implementierung einer Funktion zur Berechnung des grössten gemeinsamen Teilers (GGT) zweier Zahlen.

```
#include <iostream>
using namespace std;

int ggt(int a, int b){
  while (a !=b){
    if (a < b){
        b = b - a;
    }
    else {
        a = a - b;
    }
  }
  return a;
}
```

4.3.6 Funktion überladen

Überladen bedeutet, dass eine Funktion mehrmals definiert wird. Sie besitzt den gleichen Namen aber unterschiedliche Parameter.

Beispiel:

Überladen der Methode `foo`. Sie hat immer den gleichen Rückgabetyp, denselben Namen aber unterschiedliche Parameter.

```
#include <iostream>
using namespace std;

void foo(int m){
  // code
}

void foo(float x, int n){
  // code
}

void foo(float x, float y){
  // code
}
```

4.3.7 Defaultparameter

Für jeden Parameter, der in einer Funktion definiert wird, muss der Aufrufer entsprechende Werte übergeben. Sind diese Werte nicht korrekt oder die Parameter nicht vollständig, erscheint ein Compilerfehler.

```
1 int function(int a, int b){...}
```

Um diese Funktion aufzurufen, müssen zwei Parameter übergeben werden. Für diese Regel existiert eine Ausnahme. Defaultparameter.

```
1 int function(int a, int b = 10){...}
```

Nun kann die Funktion entweder mit zwei Parametern (a und b) oder nur mit einem Parameter (nur a) aufgerufen werden. Wird die Funktion mit einem Parameter aufgerufen, so wird für den Parameter b der Wert 10 verwendet. Aufruf: `int m = function(3);`.

Man kann allen Parametern einer Funktion Defaultwerte zuweisen. Die einzige Einschränkung besteht darin, dass wenn ein Parameter keinen Defaultwert besitzt, so kann kein weiterer Parameter, welcher zuerst in der Parameterliste vorkommt, einen Defaultwert besitzen.

```
1 // NICHT ERLAUBT!
2 int function(int a=10, int b, int c=4){...}
```

Beispiel:

Mit dem folgenden Algorithmus (Babylonian Methode) kann die Wurzel einer Zahl berechnet werden. Die Berechnung erfolgt mit Hilfe einer Schleife, welche je nach Genauigkeit des Resultates eine bestimmte Anzahl durchlaufen wird. Die Funktion kann nun verwendet werden mit oder ohne Angabe dieser Genauigkeit. Wird die Genauigkeit nicht angegeben, so wird der Defaultwert verwendet.

```
1  double sqroot(double x, double eps = 10E-5){
2    double y = (x+1) / 2;
3
4    while (abs(x-y*y) > eps){
5      y = (y + x/y) / 2;
6    }
7    return y;
8  }
9
10 // Aufruf mit einem Parameter (eps = defaultwert)
11 cout << sqroot(120) << endl;
12
13 // Aufruf mit zwei Parametern
14 cout << sqroot(120, 10E-10) << endl;
```

4.4 Struktogramme

Struktogramme unterstützen die strukturierte Programmierung. Mit ihnen kann ein strukturierter Programmablauf graphisch dargestellt werden.

4.4.1 Elemente

Strukturierte Programme bestehen aus den folgenden Elementen:

- Anweisungen, Sequenzen
- Selektionen
- Schleifen

Anweisung

Einfache Anweisungen werden in rechteckige Kästen gesetzt. Diese Art von Struktogramme ist nicht geschachtelt. Der Formalismus der Anweisung selbst ist nicht fixiert.

Abbildung 4.1: Struktogramm: Anweisung

Sequenz

Lineare Abfolgen werden durch lückenloses Untereinandersetzen von Anweisungen ausgedrückt.

Abbildung 4.2: Struktogramm: Sequenz

Selektionen

Die Bedingung regelt, welches der beiden untergeordneten Struktogramme ausgeführt wird. Das jeweils andere wird nicht durchlaufen. Überlicherweise steht der `True`-Fall auf der linken Seite.

if Selektion

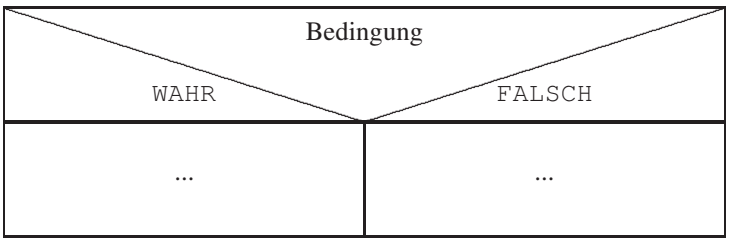


Abbildung 4.3: Struktogramm: Selektion

Mehrfach if Selektion, switch Statement

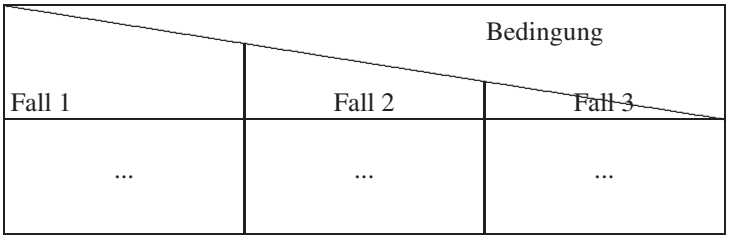


Abbildung 4.4: Struktogramm: Mehrfach Selektion

Beispiel:

Überprüfen ob ein Jahr ein Schaltjahr ist.

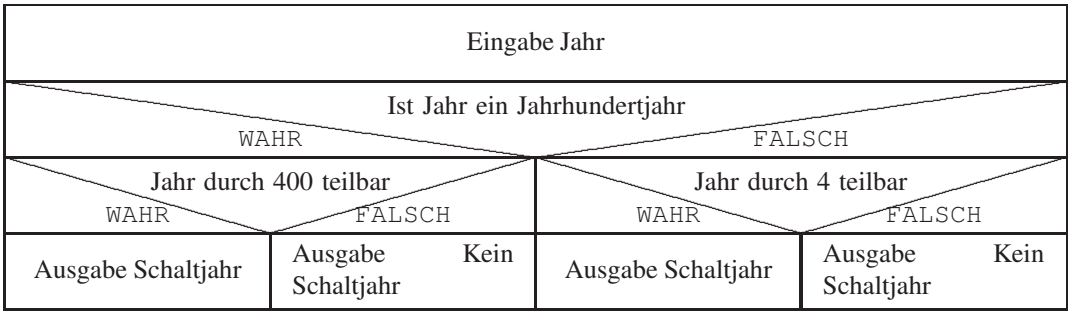


Abbildung 4.5: Struktogramm: Schaltjahrtest

Schleifen

Es existieren zwei Darstellungsarten für Schleifen:

Abweisende Schleifen
Bereits vor dem ersten Schleifendurchgang wird die Schleifenbedingung überprft (while, for).

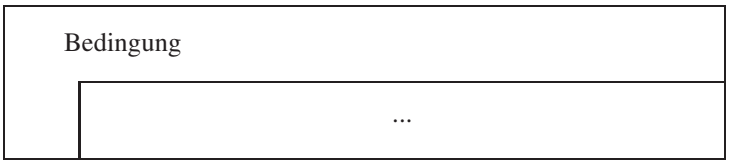


Abbildung 4.6: Struktogramm: Abweisende Schleife

Annehmende Schleifen
Erst nach dem ersten Schleifendurchgang wird die Schleifenbedingung das erste Mal überprüft (do-while).

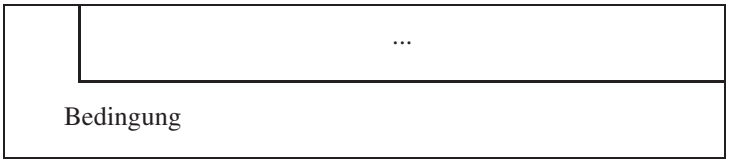


Abbildung 4.7: Struktogramm: Annehmende Scheife

Beispiel:

Überprüfung ob ein Rechteck ein Quadrat ist.

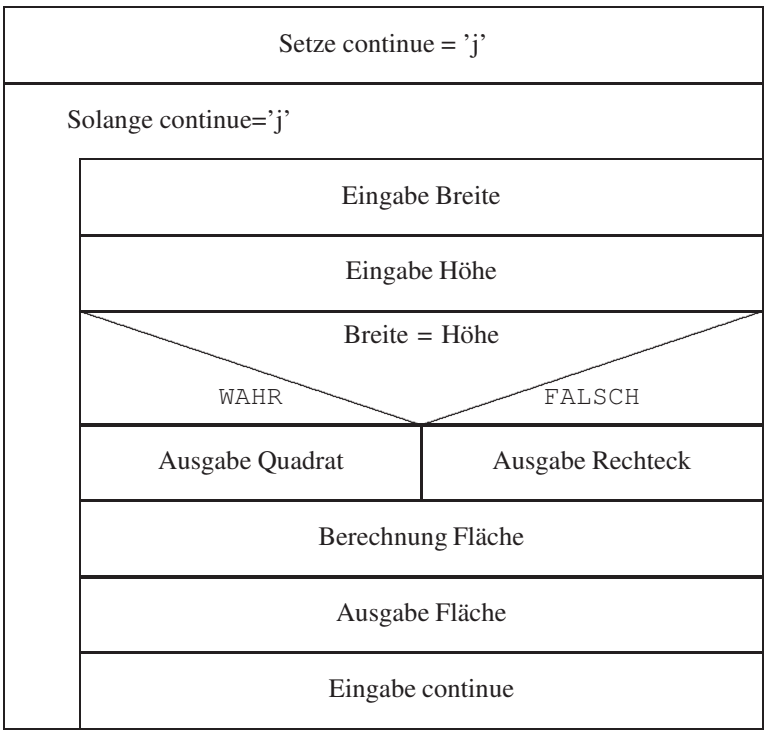


Abbildung 4.8: Struktogramm: Rechteck - Quadrat Test

4.5 Aufgaben

4.5.1 Quadratische Gleichung

Eine quadratische Gleichung $ax^2 + bx + c = 0$ kann mit der folgenden Formel gelöst werden:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2 \cdot a}$$

D ist die sogenannte *Diskriminante*. Diese kann folgendermassen bestimmt werden:

$$D = b^2 - 4 \cdot a \cdot c$$

Bei der Diskriminanten werden drei verschiedene Fälle unterschieden:
`D > 0` : Die quadratische Gleichung ergibt zwei verschiedene reelle Lösungen
`D = 0` : Die quadratische Gleichung ergibt eine reelle Lösung
`D < 0` : Die quadratische Gleichung ergibt keine reelle Lösung

Schreiben Sie ein Programm, das die Variablen a,b und c einliest und die Lösung(en) der Gleichung berechnet. In einem ersten Schritt soll nur die Diskriminante berechnet werden. Anhand der Diskriminate soll bestimmt werden wiviele Lösungen es gibt. Anschliessend werden die Anzahl Lösungen berechnet und auf dem Bildschirm ausgegeben.

4.5.2 Kleinste Zahl

Schreiben Sie ein Programm, welches 10 Zahlen über die Konsole einliest und die kleinste dieser 10 Zahlen am Schluss wieder auf der Konsole ausgibt.
Zeichnen Sie auch ein Struktogramm zu Ihrer Lösung.

4.5.3 Berechnung der Zahl PI

Archimedes Methode

Archimedes versuchte über den Umfang eines n-Ecks π zu berechnen. Er errechnete daher den Umfang eines n-Ecks, das den Kreis umschreibt, und eines n-Ecks, das vom Kreis umschrieben wird. Je grösser n wird, desto näher liegt das arithmetische Mittel dieser beiden Werte am wahren Wert von π.

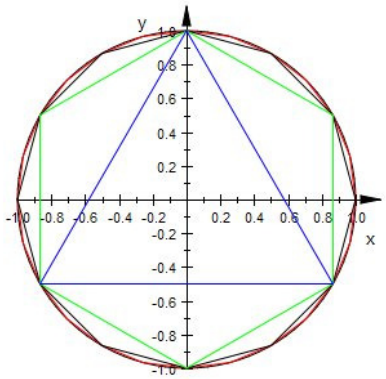


Abbildung 4.9: Methode von Archimedes zur Berechnung von PI

Schreiben Sie ein Programm, welches mit einer Schleife schrittweise den Wert für n bis 10000 erhöht und geben Sie bei jedem Durchlauf den Wert von π aus.

PI erschiessen

Eine weitere Möglichkeit die Zahl π zu berechnen ist die folgende: π wird approximiert, indem man einem Quadrat einen Viertelkreis einschreibt und dieses Quadrat dann mit Zufallspunkten beschiesst. Das Verhältnis der Punkte, die innerhalb des Kreisbogen liegen, zur Gesamtzahl der abgegebenen Schüsse nähert sich bei wachsender Schusszahl dem Verhältnis der Flächeninhalte von Viertelkreis und Quadrat. Folglich kann π aus diesem Verhältnis berechnet werden.

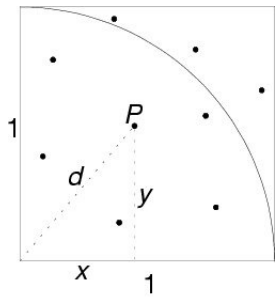


Abbildung 4.10: PI erschiessen

Ist der Abstand d eines Punktes kleiner 1, so liegt der Punkt innerhalb des Viertelkreises. Der Abstand d kann mit Hilfe des pythagoreischen Lehrsatzes berechnet werden: $d = \sqrt{x^2 + y^2}$.

4.5.4 Rekorde

Ein Programm soll eine Folge von n = 100 Zufallszahlen $x_1, x_2, x_3, ..., x_n$ erzeugen. Eine Zahl x_i der Folge ist ein Rekord wenn sie grösser ist als alle vorangegangenen Zahlen. Die Anzahl Rekorde ist also eine beliebige Zahl im Bereich $1...n$. Wir interessieren uns für die zu erwartende Anzahl Rekorde. Schreiben Sie ein Programm zur Bestimmung des Mittelwertes der Anzahl Rekorde einer Folge.

4.5.5 Roulette

Schreiben Sie ein Programm bei welchen ein Roulette Spiel simuliert wird. Beim Roulette wird jeweils zufällig eine Zahl zwischen 0 und 36 bestimmt. Gehen Sie davon aus, dass ein Spieler immer einen bestimmten Betrag auf eine gerade Zahl setzt (0 ist weder gerade noch ungerade). Kommt eine gerade Zahl, so verdoppelt der Spieler seinen Gewinn, kommt eine ungereade Zahl, so verliert der Spieler seinen Einsatz.
Gehen Sie davon aus, dass der Spieler mit 10000 Fr. beginnt und bei jedem Spiel 10 Fr. als Einsatz verwendet. Simulieren Sie 2000 Spiele. Wieviel Geld hat der Spieler am Ende der 2000 Spiele?

Kapitel 5

Fortgeschrittene Datentypen

Das Ganze ist mehr als die Summe
seiner Einzelteile.
Laotse

5.1 Arrays

Eine Definition von Array Variablen besteht in C++ aus einem Datentyp, einem Namen für den Array und der Anzahl Elemente in eckigen Klammern:

```
1 float point[2];
```

Die einzelnen Elemente werden mit eckigen Klammern indiziert. Elemente des oberen Arrays:

`point[0], point[1]`

⋆ *Achtung:*
Der Index Bereich in einem Array mit n Elementen beträgt $0..(n-1)$.

⋆ *Achtung:*
C++ prüft nicht, ob der bei einem Zugriff auf ein Array Element verwendete Index Wert gültig ist, d.h. wenn ein Wert mit einem zu grossen Index in einen Array eingetragen wird, wird Speicher überschrieben, der nicht zum Array gehört.

⋆ *Merke:*
Bei der Definition eines Arrays muss die Anzahl Elemente als konstanter Ausdruck angegeben werden, d.h. als Ausdruck mit Konstanten.

5.1.1 Initialisierung

Ein Array kann direkt bei seiner Deklaration mit Werten initialisiert werden:

```
float point[2] = {1.2, 4.2};
```

Bei der Definition eines Arrays mit Initialisierung, kann die Anzahl Elemente des Arrays weggelassen werden.

```
float point[] = {1.2, 4.2};
```

⋆ *Merke:*
Die Initialisierung in dieser Form geht nur bei der Definition des Arrays!

5.1.2 Elemente durchlaufen

Möchten Sie alle Elemente eines Arrays durchlaufen, so können Sie dies mit einer `for` Schleife tun.

```
float arrayName[1000];
for (int i=0; i<1000; i++){
  arrayName[i] = ....
}
```

5.1.3 Zweidimensionale Array

Ein zweidimensionaler Array (Matrix) ist in C++ ein Array, dessen Elemente wieder Arrays sind.

```
dataType arrayName[rows][columns];
```

Definition einer Float Matrix mit 3 Zeilen und 4 Spalten:

```
float matrix[3][4];
```

Elemente der Matrix:

`matrix[i][j]` $(0 \leq i < 3, \quad 0 \leq j < 4)$

Initialisierung bei der Definition

```
float matrix[3][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4}};
```

Elemente durchlaufen

Möchten Sie alle Elemente einer Matrix durchlaufen, so können Sie dies mit 2 for-Schleifen tun.

```
1 float matrix[rows][columns];
2
3 for (int i=0; i<rows; i++){
4   for (int j=0; j<columns; j++){
5     matrix[i][j] = ...
6 }
```

5.2 Vektoren

Da der Umgang mit Arrays nicht besonders komfortabel ist und ein Array eine feste Grösse hat, wird an dieser Stelle die Klasse `vector` eingeführt.

Um einen Vektor zu verwenden, muss die folgende `include` Anweisung im Source Code ergänzt werden:

```
#include <vector>
```

Die Anweisung

```
vector <int> v(10);
```

stellt einen Vektor `v` bereit, der 10 Elemente des Datentyps `int` aufnehmen kann. Auf die einzelnen Elemente kann genau gleich wie beim Array zugegriffen werden.

```
v[0] = 12;
cout << v[9] << endl;
```

Zusätzlich kann mit `at(index)` auf die einzelnen Elemente zugegriffen werden. Der Unterschied zu den eckigen Klammern liegt darin, dass der Zugriff mit `at` prüft, ob der Indexwert gültig ist. Dies bedeutet zwar mehr Sicherheit, jedoch auch eine schlechtere Performance.

```
cout << v.at(0) << endl;
```

Ein Vektor kann auch nach seiner Grösse abgefragt werden.

```
int size = v.size();
```

Die Klasse `vector` wird in einem späteren Kapitel noch ausführlich behandelt.

5.3 Strings

Ein String ist ein Datentyp, welcher eine Zeichenkette beinhaltet, wobei jedes Zeichen ein Character ist.

```
1 // Gibt Hello World auf den Bildschrim
2 string output = "Hello World";
3 cout << output << endl;
```

5.3.1 Einlesen eines Strings

Ein String kann von der Kommandozeile mit der Funktion `getline` eingelesen werden.

```
1 string input;
2 getline(cin,input);
```

5.3.2 Zugriff auf einen String

Der Zugriff auf ein Zeichen erfolgt genau gleich wie bei einem Array. Um die Anzahl Zeichen (Länge) eines Strings zu bestimmen, kann die Funktion `length()` oder `size()` verwendet werden.

```
1 string output = "Hello World";
2 // Zugriff auf erstes Zeichen
3 char ch = output[0]; // ohne Indexpruefung
4 char ch = output.at(0); // mit Indexpruefung
5 // Abfragen der Laenge
6 int n = output.length();
```

5.3.3 Durchlaufen eines Strings

Um den String zu durchlaufen und auf jedes Zeichen einzeln zuzugreifen, existieren zwei Möglichkeiten.

String zeichenweise ausgeben (ohne Indexprüfung)

```
1 string output = "Hello World";
2 for (int i=0; i<output.length(); i++){
3     cout << output[i] << endl;
4 }
```

5.3.4 Zusammenfügen von Strings

Um zwei Strings zusammenzufügen, kann der Operator + verwendet werden.

```
1 string s1 = "Hello";
2 string s2 = "World";
3 string s3 = s1 + s2; // in s3 steht HelloWorld
```

5.3.5 String kopieren

Ein String kann folgendermassen kopiert werden:

```
1 string source = "Hello";
2 string target(source);
3 cout << target << endl; // Hello wird ausgegeben
```

5.3.6 String matching

Exact matching: what's the problem?[1]

Given a string P called the pattern and a longer string T called the text. The exact matching problem is to find all occurences, if, any of pattern P in text T.

For example, if `P=aba` and `T=bbabaxababay` then P occurs in T starting at locations 3,7 and 9.

Es gibt verschiedene Algorithmen um dieses Problem zu lösen. Zwei sehr bekannte Verfahren sind der Boyer-Moore und der Knuth-Morris-Pratt Algorithmus.

[1] Dan Gusfield, Algorithms on strings, trees and sequences

5.4 String Streams

String Streams sind Streams, welche keine Datei oder ein Ein- bzw. Ausgabegerät benötigen, da sie im Memory verwaltet werden. Sie erlauben die folgenden Operationen:

- formatiertes Lesen (Input) von Werten aus einem String. Damit kann ein String einfach in einzelne Wörter getrennt oder in andere Datentypen umgewandelt werden.
- formatiertes Schreiben (Output) von Werten. Damit können Werte verschiedener Datentypen in einen String geschrieben werden.

5.4.1 Werte von einem String lesen

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main(int argc, char **argv){

  // string, aus welchem gelesen wird
  string s = "23.5 100 Hello, ERROR";

  // stringstream deklarieren
  istringstream sStream(s);

  int intValue;
  if (sStream.good()) sStream >> intValue;
  else cout << "Stream not ready for io" << endl;
  cout << intValue << ", Success: "
       << sStream.fail() << endl;

  float floatValue;
  if (sStream.good()) sStream >> floatValue;
  else cout << "Stream not ready for io" << endl;
  cout << floatValue << ", Success: "
       << sStream.fail() << endl;

  string stringValue;
  if (sStream.good()) sStream >> stringValue;
  else cout << "Stream not ready for io" << endl;
  cout << stringValue << ", Success: "
       << sStream.fail() << endl;

  int errorValue;
  if (sStream.good()) sStream >> errorValue;
  else cout << "Stream not ready for io" << endl;
  cout << errorValue << ", Success: "
```

```
36        << sStream.fail() << endl;
37
38    return 0;
39  }
```

5.4.2 Werte in einen String schreiben

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>
4  using namespace std;
5
6  int main(int argc, char **argv){
7
8    // string, in welchen geschrieben wird
9    string s = "";
10
11   // stringstream deklarieren
12   ostringstream sStream;
13
14   float floatValue = 23.5;
15   sStream << floatValue << " ";
16
17   int intValue = 100;
18   sStream << intValue << " ";
19
20   string stringValue = "Hello World";
21   sStream << stringValue;
22
23   string newString = sStream.str();
24   cout << newString << endl;
25
26   return 0;
27 }
```

5.5 Pointers

Eine *Pointer-Variable* ist eine Variable, in der eine Speicheradresse abgespeichert werden kann, z.B. die Adresse einer anderen Variablen. Wenn eine Pointer-Variable die Adresse einer anderen Variablen enthält, sagt man die Pointer-Variable *zeige* auf diese Variable.

Pointer-Variablen haben in C++ eine zentrale Stellung, da sie eng verbunden sind mit Arrays und auch bei der Übergabe von Parametern an ein Unterprogramm eine wichtige Rolle spielen.
Technisch gesehen, kann eine Pointer-Variable eine beliebige Speicheradresse enthalten. Zur Verminderung von Fehlerquellen ist jedoch eine Pointer-Variable in C++ (wie in anderen Sprachen) an einen Datentyp gebunden, d.h. sie kann nur Adressen von Variablen dieses Typs aufnehmen (*Basis-Typ* der Pointer-Variable).

5.5.1 Definition einer Pointer-Variablen

Die Definition einer Pointer-Variablen besteht aus der Angabe des Basis-Typs, einem `*` und einem Namen für die Pointer Variable:

```
1 dataType *pointerNname; // Allgemeine Deklaration
2 float *pointer; //Pointer-Variable fuer float
```

Dabei ist `dataType` ein beliebiger gültiger Datentyp. Der `*` ist das Kennzeichen, dass die nachfolgende Variable eine Pointer-Variable ist.
Die Anzahl Blanks vor und nach dem `*` ist beliebig, der `*` kann auch direkt (ohne Blank) vor dem Variablen-Namen stehen.

Achtung:
Der `*` muss vor jeder Pointer-Variable wiederholt werden:

```
1 float *p1, *p2; // Zwei Pointer-Variablen
2 float *p1, p2;  // hier ist p2 keine Pointer-Variable
```

5.5.2 Der Adressoperator

Das Symbol `&` vor einer Variablen `x` liefert die Speicheradresse der Variablen:

```
1 &x  <== Adresse der Variablen x
```

Die Adresse muss in einer zugehörigen Pointer-Variablen gespeichert werden:

```
1 float x = 3.14159;
2 float *p;
3 p = &x;  // Adresse von x in p abspeichern
```

5.5.3 Der Dereferenzierungsoperator

Ist `p` eine Pointer-Variable, welche auf eine Variable `x` zeigt, so kann `x` mit dem Ausdruck `*p` (Dereferenzierung von `p`) angesprochen werden. Mit anderen Worten:

`*p` ist eine Variable des Basis-Typs von `p`.

Beispiel:

Deklaration einer `float` Variablen und eines Pointers auf eine `float` Variable, sowie Benutzung des Adressoperator und Dereferenzierungsoperator.

```
1 float x;
2 float *p;
3 p = &x;    // Adresse von x in p abspeichern
4 *p = 3.4; // aequivalent mit x = 3.4
```

Der Ausdruck `*p` kann wie eine normale Variable des betreffenden Typs (`float` in unserem Beispiel) verwendet werden.

▷ *Merke:*
Die Definition einer Pointer-Variablen in der Form `float *p` kann auch so gelesen werden, dass `p` eine Pointer-Variable ist, deren Dereferenzierung `*p` eine Variable vom Typ `float` ist. Dies erklärt, wieso das Symbol `*` zur Definition einer Pointer-Variablen und bei der Dereferenzierung verwendet wird.

5.5.4 Die Speicheradresse 0 (NULL)

Im Header-File `iostream` ist die Konstante `0 (NULL)` definiert, als Wert für Pointer Variablen, die keine gültige Speicheradresse enthalten:

```
1  float *p
2  p = 0;
3  ...
4  if (p == 0)
5
6  // oder
7
8  float *p;
9  p = NULL;
10 ...
11 if (p == NULL) ...
```

Es ist sowohl `0` wie auch `NULL` gültig. In diesem Script wird der Wert `0` verwendet.

5.5.5 Der Zuweisungsoperator für Pointer-Variablen

Für Pointer-Variablen mit gleichem Basistyp, steht der Zuweisungsoperator = zur Verfügung. Bei Zuweisungen von Pointer-Variablen mit verschiedenen Basis Typen (nur für spezielle Situationen) sind explizite Konversionen erforderlich:

```
1 float *floatPtr;
2 char *charPtr;
3 charPtr = (char *)(floatPtr);
```

Die Dereferenzierung `*charPtr` stellt auch nach der oberen Zuweisung eine `char`-Variable dar.

5.5.6 void-Pointers

Für spezielle Situationen können Pointer-Variablen zum leeren Datentyp `void` definiert werden:

```
1 void *ptr;
```

Die Variable `ptr` kann dann beliebige Adressen enthalten. Bei der Dereferenzierung und bei Zuweisungen zu anderen Pointers sind Konversionen erforderlich:

```
1 float x,y;
2 float *floatPtr;
3 void *ptr;
4
5 ptr = &x;
6 floatPtr = (float *)(ptr);   // Konversion in Float-Pointer
7                              // vor der Zuweisung
8 y = *(float *)(ptr);         // Konversion in Float-Pointer
9                              // vor der Dereferenzierung
```

5.5.7 Pointers als Werte-Parameter

Wenn einem Unterprogramm die Adresse einer Variablen als Werte-Parameter übergeben wird, kann das Unterprogramm direkt auf den Speicherbereich der betreffenden Variablen des aufrufenden Programmes zugreifen und folglich den Inhalt auch verändern.
Mit diesem Prinzip kann ein Unterprogramm also Daten an das aufrufende Programm zurückgeben.
Diese Methode ist in C die einzige Möglichkeit, mittels Parametern Daten von einem Unterprogramm an das aufrufende Programm zurückzugeben (ausser mit dem Returnwert einer Funktion). Da sie sehr fehleranfällig ist, wurde in C++ die bessere Methode der Referenzparameter eingeführt.

Die Kenntnis der direkten Methode mit Adressen ist nur noch wegen den alten C-Unterprogrammen von Bedeutung, z.B. `scanf`:

```
scanf("%d", &input);  // Adresse uebergeben
```

Beispiel:

Prozedur `swap` zur Vertauschung der Inhalte zweier `int`-Variablen:

```
void swap(int *p1, int *p2){
  int temp;
  temp = *p1;  // Dereferenzierung erforderlich
  *p1 = *p2;
  *p2 = temp;
}
```

Aufruf der Prozedur:

```
int a,b;
swap(&a, &b);  // Adressen muessen uebergeben werden!
```

5.5.8 Pointers auf Funktionen

Pointers können nicht nur auf Variablen zeigen, sondern auch auf Funktionen. Diese besonderen Pointer nennt man Funktionspointer.

Nehmen wir als Beispiel eine Funktion `divide()`, der wir einen Funktionspointer übergeben, die aufgerufen werden soll, wenn eine Division durch 0 stattfindet.

```
1  #include<iostream>
2  using namespace std;
3
4  double divide(double a, double b, void callback()){
5    if(b == 0.0 ){
6      callback();
7      return 0;
8    }
9    return a/b;
10 }
11
12 void error(){
13   cout << "ERROR" << endl;
14 }
15
16 int main(int argc, char **argv){
17   double result = divide(1,0,error);
18   return 0;
19 }
```

Callback ist ein Zeiger auf eine Funktion, nämlich auf `error()`. Um einen Zeiger auf die Funktion error() zu bekommen, lässt man einfach die Klammern weg: error.

Der Typ eines Funktionszeigers ist die Signatur der Funktion. Ein Zeiger auf eine Funktion `int foo(int)` kann nicht auf eine Funktion `void foo(int)` zeigen.

5.6 Referenzen

Eine Referenz ist ein Alias für eine bereits existierende Variable. Eine Referenz kann also nicht alleine existieren. Sie ist immer an eine Variable gebunden. Wird die Referenz geändert, so wird auch die entsprechende Variable geändert. Bei der Erzeugung wird eine Referenz an eine bereits existierende Variable gebunden. Von dieser kann die Referenz nicht mehr gelöst werden.

```
1 int a;
2 int & ref = a; // ref ist eine Referenz auf die Variable a
```

5.6.1 Referenzparameter

Wie bereits oben erwähnt, existieren in C++ Referenzparamter. Wird eine Funktion aufgerufen, so wird jeweils von den Parametern eine Kopie angelegt, welche am Ende der Funktion wieder gelöscht wird.

Wird nun allerdings ein `&` vor den Parameter geschrieben, so wird keine Kopie angelegt, sondern direkt das Original des Aufrufers verwendet.

Beispiel:

Beim folgenden Code wird in der Funktion mit der Variablen `a` gearbeitet, welche im Hauptprogramm deklariert wurde.

```
#include <iostream>
using namespace std;

void func(int &n , int m){

  n = 4;
  m = 5;

}

int main(int argc, char **argv){

  int a = 2, b = 3;

  func(a,b);

  cout << a << endl; // Ausgabe: 4
  cout << b << endl; // Ausgabe: 3

  return 0;
}
```

Die Variable `a` besitzt nach der Funktion den Wert 4, die Variable `b` besitzt immer noch den Wert 3.

Diese Art von Parameterübergabe bringt bezüglich Performance einen Vorteil, da der Parameter nicht kopiert werden muss.

Falls der Parameter als Referenz übergeben wird, der Parameter jedoch nicht geändert werden darf, so kann vor den Parameter das Schlüsselwort `const` geschrieben werden.

```
void func(const int &n, int m)
```

Die Variable n kann nur gelesen, nicht aber verändert werden.

5.7 Dynamischer Speicher

Bis jetzt hatten wir immer nur soviel Speicherplatz zur Verfügung wie wir deklariert haben. Dies war immer bereits beim Compilieren des Codes bekannt.
Mit dynamischem Speicher können wir zur Laufzeit Speicher reservieren.

5.7.1 Dynamische Erzeugung von Variablen

Mit dem Operator `new` kann in C++ während der Laufzeit Speicher reserviert werden. Der Operator liefert als Resultat die Adresse des reservierten Speichers. Falls bei der Reservierung ein Fehler auftritt, gibt der Operator den Wert `0` zurück.

Beispiel:

Dynamische Erzeugung einer float Variablen

```
#include <iostream>
using namespace std;

int main(int argc, char **argv){

  float *xp;
  xp = new float;

  if (xp == 0){
    // Speicherreservierung fehlgeschlagen
  }

  // Den Wert 4 im neu reservierten Platz speichern
  *xp = 4;

  return 0;
}
```

▷ *Merke:*
Dynamisch erzeugte Variablen bleiben erhalten, bis zum Ende des Hauptprogrammes, oder bis sie mit dem Operator `delete` freigegeben wurde. Dies gilt auch für dynamisch erzeugte Variablen in Funktionen.

5.7.2 Freigabe von dynamischem Speicher

Mit dem Operator `delete` können dynamisch erzeugte Variablen wieder freigegeben werden.

```
1 delete p;  // p ist eine Pointervariable
```

Nun wird der Speicherbereich freigegeben, auf welcher `p` im Moment zeigt.

▷ *Merke:*
Die Operatoren `new` und `delete` existieren nur in C++. In C sind diese Operatoren nicht vorhanden.

5.7.3 Speicherlöcher (Memory Leaks)

Bei der Erzeugung von dynamischen Variablen muss aufgepasst werden, dass keine Speicherlöcher entstehen. Ein Speicherloch ist ein reservierter Bereich im Speicher, welcher aber nicht mehr verwendet werden kann, da seine Adresse nicht bekannt ist.

Wie kann so etwas passieren? Betrachten Sie das folgende Beispiel:

```
1 float *xp;
2 xp = new float; // Reservierung von Speicher
3 *xp = 5;
4 xp = new float;
```

Bei der ersten Speicherreservierung wird eine `float` Variable im Speicher reserviert. In diese Variable wird nun der Wert 5 geschrieben. Jetzt wird ein zweites Mal Speicher reserviert. `xp` zeigt nun auf den neu reservierten Speicher. Die zuerst erzeugte Variable mit dem Wert 5 kann nun nicht mehr verwendet werden, ist aber immer noch im Speicher. Dieser Speicherbereich kann nicht mehr benutzt werden und wird als Speicherloch bezeichnet.

Um dieses Problem zu beheben, muss vor der 2. Speicherreservation der Operator `delete` aufgerufen werden.

```
1 ...
2 *xp = 5;
3 delete xp;
4 xp = new float;
5 ...
```

5.7.4 Dynamische Erzeugung von Arrays

Mit dem Operator `new` kann bei Bedarf ein Array von Elementen eines Typs alloziert werden:

```
1 int *a;
2 a = new int[10];
```

Dabei kann die Anzahl Elemente hier als beliebiger Integer angegeben werden.
Nach der früher eingeführten Konventionen für Pointers und Arrays, kann das i-te Element des erzeugten Arrays mit `a[i]` angesprochen werden.

Die Freigabe eines dynamisch erzeugten Arrays erfolgt mit dem Operator `delete` und den eckigen Klammern.

```
1 delete [] a;
```

5.7.5 Dynamische mehrdimensionale Array

Ein mehrdimensionaler Array mit Pointers wird ähnlich wie ein Array mit Pointers definiert. Für jede weitere Dimension kommt nun ein weiterer Stern vor die Variable bei der Deklaration.

Pointer auf ein eindimensionales Array von Integer Variablen.

```
1 int *a;
2 a = new int[10];
```

Pointer auf ein eindimensionales Array von Pointers auf Integer Variablen. Jeder einzelne Pointer dieses Arrays kann wiederum auf ein Array zeigen. So entstehen zwei Dimensionen.

```
1 int **a;
2 a = new int * [10];
```

Beispiel:

Definition eines zwei dimensionalen Arrays mit Pointers.

```
1 bool **matrix;
2 const int rows = 10;     // Anzahl Zeilen
3 const int columns = 10; // Anzahl Spalten
4
5 matrix = new bool * [columns];
6 for (i=0; i<columns; i++){
7   matrix[i] = new bool[rows];
8 }
```

5.8 Datenstrukturen

Eine Datenstruktur ist eine Menge von verschiedenen Daten. Erstellt wird eine Datenstruktur mit dem Schlüsselwort `struct`.

```
struct structName{

  dataType name1;
  dataType name2;
  ...

};
```

Beispiel:

Eine Datenstruktur für Produkte.

```
struct Product {
  string productName;
  float price;
  int number;
};
```

Nun kann eine Variable des Typs `Product` genau gleich angelegt werden wie eine `float` oder `int` Variable.
Auf die einzelnen Komponenten kann mit dem `.` Operator zugegriffen werden.

```
Product p1;
p1.name = "Soap";
p1.price = 10.95;
p1.number = 104;

// Initialisierung bei der Deklaration
Product p2 = {"Soap", 10.95, 104};
```

5.8.1 Pointers auf Datenstrukturen

Nun wird ein Pointer, welcher auf eine `Product` Variable zeigt, erstellt. Über einen Pointer sieht dann der Zugriff auf die einzelnen Daten der Struktur ein wenig anders aus.

```
1  Product *p;
2  Product p1;
3
4  p = &p1;  // p zeigt nun auf p1
5
6  // Mit dem Punktoperator
7  (*p).name = "Soap";
8  (*p).price = 10.95;
9  (*p).number = 104;
10
11 // Mit dem Pfeil
12 p -> name = "Soap"
13 p -> price = 10.95
14 p -> number = 104
```

5.9 Benutzerdefinierte Datentypen

5.9.1 Typedef

In C++ können wir unsere eigenen Datentypen definieren, welche auf bereits existierenden Datentypen beruhen. Dies erfolgt mit dem Schlüsselwort `typedef`.

```
typedef existingType newType
```

Beispiel:

Neuer Datentyp für eine Farbe und ein Datentyp für Punkte:

```
#include <iostream>
using namespace std;

int main(int argc, char **argv){

  // Definition eines neuen Datentyps fuer Farben
  typedef int color;

  // Anlegen einer Variable des Datentyps color
  color c;
  c = 3;

  // Definition eines neuen Datentyps fuer Punkte
  typedef int point[2];

  point x;
  x[0] = 10;
  x[1] = 20;

  return 0;
}
```

5.10 Aufgaben

5.10.1 Galtonsches Brett

Das Galtonsche Brett ist ein vertikales Brett mit Nägeln, an welchen herunterfallende Kugeln abgelenkt werden. Die Nägel sind in r=5 horizontalen Reihen angeordnet, sodass eine fallende Kugel bei jeder Reihe zufällig nach links oder rechts abgelenkt wird, bis sie unten in ein Fach fällt.

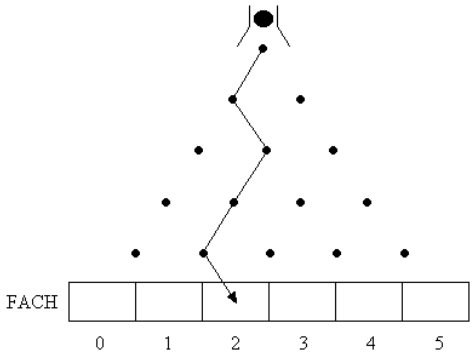


Abbildung 5.1: Galtonsches Brett

Wir interessieren uns dafür, wieviele Kugeln in den einzelnen Fächer landen.

Erstellen Sie ein Programm, welches mittels Simulation für n=100 Kugeln die Verteilung auf die Fächer bestimmt.

Ausgabe des Programms (keine Graphik):

```
Fach    Anzahl Kugeln
0       4
1       15
...     ...
```

Beachten Sie, dass die Fach-Nr. einer Kugel gleich der Anzahl Rechtsablenkungen ist.

5.10.2 Palindrome

Palindrome sind Wörter, welche vorwärts und rückwärts gelesen identisch sind.

Beispiele von Palindromen:

- OTTO
- LAGERREGAL
- NEFFEN

Implementieren Sie eine C++ Funktion, welche prüft, ob das als Parameter übergebene Wort ein Palindrom ist. Falls ja, gibt die Methode `true` zurück, ansonsten `false`.

```
1 bool isPalindrome(const string &word){
2   ...
3 }
```

5.10.3 Datenstruktur für Vielecke in der Ebene

Kapitel 6

Objektorientiertes Programmieren

Nicht weil es schwer ist, wagen wir es nicht,
sondern weil wir es nicht wagen ist es schwer.
Seneca

6.1 Klassen

Klassen sind Elemente, welche sowohl Daten als auch Funktionen beinhalten. Eine Klasse ist sehr ähnlich wie eine Datenstruktur, welche mit `struct` definiert wird. Definiert wird eine Klasse mit dem Schlüsselwort `class`.

Betrachten Sie die folgende Klasse `Product`.

```
1 class Product{
2
3   string name;
4   float price;
5   int number;
6
7 };
```

Würde man nun eine Variable der Klasse `Product` anlegen und auf die einzelnen Daten zugreifen, so würde der Compiler ein Fehler melden.

```
1 Product p;
2 p.name = "Soap";  // Ergibt einen Fehler
```

Wieso ergibt dies einen Fehler? Alle Elemente in einer Klasse sind per default nur innerhalb der Klasse zugänglich. Von aussen sind Zugriffe auf die Daten verboten. Dies kann jedoch geändert werden. Man kann innerhalb einer Klasse Daten definieren,

die von aussen zugänglich sind, so wie solche, welche von aussen nicht zugänglich sind. Dies erfolgt mit den Schlüsselwörter `private` und `public`. Alle Elemente die `public` deklariert werden, können von aussen verwendet werden.

```
class Product{

public:
  // Nun sind alle folgenden Deklarationen public
  string name;
  float price;

private:
  // Nun sind alle folgenden Deklarationen private
  int number;
};
```

Im obigen Beispiel kann also nur auf die Daten `name` sowie `price` von aussen zugegriffen werden.

Nun fragen Sie sich sicher, wie kann man denn von innerhalb einer Klasse Daten ändern? Die Antwort lautet Funktionen innerhalb einer Klasse.

6.1.1 Funktionen in einer Klasse

Eine Funktion kann innerhalb einer Klasse geschrieben werden:

```
class Product{

public:
  // Nun sind alle folgenden Deklarationen public
  string name;
  float price;

  // Deklaration einer Funktion in einer Klasse
  void displayNumber(){
    cout << "Number = " << number << endl;
  }

private:
  // Nun sind alle folgenden Deklarationen private
  int number;
};
```

Die Funktion `displayNumber()` ist im `public Teil` der Klasse definiert. Das heisst Sie kann von aussen aufgerufen werden. Da sie innerhalb der Klasse implementiert ist, kann sie auch auf die `private` Elemente der Klasse zugreifen. Nun muss allerdings zuerst eine Variable der Klasse `Products` angelegt werden, damit die Funktion `displayNumber` aufgerufen werden kann.

```
1 Product p;
2 p.displayNumber();
```

Zugriff auf die Funktion über eine Pointer Variable:

```
1 Product *p = new Product();
2 p->displayNumber();
```

6.1.2 Neue Begriffe

Für die weiteren Kapitel werden die folgenden neuen Begriffe verwendet:

- **Methode**
  Funktion innerhalb einer Klasse
- **Objekt**
  Variable einer Klasse (zb. `Product a;` `a` ist ein Objekt.)

6.1.3 Aufteilung Header- und Quellcode

Bei der Erstellung einer Klasse soll der Code jeweils in 2 Teile aufgeteilt werden. Deklaration und Implementierung. Die Deklaration beinhaltet alle internen Daten, sowie alle Methoden (nur die Deklaration). Die Implementierung der Methoden erfolgt im zweiten Teil.

Betrachten Sie die folgende Klasse `Product`.

```
class Product{

public:
  // Nun sind alle folgenden Deklarationen public
  string name;
  float price;

  // Deklaration einer Funktion in einer Klasse
  void displayNumber(){
    cout << "Number = " << number << endl;
  }

private:
  // Nun sind alle folgenden Deklarationen private
  int number;
};
```

Die Klasse kann auch so implementiert werden, dass die Deklaration und die Implementierung nicht am gleichen Ort sind.

```
// Deklaration
class Product{

public:
  string name;
  float price;
  void displayNumber();

private:
  int number;
};

// Implementierung
void Product::displayNumber(){
  cout << "Number = " << number << endl;
}
```

Jetzt haben wir die Möglichkeit, Deklarationsteil und Implementierungsteil in verschiedenen Files zu speichern.

Deklarationsteil

```
1  // Headerfile mit dem Namen product.h
2
3  #ifndef _PRODUCT_H
4  #define _PRODUCT_H
5
6  class Product{
7  public:
8    string name;
9    float price;
10   void displayNumber();
11
12 private:
13   int number;
14 };
15
16 #endif
```

Beim Deklarationsteil muss bei den Parametern nur der Datentyp angegeben werden. Die Bezeichnung kann weggelassen werden.

Implementierungsteil

```
1  // Quellcodefile mit dem Namen product.cpp
2
3  #include "product.h"
4
5  void Product::displayNumber(){
6    cout << "Number = " << number << endl;
7  }
```

Im Normalfall würde dann ein Programm mit einer Klasse aus mindestens drei Files bestehen.

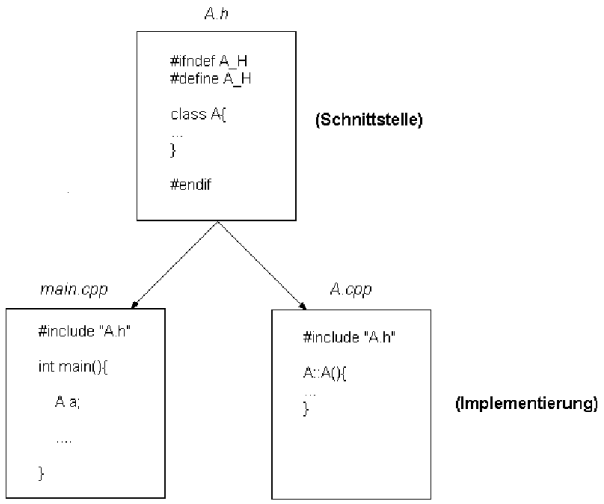


Abbildung 6.1: Aufteilung Deklaration und Implementierung

6.2 Konstruktoren und Destruktoren

6.2.1 Konstruktor

Ein Konstruktor ist eine Methode, welche die folgenden Eigenschaften besitzt:

- Keinen Rückgabewert
- Methodenname = Klassenname

Der Konstruktor wird jedesmal ausgeführt, wenn ein Objekt dieser Klasse erzeugt wird. Ein Objekt kann nur mit den vorhandenen Konstruktoren erzeugt werden. Das heisst, wenn ein Konstruktor existiert, welcher einen Parameter hat, so muss auch das Objekt mit einem Parameter erzeugt werden.

Wird kein Konstruktor definiert, so wird automatisch ein default Konstruktor erstellt. Dieser besitzt keine Parameter und beinhaltet auch keine Funktion. Also lediglich eine Methode ohne Parameter, die nichts macht.

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5    ...
6  public:
7    ...
8    A();                // Konstruktor ohne Parameter
9    A(int n);           // Konstruktor mit einem Parameter
10   A(int n, float x); // Konstruktor mit zwei Parametern
11   ...
12 };
```

Von der Klasse `A` können Objekte auf drei verschiedene Arten erzeugt werden. Es existieren drei Konstruktoren.

Bespiel:

Implementierung einer Klasse für Rechtecke.

```
#include <iostream>
using namespace std;

class Rectangle {
private:
    int width, height;
public:
    Rectangle(int,int);
    int area();
};

Rectangle::Rectangle (int a, int b){
    width = a;
    height = b;
}

int Rectangle::area(){
    return  width * height;
}

int main () {
    Rectangle rect(3,4);
    Rectangle rectb(5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

Dieser Code erzeugt den Output

```
rect area: 12
rectb area: 30
```

Initialisierungsliste

Im vorherigen Beispiel wurden dem Konstruktor 2 Werte übergeben. Diese beiden Werte wurden dann in den Attributen `width` und `height` gespeichert. Das heisst, die Attribute `width` und `height` wurden beim Erzeugen eines Objektes initialisiert.
Diese Art von Initialisierung funktioniert nicht immer. Betrachten Sie den folgenden Code:

```
1  class A{
2  private:
3    int& r;
4    const int c;
5  public:
6    A(int i);
7  };
8
9  A::A(int i){
10   r = i;
11   c = i;
12 }
```

Dieser Code funktioniert nicht, da sowohl `int& r` als auch `int const c` bereits bei der Initialisierung einen Wert erhalten müssen. Doch wann findet diese Initialisierung statt?
Die Initialisierung findet vor der Ausführung des Konstruktors statt - folglich können wir keine Referenzen und Konstanten übergeben. Aber halt - es gibt eine Lösung: die Initialisierungsliste.

```
1  class A{
2  private:
3    int& r;
4    const int c;
5  public:
6    A(int i);
7  };
8
9  A::A(int i)
10   : r(i), c(i){
11 }
```

▷ *Merke:*
Die Reihenfolge der Initialisierungen hängt von der Reihenfolge der Deklarationen (in der Klasse) ab. Wenn `A(int i) : c(i), r(i) {}` geschrieben wird, dann würde trotzdem `r` zuerst initialisiert werden.

Man sollte immer eine Initialisierungsliste verwenden, denn damit erspart man sich den Aufruf des Default Konstruktors. Dies kann einen großen Geschwindigkeitsvorteil bringen, denn es muss die Variable nicht erst mit sinnlosen Standardwerten gefüllt werden, sondern wir können gleich die richtigen Werte verwenden.

6.2.2 Copy Konstruktor

Mit Hilfe des Copy Konstruktors können Objekte kopiert werden. Es existiert ein default Copy Konstruktor, welcher als Parameter ein Objekt der selben Klasse besitzt.

```
1  class A{
2  public:
3    int m;
4    ...
5  };
6
7  int main(int argc, char **argv){
8    A obj; // Aufruf Default Konstruktor
9    A newObj(obj); // Aufruf Copy Konstruktor
10   return 0;
11 }
```

Der Copy Konstruktor kopiert nun die Werte (alle Attribute) des als Parameter übergebenen Objektes in das neu erzeugte Objekt.

Hat die Klasse jedoch dynamisch Speicher alloziert, funktioniert der default Copy Konstruktor nicht korrekt.

```
1  class A{
2  public:
3    int *m; // Irgendwo steht m = new int;
4    ...
5  };
6
7  int main(int argc, char **argv){
8    A obj; // Aufruf Default Konstruktor
9    A newObj(obj); // Aufruf Copy Konstruktor
10   return 0;
11 }
```

Jetzt existieren zwei Objekte der Klasse `A`. Jedes Objekt besitzt einen Pointer `*m`. Da der Copy Konstruktor die Werte aller Attribute kopiert, stehen in beiden Objekten in der Pointer Variable `*m` der gleiche Wert. Sie zeigen also auf die gleiche Adresse. Sie teilen sich eine dynamische `int` Variable. Wenn ein Objekt den Wert dieser Variable ändert, ist er gleich für alle Objekte der selben Klasse geändert.

Also muss der Copy Konstruktor so implementiert werden, dass dieser korrekt arbeitet.

```
1  class A{
2  public:
3    int *m;
4    A(const A &); // Deklaration Copy Konstruktor
5  };
6
7  // Implementierung Copy Konstruktor
8  A::A(const A &a){
9    m = new int;
10   *m = *(a.m);
11 }
12
13 int main(int argc, char **argv){
14   A obj; // Aufruf Default Konstruktor
15   A newObj(obj); // Aufruf Copy Konstruktor
16   return 0;
17 }
```

Jetzt wird beim Kopieren eines Objektes zuerst eine eigene dynamische `int` Variable erzeugt und dann den Wert des zu kopierenden Objektes übernommen.

Die folgende Grafik zeigt die Arbeitsweise des default Copy Konstruktors, sowie des korrekt überladenen Copy Konstruktors.

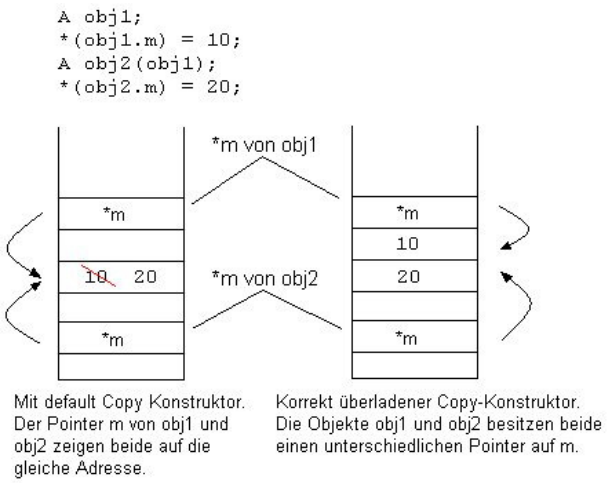


Abbildung 6.2: Copy Konstruktor

Beispiel:

Implementierung einer Klasse für Vektoren. Die Grösse des Vektors wird dem Konstruktor als Parameter übergeben. Der Konstruktor erzeugt dann einen Array, in welchem die Werte des Vektors gespeichert werden können.
Anhand dieses Beispiels soll das Implementieren des Copy Konstruktors gezeigt werden.

```
#include <iostream>
using namespace std;

class Vector {
public:
  Vector(int size); // Konstruktor
  Vector(const Vector &obj); // Copy Konstruktor
  // weitere Methoden
private:
  int *values;
  int size;
};

// Konstruktor
Vector::Vector(int size){
  this->size = size;
  values = new int[size];
}

// Copy Konstruktor
Vector::Vector(const Vector &obj){
  size = obj.size;
  values = new int[size];
  for (int i=0; i<size; i++){
    values[i] = obj.values[i];
  }
}
```

6.2.3 Destruktor

Der Destruktor ist (ähnlich wie der Konstruktor) eine Methode, welche die folgenden Eigenschaften besitzt:

- Keinen Rückgabewert
- Methodenname = ~Klassenname
- Keine Parameter

Der Destruktor wird jedesmal ausgeführt, wenn ein Objekt zerstört wird.

Der Destruktor wird implementiert, wenn in der Klasse dynamischer Speicher alloziert wird. Ansonsten kann ein Speicherloch entstehen.

Die folgende Klasse `A` besitzt eine dynamisch erzeugte float Variable. Jedesmal wenn ein Objekt erzeugt wird, wird im Konstruktor dynamisch eine float Variable angelegt.

```
1  class A{
2  private:
3      float *x;
4  public:
5      A();
6  };
7
8  A::A(){
9      x = new float;
10 }
```

Wird nun ein Objekt der Klasse `A` anlegt, wird somit auch eine dynamisch `float` Variable erzeugt. Wenn nun das Objekt gelöscht wird, so wird die Pointer Variable x auch gelöscht, nicht jedoch die Variable, auf die x zeigt. Diese bleibt im Speicher und kann nicht mehr verwendet werden. Jedesmal wenn ein Objekt der Klasse `A` zerstört wird, müssen auch alle dynamisch erzeugten Elemente innerhalb der Klasse zerstört werden.

```
1  class A{
2  private:
3      float *x;
4  public:
5      A();  // Konstruktor
6      ~A(); // Destruktor
7  };
8
9  // Implementierung Konstruktor
10 A::A(){
11     x = new float;
12 }
13
14 // Implementierung Destruktor
15 A::~A(){
16     delete x;
17 }
```

▷ *Merke:*
Wird in der Klasse dynamisch Speicher alloziert, so muss der Copy Konstruktor und der Destruktor implementiert werden.

Beispiel:

Implementierung einer Klasse für Vektoren. Die Grösse des Vektors wird dem Konstruktor als Parameter übergeben. Der Konstruktor erzeugt dann einen Array, in welchem die Werte des Vektors gespeichert werden können.
Anhand dieses Beispiels soll das Implementieren des Destruktors gezeigt werden.

```
1  #include <iostream>
2  using namespace std;
3
4  class Vector {
5  public:
6      Vector(int size); // Konstruktor
7      ~Vector(); // Destruktor
8      // weitere Methoden
9  private:
10     int *values;
11     int size;
12 };
13
14 // Konstruktor
15 Vector::Vector(int size){
16     this->size = size;
17     values = new int[size];
18 }
19
20 // Destruktor
21 Vector::~Vector(){
22     delete [] values;
23 }
```

6.3 Pointer auf Klassen

Wie bereits erwähnt können auch Pointer auf Objekte von Klassen erzeugt werden. Dies funktioniert folgendermassen:

```
1  // Pointer Variable - Ein Objekt wird nicht erzeugt!
2  Classname *pointerName;
3
4  // Erzeugen des Objektes
5  pointerName = new Classname(parameters);
```

oder in einer Zeile

```
1  // Pointer Variable + Objekt erzeugen in einer Zeile
2  Classname *pointerName = new Classname(parameters);
```

Der Zugriff auf Klassenattribute und Methoden funktioniert mit einem Pointer wiefolgt:

```
1  Classname *pointerName = new Classname(parameters);
2
3  // Zugriff auf Attribute
4  (*pointerName).attributeName ...;
5  // oder
6  pointerName -> attributeName ...;
7
8  // Zugriff auf Methoden
9  (*pointerName).methodName(...);
10 // oder
11 pointerName -> methodName(...);
```

6.4 Konstante Attribute

In C++ können auch Attribute einer Klasse das const-Schlüsselwort haben. Dies bedeutet, dass der Wert dieses Attributes nicht verwendet werden kann.
Die Konstante muss in der Initialisierungsliste definiert werden. Die Zuweisung mit dem Gleichheitszeichen bei der Deklaration oder im Konstruktor funktionieren nicht.

```
1 class A {
2 private:
3   const int value;
4 public:
5   A();
6 };
7
8 A::A() : value(100) {
9 }
```

6.5 Konstante Methoden

In C++ können auch Methoden einer Klasse das const-Schlüsselwort haben. Dies bedeutet, daß diese Methode das Objekt, über das sie aufgerufen wird, nicht verändert. Alle Attribute dieses Objektes dürfen in einer `const` Methode nur gelesen, nicht aber geschrieben werden.

```
class A{
private:
  int value;
public:
  // die Methode getValue veraendert keine Attribute
  // in einem Objekt der Klasse A
  int getValue() const;
};

int A::getValue() const {
  return value;
}
```

Der folgende Code zeigt einen verbotenen Zugriff auf ein Attribut innerhalb einer `const` Methode.

```
class A{
private:
  int value;
public:
  void doSomething() const;
};

void A::doSomething() const {
  ...
  value = 5; // Compilerfehler,
             // Attribute darf nur gelesen werden
  ...
}
```

6.6 Datenkapselung

In einer Klasse sollten alle Attribute so stark geschützt sein wie möglich. Das heisst, wenn ein Attribut als `private` deklariert werden kann, soll es auf keinen Fall als `public` deklariert werden.

Weiter soll ein Attribut, welches von aussen zugänglich sein muss, auch nicht als `public` deklariert werden. Es sollen entsprechende Methoden implementiert werden um das Attribut zu lesen und zu schreiben. Dies ermöglicht einen kontrollierten Zugriff auf die Attribute über eine definierte Schnittstelle.

Für jedes Attribut wird einen set Methode (schreiben) und eine get Methode (lesen) implementiert. Diese Art von Methoden sind immer nach dem gleichen Prinzip aufgebaut.

Get Methoden:

```
dataType getAttributename();
```

Set Methoden:

```
void setAttributename(dataType attributename);
```

Beispiel: Klasse Circle

Die folgende Klasse `Circle` besitzt ein Attribut `radius`. Im folgenden Beispiel werden für dieses Attribut die Get- und Setmethode implementiert.

```
class Circle {
private:
  float radius;
public:
  // Set Methode fuer das Element radius
  void setRadius(float);
  // Get Methode fuer das Element radius
  float getRadius();
};

void Kreis::setRadius(float r){
  radius = r;
}

float Kreis::getRadius(){
  return radius;
}
```

Die folgende Graphik zeigt nochmal den Zugriff auf private Attribute einer Klasse mit get und set Methoden.

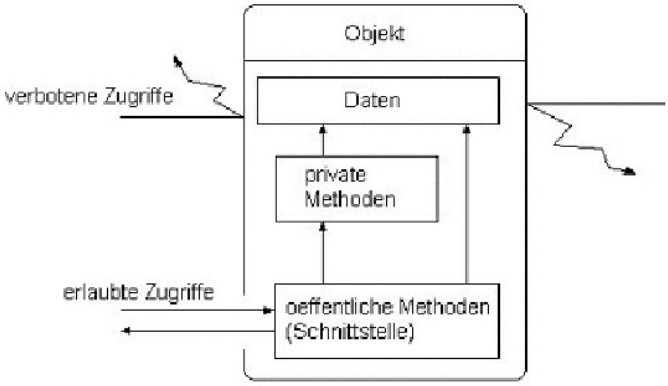


Abbildung 6.3: Datenkapselung

6.7 Überladen von Operatoren

Mit dem Schlüsselwort `operator` können in C++ Operatoren für Objekte von Klassen überladen werden. Oft ist dies sinnvoll, da eine bessere Lesbarkeit des Codes entsteht.

Betrachten Sie die folgende Klasse `Fraction`.

```
1  class Fraction {
2
3  private:
4    int numerator;
5    int denominator;
6
7   public:
8    Fraction(int numerator = 0, int denominator = 1);
9
10   // Get- und Setmethoden
11   int getDenominator();
12   void setDenominator(int denominator);
13   int getNumerator();
14   void setNumerator(int numerator);
15 };
```

Soll nun die Möglichkeit zur Addition von Brüchen bestehen, so kann

1. eine normale Methode zur Addition geschrieben werden

```
1 Fraction b1, b2, b3;
2 b3 = b1.add(b2); // Normale Methode
```

2. der Operator + überladen werden

```
1 Fraction b1, b2, b3;
2 b3 = b1 + b2; // Operator + ueberladen
```

Allgemein wird ein Operator wiefolgt überladen:

```
1 resultType operator@(parameter)
```

Das `@` wird mit dem entsprechenden Operator ersetzt.

Die Klasse `Fraction` kann mit allen wichtigen Operatoren für Brüche überladen werden.

```
class Fraction {

private:
  int numerator;
  int denominator;

 public:
  Fraction(int numerator = 0, int denominator = 1);

  // Get- und Setmethoden
  int getDenominator();
  void setDenominator(int denominator);
  int getNumerator();
  void setNumerator(int numerator);

  // Implementierung aller Bruch Operationen
  Fraction operator+ (const Fraction &f);
  Fraction operator- (const Fraction &f);
  Fraction operator* (const Fraction &f);
  Fraction operator/ (const Fraction &f);
};
```

In der Implementierung kann ein Bruch wiefolgt addiert werden:

$$\frac{z_1}{n_1} + \frac{z_2}{n_2} = \frac{z_1 \cdot n_2 + z_2 \cdot n_1}{n_1 \cdot n_2}$$

Diese Implementierung sieht wiefolgt aus:

```
Fraction Fraction::operator+ (const Fraction &f){
  Fraction result;

  result.numerator = numerator * f.denominator +
    f.numerator * denominator;
  result.denominator = f.denominator * denominator;

  return result;
}
```

6.7.1 Der Zuweisungsoperator

Mit Hilfe des Zuweisungsoperators = können Objekte kopiert werden. Jede Klasse besitzt einen default Zuweisungsoperator. Dieser funktioniert ähnlich wie der default Copy Konstruktor. Er kopiert alle Attribute in das neue Objekt. Dies führt zu Problemen, falls die Klasse dynamisch Speicher alloziert hat (gleiches Problem wie bei Copy Konstruktor).

```
1  class A{
2  public:
3    int m;
4    ...
5  };
6
7  int main(int argc, char **argv){
8    A obj; // Aufruf Default Konstruktor
9    A newObj;
10   newObj = obj; // Aufruf Zuweisungsoperator
11   return 0;
12 }
```

Der Zuweisungsoperator kopiert nun alle Werte (alle Attribute) des neuen Objektes in das alte Objekt.

Hat die Klasse jedoch dynamisch Speicher alloziert, funktioniert der default Zuweisungsoperator nicht korrekt.

```
1  class A{
2  public:
3    int *m; // Irgendwo steht m = new int;
4    ...
5  };
6
7  int main(int argc, char **argv){
8    A obj; // Aufruf Default Konstruktor
9    A newObj;
10   newObj = obj; // Aufruf Zuweisungsoperator - FEHLER
11   return 0;
12 }
```

Der folgende Code zeigt den korrekt implementierten Zuweisungsoperator

```
class A{
public:
  int *m;
  // Deklaration Zuweisungsoperator
  A operator = (const A &a);
};

// Implementierung Zuweisungsoperator
A A::operator = (const A &a){
  delete m; // Alter Speicher loeschen
  m = new int; // Neuer Speicher allozieren
  *m = *(a.m); // Inhalt kopieren
  return *this;
}

int main(int argc, char **argv){
  A obj; // Aufruf Default Konstruktor
  A newObj;
  newObj = obj; // Aufruf Zuweisungsoperator
  return 0;
}
```

▷ *Merke:*
Wird in einer Klasse dynamisch Speicher alloziert, so muss der Zuweisungsoperator überladen werden, ansonsten entstehen Memory Leaks.

▷ *Zusammenfassung:*
Wird in einer Klasse dynamisch Speicher alloziert, so müssen

- **Copy Konstruktor**
- **Destruktor**
- **Zuweisungsoperator**

implementiert werden!

6.8 This

Das Schlüsselwort `this` in einer Klasse repräsentiert die Adresse im Speicher desjenigen Objektes, das gerade verwendet wird.
Dies kann zum Beispiel verwendet werden, um Attribute einer Klasse zu verwenden, welche den gleichen Namen wie die Parameter haben.

```
1  class A{
2  private:
3    int attribute;
4  public:
5    void setAttribute(int attribute);
6  };
7
8  void A::setAttribute(int attribute){
9    this->attribute = attribute;
10   // attribute bezieht sich auf den Parameter
11   // this->attribute bezieht sich auf das Attribut
12   // der Klasse
13 }
```

6.9 Static

In einer Klasse können Methoden oder Variablen (und Objekte) mit dem Schlüsselwort `static` versehen werden.

```
1  class A{
2  public:
3    static int a;
4  };
```

`static` bezogen auf Variablen bedeutet, dass für alle Objekte der Klasse `A` ein `a` existiert. Normalerweise besitzt jedes Objekt ein `a`.
Wenn Bei einem Objekt der Wert von dem Attribut `a` geändert wird, ist der Wert von Attribut `a` auch bei allen anderen Objekten geändert.
Eine `static` definierte Variable muss immer initialisiert werden.
Auf eine static Variable kann auch direkt über die Klasse zugegriffen werden. Es ist nicht notwendig, zuerst ein Objekt der Klasse zu erzeugen.

Beispiel:

Eine Klasse `A`, welche eine `static` Variable besitzt.

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5  public:
6    static int a;
7  };
8
9  int A::a = 0; // Initialisierung
10
11 int main(int argc, char **argv){
12   A obj1, obj2;  // Erzeugen von 2 Objekten der Klasse A
13   obj1.a = 20;
14   obj2.a = 10;
15
16   cout << obj1.a << endl; // Ausgabe 10
17   // Da nur ein Attribut a existiert und dieses
18   // von obj2 zuletzt geaendert wurde.
19
20   A::a = 3; // Zugriff direkt ueber die Klasse
21
22   return 0;
23 }
```

`static` bezogen auf Methoden bedeutet, dass die Methode auch ohne Objekt der entsprechenden Klasse aufgerufen werden kann.

Beispiel:

Eine Klasse `A`, welche eine `static` Methode besitzt.

```
#include <iostream>
using namespace std;

class A{
public:
  static void display();
};

void A::display(){
  cout << "Hello World" << endl;
}

int main(int argc, char **argv){

  A::display(); // Aufruf ohne Objekt

  A a;
  a.display(); // Aufruf mit Objekt

  return 0;
}
```

▷ *Merke:*
Eine `static` deklarierte Methode kann *nicht* auf Variablen oder Methoden zugreifen, welche nicht `static` deklariert sind.

6.9.1 Singleton Pattern

Das Singleton Pattern ist ein Design Pattern, welches in der Softwareentwicklung eingesetzt wird. Es stellt sicher, dass zu einer Klasse nur ein Objekt existiert. Auf dieses Objekt wird ein globaler Zugriff ermöglicht.

Beispiel:

```
1  class Puffer{
2  private:
3    deque <string> data;
4
5    // Konstruktor muss private sein, damit ausserhalb
6    // der Klasse keine Objekte erstellt werden koennen
7    Puffer();
8
9    // static Object der Klasse Puffer - einziges
10   // existierendes Objekt dieser Klasse
11   static Puffer *instance;
12
13 public:
14   // Speichert eine Message
15   void put(string message);
16   // Entfernt eine Message
17   string get(bool &ok);
18
19   // Methode fuer den Zugriff auf das static Object
20   static Puffer* getInstance();
21 };
22
23 Puffer* Puffer::instance = 0;
24
25 Puffer::Puffer(){}
26
27 Puffer* Puffer::getInstance(){
28   if (instance == 0){
29     instance = new Puffer();
30   }
31   return instance;
32 }
33
34 void Puffer::put(string message){
35   data.push_back(message);
36 }
37
38 string Puffer::get(bool &ok){
39   if (data.size() <= 0){
40     ok = false;
41     return "";
```

```
42    }
43    ok = true;
44    string result = data.at(0);
45    data.pop_front();
46    return result;
47  }
```

6.10 Friend

6.10.1 Friend Funktionen

Von ausserhalb einer Klasse ist der Zugriff auf `private` Attribute und Methoden nicht erlaubt.
Jetzt kann eine ganz normale Funktion implementiert werden, welche dann innerhalb eine Klasse als `friend` deklariert wird. Das heisst, diese Funktion hat dann Zugriff auf alle `private` Attribute und Methode innerhalb dieser Klasse.

```
#include <iostream>
using namespace std;

class A{
private:
  int n;
public:
  A(int);
  // Funktion als friend deklarieren
  friend void display(A);
};

A::A(int n){
  this->n = n;
}

// Diese Funktion hat Zugriff auf die privaten Elemente
// von A, da sie in der Klasse als friend deklariert
// wurde
void display(A obj){
  cout << obj.n << endl;
}
```

Beispiel:

Implementierung einer Klasse für Rechtecke, sowie einer Funktion, welche auf die privaten Elemente dieser Klasse zugreifen kann.

```
#include <iostream>
using namespace std;

class Rectangle{
private:
  int width, height;
public:
  void setValues (int, int);
  int area();
  friend Rectangle duplicate(Rectangle);
};

void Rectangle::setValues (int a, int b){
  width = a;
  height = b;
}

// Diese Funktion wurde als friend in der
// Klasse Rectangle deklariert
Rectangle duplicate (Rectangle rectparam){
  Rectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int Rectangle::area(){
  return width * height;
}

int main () {
  Rectangle rect, rectb;
  rect.setValues (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

6.10.2 Friend Klassen

Das gleiche Prinzip existiert auch bei Klassen. Eine Klasse kann als `friend` deklariert werden, so hat diese Zugriff auf die privaten Elemente der Klasse.

```
#include <iostream>
using namespace std;

class A{
private:
  int n;
public:
  A(int);
  friend class B; // Klasse als friend deklarieren
};

class B{
  // Die Klasse B hat nun Zugriff auf die privaten
  // Elemente der Klasse A
};
```

Beispiel:

Implementierung von 2 Klassen `Square` und `Rectangle`, wobei die Klasse `Rectangle` auf die privaten Elemente der Klasse `Square` zugreifen kann.

```
#include <iostream>
using namespace std;

class Square;

class Rectangle{
private:
  int width, height;
public:
  int area();
  void convert(Square a);
};

class Square {
private:
  int side;
public:
  void setSide(int a);
  friend class Rectangle;
};

void Square::setSide(int a){
  side = a;
}

void Rectangle::convert(Square a) {
  width = a.side;
  height = a.side;
}

int Rectangle::area(){
  return width * height;
}

int main () {
  Square sqr;
  Rectangle rect;
  sqr.setSide(4);
  rect.convert(sqr);
  cout << rect.area() << endl;
  return 0;
}
```

6.11 Vererbung

Eine der wichtigsten Eigenschaften in der objektorientierten Programmierung ist die Vererbung. Klassen können von anderen Klassen abgeleitet werden. Das heisst, das die abgeleitete Klasse die Eigenschaften der Superklasse (Klasse von der abgeleitet wird) übernimmt.

Die folgende Abbildung zeigt eine Vererbungshirarchie.

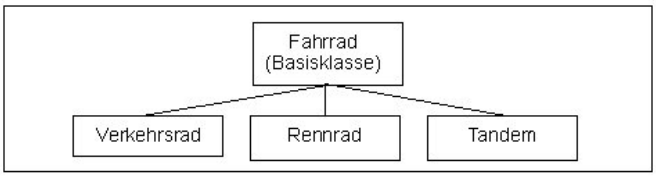


Abbildung 6.4: Vererbungshierarchie von Fahrrädern

Eine Vererbung wird folgendermassen deklariert:

```
1 class B : public A
```

Dies bedeutet, dass die Klasse `B` von der Klasse `A` abgeleitet ist.
Die Klasse `B` kann nun auf die Attribute und Methoden der Klasse `A` zugreifen, falls diese in `A` als `public` deklariert wurden. Auf `private` deklarierte Elemente kann auch eine Subklasse nicht zugreifen.

Beispiel:

Eine Klasse `Man`, welche von der Klasse `Person` abgeleitet ist.

```
#include <iostream>
using namespace std;

class Person{
public:
    int age;
    Person();
    void displayAge();
};

Person::Person(){
    cout << "Konstruktor Person" << endl;
}

void Person::displayAge(){
    cout << age << endl;
}

class Man : public Person{
public:
    Man();
};

Man::Man(){
    cout << "Konstruktor Man" << endl;
}

int main(int argc, char **argv){
    Man m;
    // Kann auf das Attribut age
    // der Superklasse zugreifen
    m.age = 12;

    // Kann auf die Methode displayAge
    // der Superklasse zugreifen
    m.displayAge();

    return 0;
}
```

▷ *Merke:*
Da die Klasse `Man` auf die Elemente der Klasse `Person` zugreift, muss also auch ein Objekt von `Person` erzeugt werden, falls ein Objekt von `Man` erzeugt wird.
Objekte der Klasse, welche zuoberst in der Vererbungshierarchie stehen werden zuerst erzeugt.

Hier wird also zuerst der Konstruktor von `Person` und dann der von `Man` ausgeführt. Beim Destruktor ist es genau umgekehrt. Zuerst der Destruktor des in der Vererbungshierarchie zuuntersete Elemente, und dann sequentiell nach oben.

Der Output sieht dann folgendermassen aus:

```
1 Konstruktor Person
2 Konstruktor Man
3 12
```

6.11.1 Begriffe

Eine abgeleitete Klasse nennt man *Subklasse*. Die Klasse von der abgeleitet wurde, ist die *Superklasse*.

6.11.2 Protected

Im Moment kennen wir zwei verschiedene Zugriffsarten: `public` und `private`. Nun ist es manchmal sinnvoll, dass von ausserhalb der Klasse kein Zugriff erfolgen darf, jedoch innerhalb der eigenen Klasse sowie in allen Subklassen. Ist dies der Fall, kann ein Element als `protected` deklariert werden. So haben alle Subklassen (und dessen Subklassen, u.s.w.) dieser Klasse Zugriff.

Die folgende Tabelle zeigt alle möglichen Zugriffsarten.

Zugriff	Innerhalb Klasse	In abgeleiteten Klassen	Ausserhalb Klasse
`private`	JA	NEIN	NEIN
`protected`	JA	JA	NEIN
`public`	JA	JA	JA

Beispiel:

Eine Klasse mit drei Elementen: Eine `public` Variable c, eine `protected` Variable b und eine `private` Variable c.

```
1  class A{
2
3  private:
4      int c;
5
6  protected:
7      int b;
8
9  public:
10     int a;
11
12 };
```

6.11.3 Konstruktoren und Vererbung

Wie Sie gesehen haben, wird auch immer der Konstruktor der Superklasse ausgeführt. Eine Superklasse kann jedoch mehrere Konstruktoren besitzen. Welcher wird nun ausgeführt? Dies muss immer in der Subklasse angegeben werden. Wird nichts angegeben, so wird automatisch der Konstruktor ohne Parameter aufgerufen. Falls dieser jedoch nicht existiert ergibt es einen Fehler beim Compilieren.

Der folgende Code verursacht einen Fehler, da in der Superklasse kein Konstruktor ohne Parameter exisitert.

```
1  class A{
2  public:
3      A(int);
4  };
5
6  class B : public A{
7  public:
8      B(int);
9  };
10
11 A::A(int m){
12 }
13
14 B::B(int n){   // Ergibt einen Compiler fehler!!!
15 }
```

Beim Konstruktor der Subklasse muss also angegeben werden, welcher Konstruktor der Superklasse verwendet werden soll.
In der Klasse `A` existiert ein Konstruktor mit einem `int` Parameter. Also müssen wir im Konstruktor der Subklasse dies angeben. Die korrekte Implementierung des Konstruktors der Klasse `B` lautet dann:

```
B::B(int n) : A(n){
}
```

6.11.4 Methoden überschreiben

Existiert in der Superklasse eine Methode, so kann diese in der Subklasse überschrieben werden. Dies bedeutet, dass die Methode in der Subklasse nochmal implementiert wird. Sie hat den gleichen Namen, gleicher Rückgabewert und die gleichen Parameter.

Beispiel:

Eine Klasse `Rectangle` und eine Klasse `Square`, welche von `Rectangle` abgeleitet ist. Beide Klassen implementieren die Methode `area()`.

```
#include <iostream>
using namespace std;

class Rectangle{
protected:
    int width, height;
public:
    Rectangle(int, int);
    int area();
};

Rectangle::Rectangle(int w, int h){
    width = w;
    height = h;
}

int Rectangle::area(){
    return width * height;
}

class Square : public Rectangle{
public:
    Square(int);
    int area();
};

Square::Square(int side) : Rectangle(side, side){
}

int Square::area(){
    return width * width;
}
```

6.11.5 Methoden der Superklasse aufrufen

Betrachten Sie die im folgenden Beispiel die Klassen `A` und `B`. Die Klasse `A` besitzt eine Methode `xy()`. Diese Methode wird in der Klasse `B` überschrieben.

```
1  class A{
2  public:
3    void xy();
4  };
5
6  void A::xy(){
7    cout << "xy from class A" << endl;
8  }
9
10 class B : public A{
11 public:
12   void xy();
13 };
14
15 void A::xy(){
16   cout << "xy from class B" << endl;
17 }
```

Soll nun in der Methode `xy` der Klasse `B` die überschriebene Methode der Superklasse aufrufen, d.h. die Methode `xy` der Klasse `A`. Dies geschieht in der Methode `xy` der Klasse `B` an einer beliebigen Stelle mit folgendem Aufruf:

```
1  A::xy();
```

Mit diesem Befehl wird die `xy` Methode der Superklasse aufgerufen.

6.11.6 Virtual

Gegeben sind die folgenden Klassen `A` und `B`.

```
#include <iostream>
using namespace std;

class A{
public:
    void display();
};

class B : public A{
public:
    void display();
};

void A::display(){
    cout << "A" << endl;
}

void B::display(){
    cout << "B" << endl;
}
```

Pointers auf Basisklassen

Als erstes wird ein Pointer auf die Klasse `A` erstellt.

```
A *ptr;
```

Nun soll `ptr` auf ein Objekt der Klasse `B` zeigen.

```
ptr = new B();
```

Dies funktioniert, da ein `B` Objekt auch ein `A` Objekt ist.
Nun wird auf diesem neu erzeugten Objekt die Methode `display()` aufgerufen.

```
ptr->display();
```

Nun sollte die `display` Methode der Klasse `B` ausgeführt werden, da `ptr` auf ein `B` Objekt zeigt. Dies ist aber nicht der Fall, da `ptr` vom Typ `A*` ist. Es wird also die falsche Methode ausgeführt.

Virtuelle Methoden

Um den obigen Fehler zu beheben, muss die Methode `display()` in der Klasse `A` auf `virtual` gesetzt werden.
`virtual` bedeutet, dass erst während der Laufzeit entschieden wird, welche Methode aufgerufen werden soll.
Bei der Vererbung sollte man sich also an folgende Regel halten:

- Nicht virtuelle Methoden dürfen nicht überschrieben werden
- Falls dies zur Implementierung der Subklasse notwendig ist, sollte nicht abgeleitet werden.

Der vorherige Code sieht verbessert folgendermassen aus:

```
#include <iostream>
using namespace std;

class A{
public:
  // Wird neu als virtuelle Methode deklariert
  virtual void display();
};

class B : public A{
public:
  void display();
};

void A::display(){
  cout << "A" << endl;
}

void B::display(){
  cout << "B" << endl;
}
```

Virtuelle Destruktoren

Das gleiche wie für Methoden gilt auch für Destruktoren.

```
A *ptr = new B();
delete ptr;
```

Wäre hier der Destruktor in der Klasse `A` nicht `virtual`, dann würde beim `delete` Befehl nur der Destruktor der Klasse `A` aufgerufen werden, was falsch wäre. Es müsste zuerst der Destruktor von `B` aufgerufen werden.
Klassen, bei denen der Destruktor nicht `virtual` ist, sollten nicht abgeleitet werden.

6.11.7 Abstrakte Klassen

Abstrakte Klassen sind Klassen, von denen keine Objekte erzeugt werden können. Sie dienen nur dazu, Schnittstellen, Methoden bzw. Eigenschaften vorzugeben, welche dann in den abgeleiteten Klassen implementiert werden.
Eine Klasse ist abstrakt, falls mindestens eine Methode der Klasse mit 0 initialisiert wird. Eine solche Methode wird als *rein virtuelle Methode* bezeichnet.

Beispiel:

Eine Klasse `SuperClass`, welche eine rein virtuelle Methode besitzt.

```
1  #include <iostream>
2  using namespace std;
3
4  class SuperClass{
5
6  public:
7
8    virtual void display() = 0;  // rein virtuelle Methoden
9
10   // Hier koennen noch mehr rein virtuelle oder normale
11   // Methoden deklariert werden.
12
13 };
```

Diese Klasse ist abstrakt, da sie mindestens eine Methode besitzt, welche rein virtuell ist. Wird die Klasse nun als Superklasse verwendet, dann muss in allen Subklassen die Methode `display()` implementiert werden. Ansonsten ist die Subklasse auch abstrakt.

An diesem Beispiel sollte auch klar werden, warum keine Objekte einer abstrakten Klasse erzeugt werden können. Würde es funktionieren, was passiert dann beim Aufruf der Methode `display()` ? Diese ist ja gar nicht implementiert.

6.12 Klassendiagramme

Ein Klassendiagramm ist eine graphische Darstellung von Klassen sowie der Beziehungen zwischen diesen Klassen.
Klassendiagramme werden meistens in der Notation der Unified Modelling Language (UML) dargestellt.

6.12.1 Klassen

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse tragen oder zusätzlich auch Attribute und Methoden. Klassenname, Attribute und Methoden werden jeweils mit einem horizontalen Strich getrennt. Oberhalb des Klassennames können zusätzliche Schlüsselwörter (z.b. abstrakt) stehen.

Die Sichtbarkeit von Attributen und Methoden wird wie folgt gekennzeichnet:

- \+ für public
- – für private
- # für protected

Beispiel:

Für die Klasse `Circle` kann das Klassendiagramm wie folgt aussehen:

```
1 class Circle {
2 private:
3   double radius;
4 public:
5   void setRadius(double radius);
6   double getRadius();
7   double getArea();
8 };
```

Klassenname

Abbildung 6.5: Klassendiagramm mit Klassenname

Klassenname und Attribute

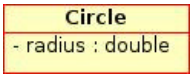


Abbildung 6.6: Klassendiagramm mit Klassenname und Attribute

Klassenname, Attribute und Methoden

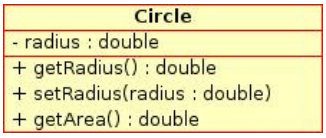


Abbildung 6.7: Klassendiagramm mit Klassenname, Attribute und Methoden

6.12.2 Beziehungen

Vererbung

Eine Vererbung wird als durchgezogene Linie zwischen den beteiligten Klassen dargestellt. Bei der Superklasse wird eine geschlossene, nicht ausgefüllte Pfeilspitze gezeichnet.

Beispiel:

Eine Klasse `Circle`, welche von der Klasse `Shape` abgeleitet ist.

```
1  class Shape {
2  public:
3    virtual double getArea() = 0;
4  };
5
6  class Circle : public Shape {
7  private:
8    double radius;
9  public:
10   void setRadius(double radius);
11   double getRadius();
12   double getArea();
13 };
```

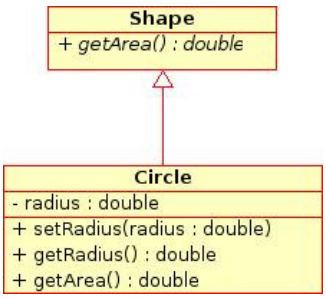


Abbildung 6.8: Klassendiagramm mit Vererbung

Assoziation

Eine Assoziation beschreibt eine Beziehung zwischen zwei Klassen.

Abbildung 6.9: Klassendiagramm mit Assoziation

Komposition und Aggregation

Häufig wird eine Klasse aus Teilen zusammengesetzt, welche wiederum Klassen sind. Dabei werden zwei Arten unterschieden:

- **Aggregation**
  Die Teile können auch nach der Zerstörung des Ganzen weiter existieren. Dies wird durch eine leere Raute an demjenigen Ende der Linie dargestellt, an welchem die Klasse als Ganzes steht.

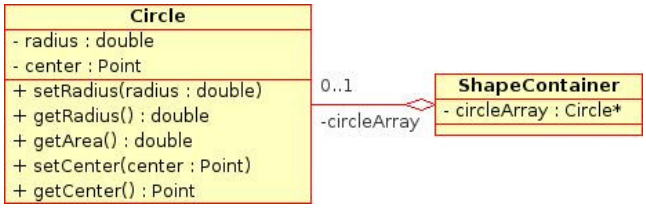


Abbildung 6.10: Klassendiagramm mit Aggregation

Wird ein Objekt der Klasse `ShapeContainer` zerstört, wo können die `Circle` Objekte in diesem Container weiterexistieren.

- **Komposition**
  Die Teile können nach der Zerstörung des Ganzen nicht weiter existieren. Dies wird durch eine gefüllte Raute an demjenigen Ende der Linie dargestellt, an welchem die Klasse als Ganzes steht.

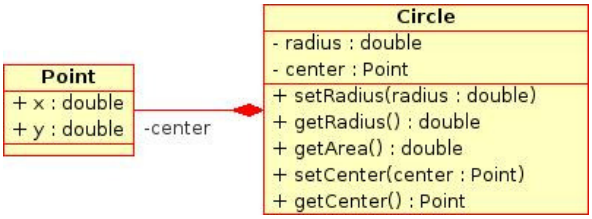


Abbildung 6.11: Klassendiagramm mit Komposition

Wird ein Objekt der Klasse `Circle` zerstört, so wird auch das Objekt `center`, welches als Attribut in der Klasse `Circle` ist, zerstört.

6.13 Mehrfachvererbung

C++ erlaubt bei der Vererbung von Klassen die Angabe von mehreren Superklassen. Die Syntax bei der Mehrfachvererbung sieht folgendermassen aus:

```
class A{
    ...
};

class B{
    ...
};

class C : public A, public B{
    ....
};
```

Bei der Mehrfachvererbung gelten nun die folgenden Punkte:

- Die Subklasse wird wie bisher deklariert. Nach dem Doppelpunkt werden dann die Superklassen, getrennt mit einem Komma, angegeben.
- Die Konstruktoren der Basisklassen werden in der Reihenfolge ihrer Deklaration ausgeführt.
- Die Subklasse hat nun wie bei der einfachen Vererbung Zugriff auf die Attribute und Methoden der Superklasse.

6.13.1 Namenskonflikte

Gegeben ist das folgende Beispiel:

```
class A{
protected:
  int n;
};

class B{
protected:
  int n;
};

class C : public A, public B{
public:
  C();
};

C::C(){
  n = 5; // Ergibt einen Fehler!
}
```

Hier wird die Klasse `C` von der Klasse `A` und `B` abgeleitet. Sowohl in der Klasse `A` wie auch `B` befindet sich ein `protected int n`. Wenn nun in der Klasse `C` auf das Attribut `n` zugegriffen wird, ergibt dies einen Compilerfehler, da nicht genau spezifiziert ist, ob es sich um das `n` von `A` oder `B` handelt.
Es muss also genau spezifiziert werden, mit welchem `n` gearbeitet werden soll. Der korrigierte Code sieht dann folgendermassen aus:

```
1  class A{
2  protected:
3    int n;
4  };
5
6  class B{
7  protected:
8    int n;
9  };
10
11 class C : public A, public B{
12 public:
13   C();
14 };
15
16 C::C(){
17   A::n = 5;   // n von A
18   B::n = 5;   // n von B
19 }
```

6.13.2 Virtuelle Basisklassen

In C++ darf eine Klasse nicht direkt von ein und derselben Klasse mehrfach abgeleitet werden. Dadurch würden Namenskonflikte entstehen, da jeder Name doppelt vorkommen würde.

```
1 class A {...};
2 class B : public A {...};
3 class C : public A {...};
4 class D : public B, public C {...};
```

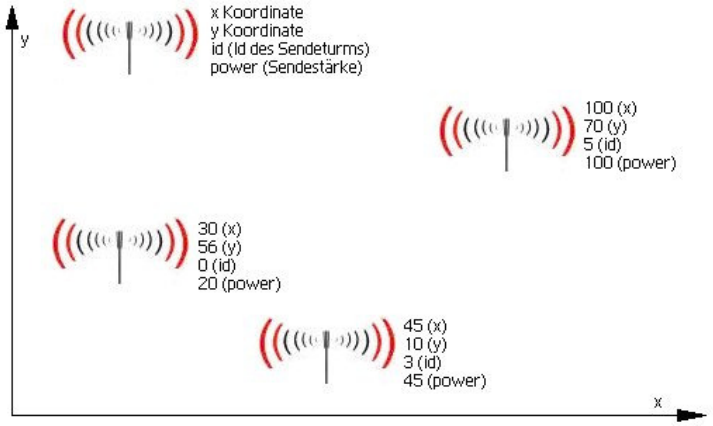
Sollen alle Daten der Klasse `A` nur einmal in der Klasse `D` vorhanden sein, so müssen die Klassen `B` und `C` virtuell von der Klasse `A` abeleitet werden.

```
1 class A {...};
2 class B : virtual public A {...};
3 class C : virtual public A {...};
4 class D : public B, public C {...};
```

6.14 Aufgaben

6.14.1 Cell Phone

Gegeben sind eine Menge von Sendetürmen für Mobiltelefone in der Ebene.



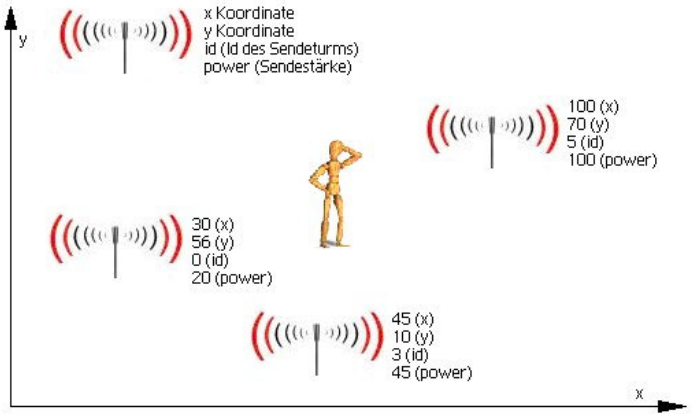
Jeder Sendeturm besitzt die folgenden Attribute:

- x Koordinate
- y Koordinate
- Identifikationsnummer
- Sendestärke

Die Klasse für einen Sendeturm sieht wie folgt aus:

```
1 class Transmitter {
2 public:
3   int x, y;
4   int id;
5   int power;
6 };
```

Nun befindet sich eine Person in der Reichweite der Sendetürme:



Nun soll die folgende Funktion entwickelt werden:

```
1   Transmitter *tms, int nTransmitter, int x, int y);
```

Diese Methode soll aus einer Menge von Sendetürmen, gegeben durch `tms` (Array von Transmitter Objekten) und `nTransmitter` (Grösse des Arrays), die zwei der Person am nächsten liegenden Sendetürme bestimmen und dann von diesen zwei die ID desjenigen zurückgeben, welcher die stärkere Sendeleistung hat. Die Koordinaten der Person sind die Parameter `x` und `y`. Es kann davon ausgegangen werden, dass der Transmitter Array mindestens zwei Eintärge besitzt.

6.14.2 Geometrische Figuren - Vererbung

Gegeben ist das folgende Klassendiagramm. Implementieren Sie die Klassen und schreiben Sie ein kurzes Testprogramm um die korrekte Funktionsweise zu demonstrieren. Halten Sie sich bei der Implementierung an das Klassendiagramm und führen Sie keine zusätzlichen Attribute oder Methoden ein.

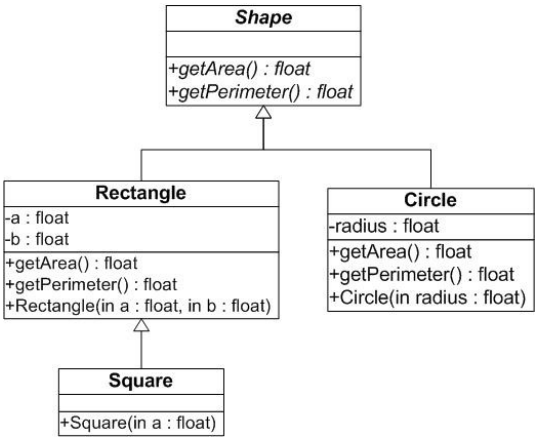


Abbildung 6.12: Klassendiagramm für geometrische Figuren

6.14.3 BigInt - Operatoren überladen

In dieser Aufgabe soll eine Klasse `BigInt` entwickelt werden, welche zur Speicherung von sehr grossen ganzen positiven Zahlen verwendet werden kann.
Implementieren Sie die Klasse gemäss dem folgenden Klassendiagramm und Headerfile.

6.14.4 Diskrete Fourier Transformation

Kapitel 7

Fortgeschrittene Themen

Nothing is ever as simple as it looks.
Murphy's Law

7.1 Templates

Oft kommt es vor, dass Funktionen mehrmals definiert werden müssen, da diese für verschiedene Datentypen verwendet werden, oder eine Klasse mit verschiedenen Attribut Typen eingesetzt werden soll (z.b. Stack mit `int`, `double`, ...).
Dies führte häufig dazu, dass der Code lediglich kopiert und der entsprechende Datentyp geändert wird. Ist der Code fehlerhaft, so muss der Fehler an meherern Stellen behoben werden.
Aus diesem Grund wurden in C++ Templates eingeführt. Mit Templates können in Funktionen und Klassen Typen eingesetzt, welche erst bei Gebrauch bestimmt werden.

7.1.1 Funktionen Templates

Gegeben ist die folgende Funktion:

```
1 int max(int a, int b){
2   return (a>b ? a:b);
3 }
```

Diese Funktion ist im Moment nur für `int` Werten brauchbar. Soll die Funktion mit `double` Werten verwendet werden können, so muss die Funktion überladen werden.
Mit Templates kann nun auch eine Funktion geschrieben werden, welche für beliebige Datentypen verwendet werden kann.

```
1 template <class T>
2 T max(T a, T b){
3   return (a>b ? a:b);
4 }
```

`T` ist kein bestimmter Datentyp. An seiner Stelle kann ein beliebiger Datentyp verwendet werden. Wenn diese Funktion aufgerufen wird, muss bestimmt werden, was für ein Datentyp `T` ist.

```
// Fuer T wird double eingesetzt
double x = max<double>(10.3,10.2);
```

In diesem Fall könnte die Angabe des Datentypes auch weggelassen werden, da beide Parameter vom Typ `double` sind, findet der Compiler automatisch den richtigen Datentyp.
Natürlich können auch mehrere generische Typen verwendet werden.

```
template <class T, class U, class Z>
T functionName(U a, Z b) {
  // Implementation
}
```

7.1.2 Klassen Templates

Es gibt auch die Möglichkeit Templates für Klassen zu schreiben. Diese können dann für Attribute der Klasse verwendet werden.

```
template <class T>
class Stack{
private:
  int actual, max;
  T *elements;
public:
  Stack(int);
  ~Stack();
  Stack(const Stack &obj);
  Stack operator = (const Stack &obj);
  bool push(T);
  bool pop(T&);
  bool top(T&);
};

// Implementierung des Konstruktors
template <class T>
Stack<T>::Stack(int m){
  actual = 0;
  max = m;
  elements = new T[max];
}
```

Um nun ein Objekt der Klasse Stack zu erzeugen, kann folgendes Statement verwendet werden:

```
1 // Stack mit 100 Elementen vom Typ Integer
2 Stack <int>s(100);
```

oder

```
1 Stack <int> *s = new Stack<int>(100);
```

Da es sich bei Templates nur um Muster für die Erzeugung wirklicher Datentypen handelt, muss für den Compiler ber der Übersetzung eines Source Code, welcher das Template benutzt, der Template Source Code sichtbar sein. Am einfachsten wird das garantiert, indem aller zur Klasse gehörender Code im Headerfile untergebracht wird.

7.2 Exception Handling

In C++ gibt es die Möglichkeit Exceptions zur Fehlerbehandlung zu verwenden. Wenn in einer Funktion oder Methode ein Fehler erkannt wird, wird eine Exception ausgelöst, welche dem Benutzer den Fehler mitteilt.

In der folgenden Funktion wird geprüft, ob der übergebene Pointer nicht gleich 0 ist.

```
1  void test(Point *p){
2    if (p==0) return;
3    // Implementation
4  }
```

Somit ist sichergestellt, dass der Pointer `p` nur verwendet wird, falls er auf ein gültiges Object zeigt. Der Aufrufer jedoch bekommt nicht mit, dass die Funktion nicht abgearbeitet wurde, weil der Pointer `p` auf `0` zeigt.
Unter der Verwendung von Exceptions würde die Funktion wiefolgt aussehen:

```
1  #include <exception>
2  using namespace std;
3
4  void test(Point *p) throw (exception) {
5    if (p == 0){
6      throw exception("error message");
7    }
8    // Implementation
9  }
```

`exception` ist eine ganz normale Klasse und dient als Basis für Ausnahmefehler. Mit dem Schlüsselwort `throw` wird eine Exception ausgelöst. Die ausgelöste Exception kann vom Aufrufer abgefangen werden. Somit kann der Aufrufer in einem Fehler entsprechend reagieren. Wird der Fehler vom Aufrufer nicht abgefangen, so wird die Ausführung des Programmes gestoppt.
3mm

```
1  void test(Point *p) throw (exception) {
2    if (p == 0){
3      throw exception("error message");
4    }
5    // Implementation
6  }
7
8  // Aufruf
9  int main(int argc, char **argv){
10   try {
11     test(0);
12   }
13   catch (exception &ex){
14     cout << ex.what() << endl;
```

```
15      // Fehlerbehandlung
16    }
17    return 0;
18  }
```

Eine Funktion kann verschiedene Arten von Exceptions auslösen. Auch diese können vom Aufrufer unterschieden werden. Dazu werden lediglich mehrere `catch` Statements verwendet.

```
1   void test(Point *p) throw (exception) {
2     if (p == 0){
3       throw exception("exception");
4     }
5     // some code
6     int *n = new int[100];
7     n = 0; // JUST FOR ERROR SIMULATION !
8     if (n == 0){
9       throw bad_alloc("bad_alloc");
10    }
11    // Implementation
12  }
13
14  int main(int argc, char **argv){
15    try {
16      test(new Point());
17    }
18    catch (bad_alloc &ex_ba){
19      cout << ex_ba.what() << endl;
20    }
21    catch (exception &ex){
22      cout << ex.what() << endl;
23    }
24  }
```

Die Klassen, welche zur Fehlerbehandlung zur Verfügung stehen, sind die folgenden:

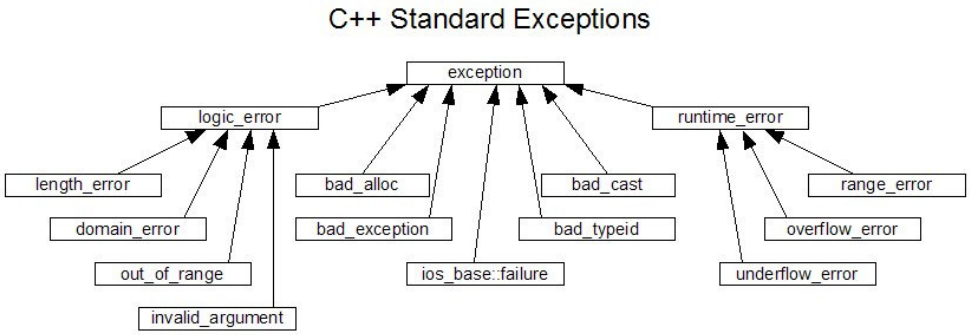


Abbildung 7.1: C++ Standard Exceptions

Natürlich können eigene Fehlerklassen implementiert werden.

7.3 Ein- und Ausgabe mit Files

C++ unterstützt Files mit den folgenden Klassen:

- `ofstream` File für Schreibeoperationen
- `ifstream` File für Leseoperationen
- `fstream` File für Lese- und Schreibeoperationen

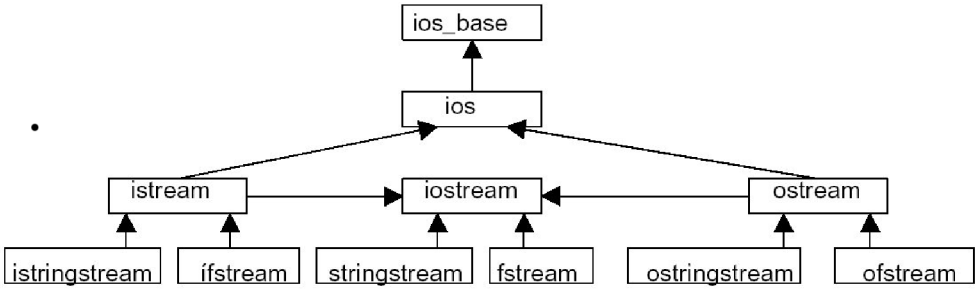


Abbildung 7.2: Klassen für die Ein- und Ausgabe mit Files

Möchten wir also mit Files arbeiten, müssen wir die Headerdatei `fstream.h` einbinden. Da `iostream.h` bereits in `fstream.h` verwendet wird, brauchen wir diese nicht mehr speziell einzubinden.

7.3.1 File lesen

Um von einem File zu lesen, muss zuerst ein Objekt der Klasse `ifstream` angelegt werden. Dem Konstruktor können wir als Parameter den Names des Files mitgeben.

```
ifstream file("xy.txt");
```

Als nächstes sollte überprüft werden, ob das File erfolgreich geöffnet werden konnte.

```
if (!file.is_open()) // Fehlerbehandlung
```

Ist das File erfolgreich geöffnet, kann mit folgenden Methoden gelesen werden:

- `bool get(char ch);`
  Liest das nächste Zeichen vom File in die Variable `ch`. Ist das File bereits am Ende, so liefert die Methode den Wert `false`.
- `file >> string st;`
  Liest alle Zeichen von der aktuellen Position bis zum nächsten Leerzeichen in die Variable `st`.

Wahlfreier Zugriff

Zur Positionierung in Files existieren die folgenden Funktionen:

`seekg()`	setzt eine Leseposition
`seekp()`	setzt eine Schreibposition
`tellg()`	liefert eine Leseposition
`tellp()`	liefert eine Schreibposition

Beispiel:

Der folgende Code kopiert alle Zeichen in den String `line` bis ein Leerzeichen kommt. Dann wird `line` auf den Bildschirm geschrieben. Dies passiert solange, bis das File komplett gelesen wurde.

```
#include <string>
#include <fstream>
using namespace std;

int main(int argc, char **argv){

    ifstream file;
    file.open("xy.txt");

    if (!file.is_open()) exit(1);

    string line;

    while (!file.eof()){
        file >> line;
        cout << line << endl;
    }

    file.close();

    return 0;
}
```

Beispiel:

Implementierung des Unix Programmes `cat`.

```
#include <fstream>
using namespace std;

int main(int argc, char **argv){

    ifstream file(argv[1]);

    if (!file.is_open()){
        cout << "File not found!" << endl;
        exit(1);
    }

    char ch;
    while (file.get(ch)){
        cout << ch;
    }

    file.close();

    return 0;
}
```

Beispiel:

Einlesen eines kompletten Files in einen Buffer.

```
#include <fstream>
using namespace std;

int main () {

    char *buffer;
    long size;
    ifstream file ("harmonisch.cpp", ios::in|ios::binary|ios::ate);
    size = file.tellg();
    file.seekg (0, ios::beg);
    buffer = new char[size];
    file.read (buffer, size);
    file.close();

    cout << "the complete file is in a buffer" << endl;

    for (int i=0; i<size; i++){
        cout << buffer[i];
    }

    delete[] buffer;
    return 0;
}
```

7.3.2 File schliessen

Wenn alle Operationen (lesen oder schreiben) auf einem File ausgeführt wurden, sollte die Methode `close()` aufgerufen werden. Danach kann das Objekt für ein neues File verwendet werden.

7.3.3 File schreiben

Um in ein File zu schreiben, muss zuerst ein Objekt der Klasse `ofstream` angelegt werden. Dem Konstruktor können wir als Parameter den Namen des Files mitgeben.

```
ofstream file("xy.txt");
```

Als nächstes sollte überprüft werden, ob das File erfolgreich angelegt werden konnte.

```
if (!file.is_open())   // Fehlerbehandlung
```

Ein Fehler kann der Fall sein, wenn Sie keine Schreibrechte besitzten.

Ist das File erfolgreich geöffnet, kann mit folgenden Methoden gelesen werden:

- `void put(char ch);`
  Schreibt das Zeichen von `ch` in das File.
- `file << ch;`
  Schreibt das Zeichen von `ch` in das File (arbeitsweise wie bei `cout`).

7.3.4 File Flags

Für die Bearbeitung existieren Flags, welche wie folgt definiert sind:

Flag	*Bedeutung*
`ios::in`	Lesen (Default bei `ifstream`)
`ios::out`	Schreiben (Default bei `ofstream`)
`ios::app`	Anhängen
`ios::ate`	ans Ende positionieren
`ios::trunc`	alten Dateiinhalt löschen
`ios::nocreate`	File muss existieren
`ios::noreplace`	File darf nicht existieren

Beispiel:

Der folgende Code öffnet ein File, das bereits existieren muss, zum Schreiben verwendet werden kann und der geschriebene Text an das Ende des Files anhängt.

```
ofstream file("xy.txt", ios::out | ios::app | ios::nocreate);
```

Kapitel 8

Rekursion

It is much easier to be critical
than to be correct.
B.Disraeli

Ein Objekt heisst rekursiv, wenn es sich selbst als ein Teil enthält oder durch sich selbst definiert ist.
Eine Funktion/Methode heisst rekursiv, wenn es mindestens einen Aufruf von sich selbst enthält.

Rekursives Unterprogramm:

```
1 double calculate(...) {
2
3   ...
4   return calculate(...)
5
6 }
```

Die Funktion `calculate` enthält sich selbst. Während des Ablaufs der Funktion ruft sich diese selbst wieder auf.

Vorteile der Rekursion

- Mit der Rekursion lassen sich manche Probleme extrem kurz formulieren bzw. programmieren.

Nachteile der Rekursion

- Ein rekursives Programm braucht mehr Arbeitsspeicher. Mit jedem Aufruf wird neuer Speicher verwendet.
- Bei einigen Problemen ist es schwierig eine für die Rekursion funktionierende Abbruchbedingung zu finden.
- Ein rekursives Programm ist in der Regel langsamer.

8.1 Beispiele Rekursion

8.1.1 Summe aller Zahlen von 0..i

Das folgende Unterprogramm soll die Summe aller Zahlen von `0` bis `i` berechnen.

iterativ

```
1 int summe(int i){
2   int sum = 0;
3   for (int j=0; j<=i; j++){
4     sum = sum + j;
5   }
6   return sum;
7 }
```

rekursiv

```
1 int summe(int i){
2   if (i==1) return 1;
3   return i+summe(i-1);
4 }
```

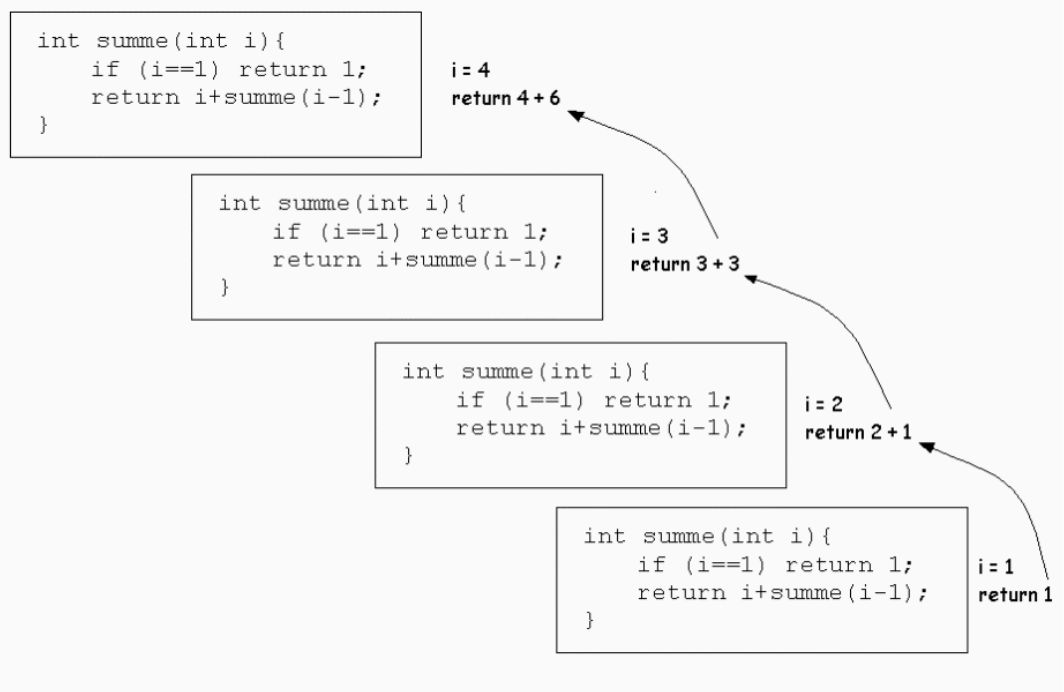


Abbildung 8.1: Rekursiver Aufruf einer Funktion

8.2 Aufgaben

8.2.1 Fibonacci Zahlen

Die Fibonacci Zahlen sind 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Jede Fibonacci Zahl ist die Summe von seinen zwei Vorgängern. Die Definition lautet:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

Schreiben Sie ein Programm, welches mit Hilfe der Rekursion die ersten 20 Fibonacci Zahlen berechnet.
Ergänzen Sie Ihr Programm so, dass die Anzahl Aufrufe der Funktion zur Berechnung einer Fibonacci Zahl angegeben wird.

8.2.2 Rekursive Exponentiation

Die algebraische Definition der Exponentiation lautet: $x^n = x \cdot x \cdot ... \cdot x$
Eine iterative Implementierung könnte wiefolgt aussehen:

```
1 double exp(double x, int n){
2   double y = 1.0;
3   for (int i=0; i<n; i++){
4     y *= x;
5   }
6   return y;
7 }
```

Dieser Algorithmus läuft in linearer Zeit zur Eingangsgrösse n. Ein rekursiver Ansatz kann folgendermassen aussehen:

$$x^n = \begin{cases} (x^2)^{n/2} & n \ \ ist \ \ gerade \\ x(x^2)^{n-1/2} & n \ \ ist \ \ ungerade \end{cases}$$

Schreiben Sie eine rekursive Funktion für die Exponentiation. Wie ist die Laufzeit des rekursiven Ansatzes im Vergleich zum iterativen?

Kapitel 9

Performance

Ein Fehler von Haaresbreite
kann dich tausend Li
in die Irre führen.
Chinesisches Sprichwort

Bei der Entwicklung eines Algorithmus ist es interessant zu wissen, wie viel Zeit die Ausführung bestimmter Code Fragmente beansprucht. Dieses Kapitel beschreibt eine Möglichkeit, wie die Zeit einzelner Code Fragmente gemessen werden kann.

```
#include <ctime>


double start, stop;
start = clock();
// Code Fragment
stop = clock();
```

Mit der Funktion `clock()` wird die verbrauchte CPU Zeit ermittelt. Dies bedeutet, dass die Zeit von parallel laufenden Programmen nicht berücksichtigt wird. Mit der Konstanten `CLOCKS_PER_SEC` kann der Wert in Sekunden umgerechnet werden.

Kapitel 10

Dynamische Datenstrukturen

Ein Geiger zerreisst viele Saiten
ehe er Meister ist.
Sprichwort

In diesem Kapitel werden wichtige Datenstrukturen der Informatik besprochen. Dies sind:

- Array
- Verkettete Liste
- Baum
- Stack

10.1 Array

Ein Array beinhaltet eine fixe Anzahl von einzelnen Daten, welche im Speicher zusammenhängend abgelegt werden. Mit einem Index kann auf die einzelnen Daten zugegriffen werden. Arrays wurden bereits in diesem Script bereits besprochen und es soll an dieser Stelle nicht noch einmal darauf eingegangen werden. Stattdessen sollen die Vor- und Nachteile dieser Datenstruktur untersucht werden:

Vorteile:

- Der Zugriff auf jedes Element im Array benötigt gleich viel Zeit.
- Direkter Zugriff auf das i-te Element.

Nachteile:

- Ein Array hat eine fixe Grösse, welche nicht geändert werden kann. Daraus entsteht der Nachteil, dass das Einfügen und Entfernen einen grossen Aufwand benötigt, da die verbleibenden Daten verschoben werden müssen.

10.2 Verkettete Liste

Bei einer verketteten Liste können Element beliebig eingefügt und entfernt werden. Dies wrid erreicht, dass die Element an einem beliebigen Ort im Speicher liegen. Jedes Element besitzt zusätzlich zu den Daten auch noch einen Pointer, welcher auf das nächste Element zeigt.

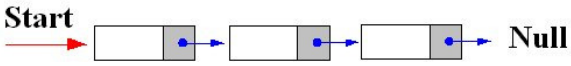


Abbildung 10.1: Verkettete Liste

10.2.1 Einfügen

Ein Element kann an einer beliebigen Stelle eingefügt werden. Dazu wird das Element irgendwo im Speicher abgelegt und die Pointer entsprechend gesetzt. Beim Einfügen wird also nur der Pointer des vorherigen Elementes neu gesetzt.

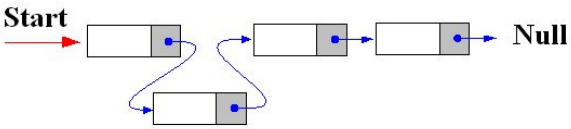


Abbildung 10.2: Einfügen eines Elementes in eine verkettete Liste

10.2.2 Entfernen

Ein Element kann an einer beliebigen Stelle entfernt werden. Dazu werden, analog wie beim Einfügen, die Pointers entsprechend gesetzt.

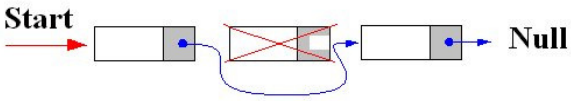


Abbildung 10.3: Enternen eines Elementes aus einer verketteten Liste

Ein Element einer verketteten Liste kann wiefolgt implementiert werden:

```
class Element {
public:
  ElementType data;
  Element *next;
};
```

Für `ElementType` kann ein beliebiger primitiver Datentyp (int, float, ...) oder auch eine Klasse eingesetzt werden.

Vorteile:

- Elemente können beliebig in die Liste eingefügt oder von der Liste entfernt werden.
- Die Grösse (Anzahl Elemente) der Liste ist nicht fix.

Nachteile:

- Die Zugriffszeit auf ein Element ist nicht konstant. Wird auf ein Element zugegriffen, muss die Liste durchlaufen werden.
- Kein direkter Zugriff auf das i-te Element.

10.3 Doppelt verkettete Liste

Eine doppelt verkettete Liste ist ähnlich aufgebaut wie eine einfach verkettete Liste. Der Unterschied besteht darin, dass bei der doppelt verketteten Liste jedes Element einen zusätzlichen Pointer auf das vorherige Element besitzt.

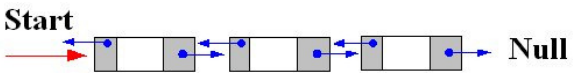


Abbildung 10.4: Doppelt verkettete Liste

Ein Element einer doppelt verketteten Liste kann wiefolgt aussehen:

```
class Element {
public:
  ElementType data;
  Element *next;
  Element *prev;
};
```

Für `ElementType` kann ein beliebiger primitiver Datentyp (int, float, ...) oder auch eine Klasse eingesetzt werden.

10.4 Baum

Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik. Sie sind ähnlich aufgebaut wie verkettete Listen, mit dem Unterschied, dass ein Knoten mehrere Nachfolger haben kann.

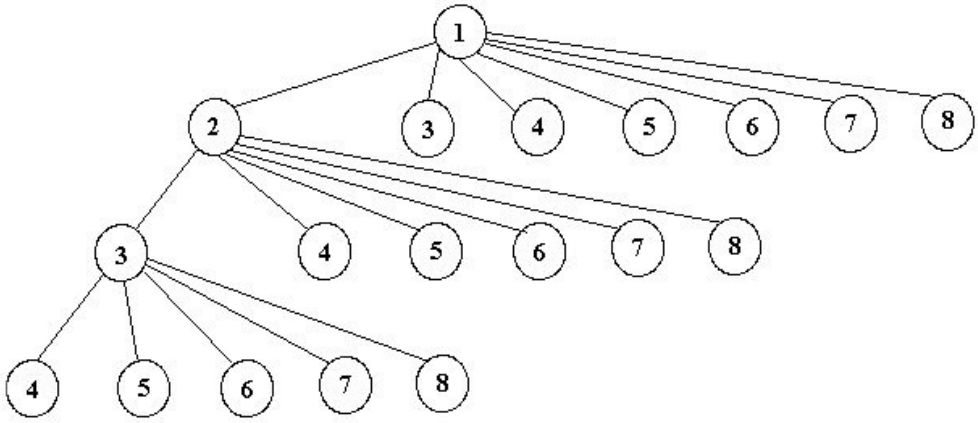


Abbildung 10.5: Beispiel eines Baumes - Die Wurzel wird jeweils oben, die Blätter unten dargestellt

Im folgenden Kapitel sollen lediglich binäre Bäume betrachtet werden.

10.4.1 Binärer Baum

Bei einem binären Baum handelt es sich um einen Baum, bei dem jeder Knoten höchstens zwei Nachfolger besitzt. Die rekursive Definition lautet:

Ein binärer Baum ist entweder leer, oder er besteht aus einer Wurzel mit einem linken und rechten Teilbaum, die wiederum binäre Bäume sind.

10.4.2 Sortierter binärer Baum

Ein binärer Baum heisst *sortiert* bezüglich einem Datenfeld *key*, wenn für jeden Knoten N des Baumes gilt:

1. Für jeden Knoten L des linken Teilbaumes von N gilt $L.key \leq N.key$.
2. Für jeden Knoten R des rechten Teilbaumes von N gilt $R.key > N.key$.

10.4.3 Implementierung

Bei einem binären Baum kann ein Element folgendermassen implementiert werden:

```
1 class Node {
2 public:
3   ElementType data;
4   Node *nextLeft;
5   Node *nextRight;
6 };
```

10.4.4 Traversierung

Ein Baum kann auf drei verschiedene Arten durchlaufen werden.

- **Postorder**
  Besuche den linken Teilbaum, besuche den rechten Teilbaum, besuche die Wurzel.
- **Inorder**
  Besuche den linken Teilbaum, besuche die Wurzel, besuche den rechten Teilbaum.
- **Preorder**
  Besuche die Wurzel, besuche den linken Teilbaum, besuche den rechten Teilbaum.

Betrachten Sie den folgenden binären Baum, welcher entsteht, wenn die folgenden Elemente der Reihe nach eingefügt werden:

22, 28, 23, 5, 89, 47, 11, 19, 27, 1

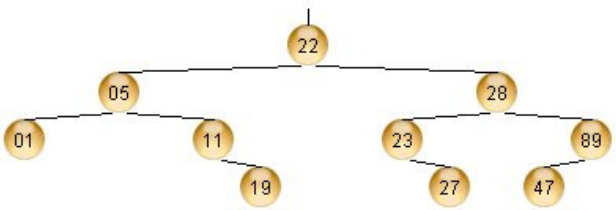


Abbildung 10.6: Binärer Baum

Dieser hat die folgenden Traversierungen:

- Postorder
  1, 19, 11, 5, 27, 23, 47, 89, 28, 22

- Inorder
  1, 5, 11, 19, 22, 23, 27, 28, 47, 89
- Preorder
  22, 5, 1, 11, 19, 28, 23, 27, 89, 47

10.4.5 Eigenschaften

Ein binärer Baum besitzt die folgenden Eigenschaften:

- Für je zwei beliebige Knoten in einem binären Baum existiert genau ein Pfad der sie verbindet.
- Ein Baum mit N Knoten besitzt $N - 1$ Kanten.
- Ein binärer Baum mit N inneren Knoten hat $N + 1$ äussere Knoten.
- Die äussere Pfadlänge eines beliebigen binären Baumes mit N inneren Knoten ist um $2N$ grösser als die innere Pfadlänge.
- Die Höhe eines vollständigen binären Baumes mit N inneren Knoten beträgt etwa $log_2 N$.

10.5 Stack

Die Bezeichnung Stack kommt von der Art und Weise wie diese Datenstruktur verwendet wird. Nämlich wie ein Tellerstapel, bei dem die Teller oben auf den Stapel gelegt werden, und bei der Entfernung eines Tellers wird jeweils der oberste entnommen.
Die folgende Grafik zeigt das Prinzip eines Stacks. Element 4 wird als letztes hinzugefügt und als erstes wieder entfernt. Das hinzufügen eines Elements wird mit `push` bezeichnet. Das Entfernen mit `pop`.

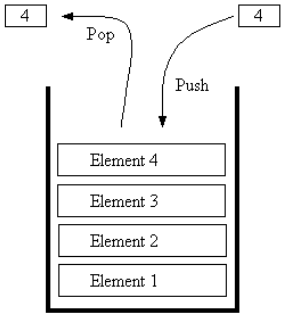


Abbildung 10.7: Stack

Das Datenverarbeitungsprinzip mit Stacks kann auch als LIFO (Last in first out) bezeichnet werden.

Auf einem Stack sollten die folgenden Operationen ausgeführt werden können:

- `int size()`
  Gibt die aktuelle Grösse des Stacks zurück.
- `bool push(ElementType t)`
  Gibt `true` zurück, falls der Wert erfolgreich auf den Stack gelegt werden konnte. Andernfalls `false`.
- `ElementType pop()`
  Liefert das oberste Element des Stack und entfernt dieses zugleich. Zuerst sollte mit `size()` abgefragt werden, ob überhaupt Elemente im Stack vorhanden sind.
- `bool isEmpty()`
  Gibt `true` zurück, falls der Stack leer ist. Andernfalls `false`.
- `ElementType top()`
  Gibt das oberste Element zurück, ohne dieses vom Stack zu entfernen. Zuerst sollte mit `size()` abgefragt werden, ob überhaupt Elemente im Stack vorhanden sind.

10.5.1 Implementierung

Ein Stack kann mit Hilfe eines Arrays oder einer verketteten Liste implementiert werden. Der Nachteil beim Array ist die zu Beginn festgelegte Grösse.

Beispiel:

Implementierung eines Stacks für beliebige Elemente (Templates) mit Hilfe eines Arrays.

```
template <class T>
class Stack{
  int actual, max;
  T *elements;
public:
  Stack(int);
  ~Stack();
  bool push(T);
  bool pop(T&);
};

template <class T>
Stack<T>::Stack(int max){
  actual = 0;
  this->max = max;
  elements = new T[max];
}

template <class T>
Stack<T>::~Stack(){
  delete [] elements;
}

template <class T>
bool Stack<T>::push(T element){
  if (actual >= max) return false;
  elements[actual] = element;
  actual++;
  return true;
}

template <class T>
bool Stack<T>::pop(T &element){
  if (actual <= 0) return false;
  actual--;
  element = elements[actual];
  return true;
}
```

10.6 Aufgaben

10.6.1 Verkettete Liste

Implementieren Sie gemäss dem folgenden Klassendiagramm sowie Headerfile die Klasse `LinkedList`, welche eine verkettete Liste implementiert. Implementieren Sie ein Testprogramm um die korrekte Funktionalität zu zeigen.

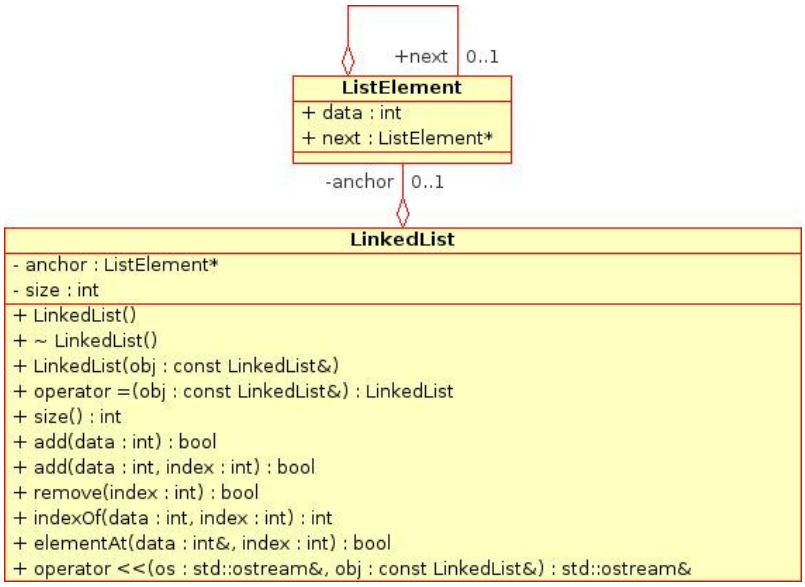


Abbildung 10.8: Klassendiagramm LinkedList

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <iostream>

class ListElement {
public:
  int data;
  ListElement *next;
};

class LinkedList {
private:
  ListElement *anchor;
  int size; // schnellen Zugriff fuer die Abfrage der
            // Anzahl Elemente

public:
  // Konstruktor - Setzt den achor auf NULL
  LinkedList();
```

```
21
22    // Destruktor - Loescht die verkettete Liste
23    ~LinkedList();
24
25    // Copy Konstruktor
26    LinkedList(const LinkedList &obj);
27
28    // Zuweisungsoperator
29    LinkedList operator = (const LinkedList &obj);
30
31    // Gibt die Anzahl Element zurueck
32    int size();
33
34    // Hinzufuegen eines Datums am Ende der Liste
35    bool add(int data);
36
37    // Hinzufuegen eines Datums an Stelle index
38    bool add(int data, int index);
39
40    // Entfernen des Elementes an Stelle index
41    bool remove(int index);
42
43    // Sucht nach einem Element in der Liste ab
44    // der Position index. Wird das Element gefunden,
45    // wird der Index zurueckgegeben, ansonsten der Wert -1
46    int indexOf(int data, int index=0);
47
48    // Gibt das Element an der Stelle index zurueck
49    bool elementAt(int &data, int index);
50
51    // Globale Funktion zur Ausgabe einer verketteten Liste
52    // mit cout
53    friend std::ostream & operator <<
54          (std::ostream & os, const LinkedList & obj);
55 };
56
57 #endif
```

Kapitel 11

Suchen und Sortieren

Wissen ist Macht.
F.Bacon

Im folgenden Kapitel werden Algorithmen zur Sortierung von Arrays und zur Suche von Elementen in Arrays vorgestellt.

Sortieralgorithmen setzen voraus, dass je zwei Elemente nach irgendeinem Kriterium *vergleichbar* sind. Der Vergleich kann so einfach sein wie die Grösse von Zahlen oder so kompliziert wie die Bewertung der Siegeschancen einer Stellung im Schachspiel. Für den Algorithmus spielt die Art der verglichenen Information keine Rolle.

Die Qualität von Sortieralgorithmen lässt sich unter verschiedenen Gesichtspunkten vergleichen. Dabei ist in erster Linie die Geschwindigkeit interessant, die üblicherweise an der Anzahl Elementvergleiche sowie der Anzahl Kopieraktionen gemessen wird. Die konkrete Anzahl Operationen hängt stark von der zu sortierenden Datenstruktur ab. Deshalb berücksichtigt man den besten, den schlechtesten und den durchschnittlichen Fall.

Im folgenden wird jeweils angenommen, dass die Daten in aufsteigender Reihenfolge sortiert werden.

11.1 Bubblesort

Der Name Bubblesort rührt von der bildhaften Darstellung her, dass der Algorithmus kleine Elemente wie Luftblasen im Wasser nach oben (oben = Beginn der Daten) steigen lässt.

Bubblesort beruht auf der folgenden Idee: Die Daten werden vom Anfang bis zum Ende durchgegangen und die Elemente dabei paarweise verglichen. Wenn beide in der richtigen Reihenfolge stehen (das kleinere vor dem grösseren), dann wird mit dem nächsten Paar fortgefahren. Falls die Elemente in der falschen Reihenfolge stehen, werden sie zuerst vertauscht. Das grösste Element wandert somit an das Ende der Daten.

Nun wird der Prozess wiederholt, jedoch ohne das letzte Element. Im übernächsten Durchgang werden die letzten beiden Elemente ausgelassen, u.s.w. Die Daten sind sortiert, falls nur noch ein Element zum sortieren übriggeblieben ist.

Der folgende Array soll nun mit Hilfe des Bubblesort Algorithmus sortiert werden.

```
int a[] = {45, 34, 91, 23, 12, 67, 2, 36};
```

Dazu sind die folgenden Schritte notwendig:

```
45, 34, 91, 23, 12, 67, 2, 36    Start
34, 45, 23, 12, 67, 2, 36, 91    1.Schritt
34, 23, 12, 45, 2, 36, 67, 91    2.Schritt
23, 12, 34, 2, 36, 45, 67, 91    3.Schritt
12, 23, 2, 34, 36, 45, 67, 91    4.Schritt
12, 2, 23, 34, 36, 45, 67, 91    5.Schritt
2, 12, 23, 34, 36, 45, 67, 91    6.Schritt Ende
```

11.1.1 Analyse

Nehmen Sie an, die Anzahl der sortierenden Elemente beträgt n. Bubblesort benötigt im Durchschnitt und im ungünstigsten Fall ungefähr $\frac{n^2}{2}$ Vergleiche und $\frac{n^2}{2}$ Austauschoperationen.

11.1.2 Source Code

Der folgende Code zeigt die Implementierung des Bubblesort Algorithmus in C++.

```
1  // Bubble Sort Algorithmus
2  void bubbleSort(vector <int> &v){
3    int tmp;
4    bool change;
5    for (int i(0); i<v.size(); i++){
6      change = false;
7      for (int j(0); j<v.size()-i-1; j++){
8        if (v[j] > v[j+1]){
9          tmp = v[j];
10         v[j] = v[j+1];
11         v[j+1] = tmp;
12         change = true;
13       }
14     }
15     if (!change) break;
16   }
17 }
```

11.2 Selection Sort

Selection Sort arbeitet nach folgendem Mechanismus: Die Daten nach dem kleinsten Element durchsuchen, anschliessend dieses an den Beginn setzen. Im nächsten Schritt wird wieder das kleinste Element gesucht jedoch wird das erste Element nicht mehr berücksichtigt, u.s.w. Der Algortithmus ist am Ende, falls nur noch ein Element übrig geblieben ist.

Der Vorteil von Selection Sort gegenüber Bubblesort liegt in der Art, wie ein Element an seinen Zielort transportiert wird. Bei Bubblesort wird ein Element aus der Quelle herausgenommen, alle Elemente zwischen dem Quellort und dem Zielort um eine Position verschoben und dann das Element an seinem Zielort eingefügt. Selection Sort vertauscht nur die beiden Elemente am Quell- und Zielort und lässt alle Elemente dazwischen bestehen.

Der folgende Array soll nun mit Hilfe des Selection Sort Algorithmus sortiert werden.

```
int a[] = {45, 34, 91, 23, 12, 67, 2, 36};
```

Dazu sind die folgenden Schritte notwendig:

```
45, 34, 91, 23, 12, 67, 2, 36    Start
2, 34, 91, 23, 12, 67, 45, 36    1.Schritt
2, 12, 91, 23, 34, 67, 45, 36    2.Schritt
2, 12, 23, 91, 34, 67, 45, 36    3.Schritt
2, 12, 23, 34, 91, 67, 45, 36    4.Schritt
2, 12, 23, 34, 36, 67, 45, 91    5.Schritt
2, 12, 23, 34, 36, 45, 67, 91    6.Schritt Ende
```

11.2.1 Analyse

Nehmen Sie an, die Anzahl der sortierenden Elemente beträgt n. Selection Sort benötigt im Durchschnitt und im ungünstigsten Fall ungefähr $\frac{n^2}{2}$ Vergleiche und n Austauschoperationen.

11.2.2 Source Code

Der folgende Code zeigt die Implementierung des Selection Sort Algorithmus in C++.

```
1  // Selection Sort Algorithmus
2  void selectionSort(vector <int> &v){
3    int min, indexMin;
4    for (int i(0); i<v.size(); i++){
5      min = v[i];
6      indexMin = i;
7      for (int j(i+1); j<v.size(); j++){
8        if (min > v[j]){
9          min = v[j];
10         indexMin = j;
11       }
12     }
13     v[indexMin] = v[i];
14     v[i] = min;
15   }
16 }
```

11.3 Insertion Sort

Insertion Sort ist in gewissem Sinn das Gegenstück von Selection Sort. Von vorne nach hinten im Datensatz wird ein Element nach dem anderen ausgewählt. Für ein ausgewähltes Element wird im vorderen, schon sortierten Teil der Daten die passende Position gesucht. Dann wird an dieser Position durch Verschieben des restlichen Abschnittes Platz geschaffen und das ausgewählte Element eingefügt. Diesen Algorithmus wenden die meisten Leute an, falls ein gemischter Kartenstapel sortiert werden soll.

Der folgende Array soll nun mit Hilfe des Insertion Sort Algorithmus sortiert werden.

```
int a[] = {45, 34, 91, 23, 12, 67, 2, 36};
```

Dazu sind die folgenden Schritte notwendig:

```
45, 34, 91, 23, 12, 67, 2, 36    Start
34, 45, 91, 23, 12, 67, 2, 36    1.Schritt
34, 45, 91, 23, 12, 67, 2, 36    2.Schritt
23, 34, 45, 91, 12, 67, 2, 36    3.Schritt
12, 23, 34, 45, 91, 67, 2, 36    4.Schritt
12, 23, 34, 45, 67, 91, 2, 36    5.Schritt
2, 12, 23, 34, 45, 67, 91, 36    6.Schritt
2, 12, 23, 34, 36, 45, 67, 91    7.Schritt Ende
```

11.3.1 Analyse

Nehmen Sie an, die Anzahl der sortierenden Elemente beträgt n. Selection Sort benötigt im Durchschnitt $\frac{n^2}{4}$ Vergleiche und $\frac{n^2}{8}$ Austauschoperationen, im ungünstigsten Fall fast doppelt so viele.

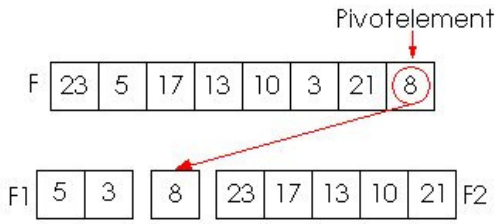
11.3.2 Source Code

Der folgende Code zeigt die Implementierung des Insertion Sort Algorithmus in C++.

```
1  // Insertion Sort Algorithmus
2  void insertionSort(vector <int> &v){
3    int key, index;
4    for (int i(1); i<v.size(); i++){
5      key = v[i];
6      index = i;
7      for (int j(i); j>0; j--){
8        if (key < v[j-1]){
9          v[j] = v[j-1];
10         index = j-1;
11       }
12     }
13     v[index] = key;
14   }
15 }
```

11.4 Quicksort

Um eine Folge $F = k_0, ..., k_{n-1}$ zu sortieren wird ein Pivotelement k gewählt. Danach wird die Folge in zwei Teilfolgen F_1 und F_2 aufgeteilt. F_1 besteht aus Elementen von F welche kleiner oder gleich k sind. F_2 besteht aus Elementen von F welche grösser oder gleich k sind. Das Pivotelement selbst kommt weder in F_1 noch in F_2.



Das Sortierproblem kann nun so gelöst werden, dass auf die Teilfolgen F_1 und F_2 das gleiche Prinzip angewandt wird. Dies kann durch Rekursion erfolgen. Die Ergebnisse werden jeweils am Schluss zusammengesetzt.

11.4.1 Source Code

Der folgende Code zeigt eine mögliche Implementierung des Quicksort Algorithmus in C++. Als Pivotelement einer Folge wird jeweils das Element gewählt, welches in der Folge ganz rechts liegt.

```
1  // Quicksort Algorithmus
2  void quickSort(vector <int> &v){
3
4    // Abbruchbedingung
5    if (v.size() <= 1){
6      return;
7    }
8
9    // Divide
10   int pivot = v.at(v.size()-1);
11   vector <int> f1;
12   vector <int> f2;
13   for (int i=0; i<v.size()-1; i++){
14     int key = v.at(i);
15     if (key<=pivot) f1.push_back(key);
16     else f2.push_back(key);
17   }
18
19   // Conquer
20   quickSort(f1);
21   quickSort(f2);
22
23   // Merge
24   int i;
25   for (i=0; i<f1.size(); i++){
26     v[i] = f1.at(i);
27   }
28   v[i] = pivot;
29   i++;
30   for (int j=0; j<f2.size(); j++){
31     v[i+j] = f2.at(j);
32   }
33 }
```

11.5 Lineare Suche

Die lineare Suche ist die einfachste Suche. Gegeben ist eine Folge $F = k_0, ..., k_{n-1}$ mit n Elementen, in welcher nach einem bestimmten Element gesucht werden soll. Man geht dazu die komplette Folge durch, bis das Element gefunden wurde. Der Suchaufwand wächst linear mit der Anzahl Elemente. Im besten Fall ist das gesuchte Elemente an erster Stelle, im schlechtesten Fall an letzter Stelle.

11.5.1 Source Code

```
1  // Lineare Suche nach einem Element
2  int linearSearch(const vector <int> &v, int element){
3    int index = -1;
4
5    for (int i=0; v.size(); i++){
6      if (v.at(i) == element){
7        index = i;
8        break;
9      }
10   }
11
12   return index;
13 }
```

11.6 Binäre Suche

Bei der binären Suche muss die Folge $F = k_0, ..., k_{n-1}$, in welcher gesucht wird, sortiert sein. Der Algorithmus funktioniert dann wiefolgt:
Zuerst wird das mittlere Element der Folge überprüft. Es kann kleiner, grösser oder gleich dem gesuchten Element sein. Ist es kleiner als das gesuchte Element, muss in der hinteren Hälfte weitergesucht werden. Ist es grösser, wird in der vorderen Hälfte gesucht. Die jeweils andere Hälfte wird nicht mehr betrachtet. Ist das mittlere Element gleich dem gesuchten Element kann die Suche beendet werden.
Jede weiterhin zu untersuchende Hälfte wird wieder gleich behandelt. Das mittlere Element liefert wieder die Entscheidung wo weitergesucht wird.
Die Grösse des Suchbereichs wird in jedem Schritt halbiert. Spätestens wenn der Suchbereich nur noch ein Element hat, ist die Suche beendet. Dieses ist dann entweder das gesuchte Element oder das gesuchte Element kommt nicht vor.
Die maximale Anzahl Schritte beträgt $log_2 n$. Dies ist deutlich schneller als die lineare Suche, hat aber den Nachteil, dass die Daten sortiert sein müssen.

11.6.1 Source Code

```
1  // Binaere Suche
2  int binarySearch(const vector <int> &v, int element){
3    int first = 0;
4    int last = v.size()-1;
5
6    while (first <= last){
7      int mid = (first + last) / 2;
8      if (element > v.at(mid)){
9        // Suche in oberer Menge fortsetzen
10       first = mid + 1;
11     }
12     else if (element < v.at(mid)){
13       // Suche in unterer Menge fortsetzen
14       last = mid - 1;
15     }
16     else {
17       // Element gefunden
18       return mid;
19     }
20   }
21
22   // failed
23   return -1;
24 }
```

11.7 Binäre Suche rekursiv

Die binäre Suche rekursiv funktioniert gleich wie die binäre Suche, mit dem Unterschied das keine Schleife verwendet wird, sondern das sich die Funktion selbst wieder aufruft.

11.7.1 Source Code

```
1  // Binaere Suche rekursiv
2  int rbinarySearch(const vector <int> &v, int element,
3                    int first, int last){
4
5    // Abbruchbedingung
6    if (first > last) return -1;
7
8    int mid = (first + last) / 2;
9    if (element > v.at(mid)){
10     // Suche in oberer Menge fortsetzen
11     return rbinarySearch(v, element, mid+1, last);
12   }
13   else if (element < v.at(mid)){
14     // Suche in unterer Menge fortsetzen
15     return rbinarySearch(v, element, first, mid-1);
16   }
17   else {
18     // Element gefunden
19     return mid;
20   }
21 }
```

11.8 Zeitkomplexität

Die folgende Tabelle zeigt eine Übersicht der Laufzeiten über die in diesem Kapitel besprochenen Algorithmen. Die Laufzeit ist proportional zun den Angegebenen Funktionen. Die Anzahl der zu verarbeitenden Daten beträgt n.

Algorithmus	**Laufzeit**
Bubblesort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Quick Sort	$O(n \cdot log(n))$
Linare Suche	$O(n)$
Binäre Suche	$O(log(n))$

Kapitel 12

STL

Perfection is attained if there is
nothing left to remove.
Perspective Technologies

Im folgenden Kapitel wird die STL - Standard Template Library beschrieben. Die STL ist eine Klassenbibliothek, welche aus den folgenden drei Elementen besteht:

- Container
- Iteratoren
- Algorithmen

Die wichtigsten Algortihmen und Datenstrukturen sind mit der STL abgedeckt.

12.1 Container

Ein Container kann zur Speicherung von Daten eines beliebigen Datentypes verwendet werden. Es existieren verschiedene Arten von Containern. Jeder Container besitzt Vor- und Nachteile. Es gibt sequentielle Container und assoziative Container. Sequentielle Container speichern die Daten in der Reihenfolge ab, wie diese in den Container gelegt werden. Assoziative Container speichern die Daten in einer internen Ordnung ab. Dies erlaubt später einen sehr schnellen Zugriff auf die Daten.

12.1.1 Sequentielle Container

Es existieren die folgenden sequentiellen Container:

- Vector
- List
- Deque

Vector

Ein `vector` ist ein linearer dynamischer Array, welcher schnelles Einfügen und Entfernen von Daten am Ende erlaubt.

Die wichtigsten Methoden:

`front()`	gibt das erste Element im Vector zurück
`back()`	gibt das letzte Element im Vector zurück
`push_back()`	Einfügen am Ende
`pop_back()`	Entfernen am Ende
`size()`	Anzahl Elemente im Vector
`[n]`	Zugriff auf n-tes Element
`at(n)`	Zugriff auf n-tes Element

Um die Klasse im Code zu verwenden wird das folgende include Statement verwendet:
`#include <vector>`

Beispiel:

```
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char **argv){

        // Erzeugen eines Vectors
        vector <int> v1;
        vector <int> v2(10);

        // Abfragen der Groesse
        cout << v1.size() << endl; // 0
        cout << v2.size() << endl; // 10;

        // 10 Werte am Ende einfuegen
        for (int i=0; i<10; i++){
                v1.push_back(i);
                v2.push_back(i);
        }

        // Abfragen der Groesse
        cout << v1.size() << endl; // 10
```

```
23          cout << v2.size() << endl; // 20;
24
25          // n-tes Element lesen
26          int n = 3;
27          int value = v1[n]; // oder v1.at(n)
28
29          // n-tes Element schreiben
30          v1[n] = value;
31
32          // Element am Anfang
33          int first = v1.front();
34
35          // Element am Ende
36          int last = v1.back();
37
38          // Element am Ende entfernen
39          v1.pop_back();
40
41          return 0;
42  }
```

Deque

Eine `deque` ist ein linearer dynamischer Array, welcher schnelles Einfügen und Entfernen von Daten am Ende und Anfang erlaubt.

Die wichtigsten Methoden:

`front()`	gibt das erste Element der Deque zurück
`back()`	gibt das letzte Element der Deque zurück
`push_back()`	Einfügen am Ende
`push_front()`	Einfügen am Anfang
`pop_back()`	Entfernen am Ende
`pop_front()`	Entfernen am Anfang
`size()`	Anzahl Elemente in der Deque
`[n]`	Zugriff auf n-tes Element
`at(n)`	Zugriff auf n-tes Element

Um die Klasse im Code zu verwenden wird das folgende include Statement verwendet:
`#include <deque>`

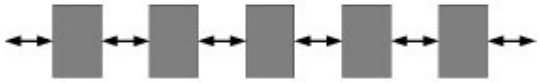
Beispiel:

```
#include <iostream>
#include <deque>
using namespace std;

int main(int argc, char **argv){

        // Erzeugen eines Vectors
        deque <int> d;

        // Abfragen der Groesse
        cout << d.size() << endl; // 0

        // 10 Elemente am Ende einfuegen
        for (int i=0; i<10; i++){
                d.push_back(i);
        }

        // 10 Elemente am Anfang einfuegen
        for (int i=0; i<10; i++){
                d.push_front(i);
        }
```

```
22
23          // Abfragen der Groesse
24          cout << d.size() << endl; // 20
25
26          // n-tes Element lesen
27          int n = 3;
28          int value = d[n]; // oder v1.at(n)
29
30          // n-tes Element schreiben
31          d[n] = value;
32
33          // Element am Anfang
34          int first = d.front();
35
36          // Element am Ende
37          int last = d.back();
38
39          // Element am Anfang entfernen
40          d.pop_front();
41
42          // Element am Ende entfernen
43          d.pop_back();
44
45          // Abfragen der Groesse
46          cout << d.size() << endl; // 18
47
48          return 0;
49  }
```

List

Eine `list` ist eine doppelt verkettete Liste, welcher schnelles Einfügen und Entfernen von Daten am Ende und Anfang erlaubt.



Die wichtigsten Methoden:

`front()`	gibt das erste Element der Liste zurück
`back()`	gibt das letzte Element der Liste zurück
`push_back()`	Einfügen am Ende
`push_front()`	Einfügen am Anfang
`pop_back()`	Entfernen am Ende
`pop_front()`	Entfernen am Anfang
`size()`	Anzahl Elemente in der Liste

Um die Klasse im Code zu verwenden wird das folgende include Statement verwendet:
`#include <list>`

Beispiel:

```
#include <iostream>
#include <list>
using namespace std;

int main(int argc, char **argv){

        // Erzeugen einer List
        list <int> l;

        // Abfragen der Groesse
        cout << l.size() << endl; // 0

        // 10 Elemente am Ende einfuegen
        for (int i=0; i<10; i++){
                l.push_back(i);
        }

        // 10 Elemente am Anfang einfuegen
        for (int i=0; i<10; i++){
                l.push_front(i);
        }

        // Abfragen der Groesse
        cout << l.size() << endl; // 20
```

```
25
26          // Element am Anfang
27          int first = l.front();
28
29          // Element am Ende
30          int last = l.back();
31
32          // Element am Anfang entfernen
33          l.pop_front();
34
35          // Element am Ende entfernen
36          l.pop_back();
37
38          // Abfragen der Groesse
39          cout << l.size() << endl; // 18
40
41          return 0;
42  }
```

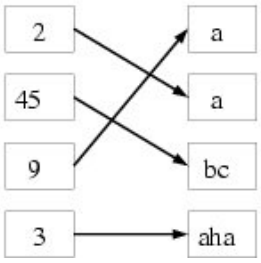
12.1.2 Assoziative Container

Es existieren die folgenden assoziativen Container:

- Map / Multimap
- Set / Multiset

Map / Multimap

Eine `Map` speichert Paare von Schlüsseln und zugehörigen Daten. Der Schlüssel ist eindeutig, d.h. er darf nur einmal vorkommen. Bei einer `Multimap` darf ein Schlüssel mehrmals vorkommen. Die Daten werden in einer Baumstruktur sortiert gespeichert.



Die wichtigsten Methoden:

`insert()`	Einfügen eines Elementes
`back()`	gibt das letzte Element im Vector zurück
`clear()`	Map löschen
`swap()`	Inhalt zweier Map's austauschen
`erase()`	Löschen eines Elementes
`size()`	Anzahl Elemente in der Map

Um die Klasse im Code zu verwenden wird das folgende include Statement verwendet:
`#include <map>`

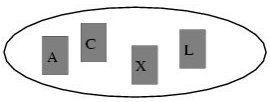
Beispiel:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main(int argc, char **argv){

  // Erzeugen einer Map (int - string Werte)
  map <int, string> m1;

  // Abfragen der Groesse
  cout << m1.size() << endl; // 0

  // Elemente einfuegen

  // Unter Schlussel 0 wird der Wert 'Brad Pitt'
  // gespeichert
  m1[0] = "Brad Pitt";

  // Da der Schluessel 0 bereits vorkommt, wird
  // der Wert fuer Schluessel 0 ueberschrieben.
  // D.h. der alte Wert 'Brad Pitt' wurde
  // geloescht.
  m1[0] = "Claire Forlani";

  // Elemente einfuegen mit der pair Klasse
  pair <int, string> p;
  m1.insert(make_pair(1, "Adrian Cronauer"));
  m1.insert(pair<int, string>(2, "Francis Hummel"));

  // Element einfuegen mit exaktem Typ
  m1.insert(map<int, string>::value_type(3,
    "Stanley Goodspeed"));

  // Abfragen der Groesse
  cout << m1.size() << endl; // 4

```

```
38   // Erzeugen einer Map
39   map <int, string> m2;
40   m2[0] = "James Bond";
41
42   // Werte der beiden Maps austauschen
43   m1.swap(m2);
44
45   // Eintrag mit Schluessel 0 aus der Map loeschen
46   m1.erase(0);
47
48   // Abfragen der Groesse
49   cout << m1.size() << endl; // 0
50
51   // Map loeschen
52   m2.clear();
53
54   return 0;
55 }
```

Set / Multiset

Eine `Set` speichert einzelne Schlüssel. Der Schlüssel ist eindeutig, d.h. er darf nur einmal vorkommen. Bei einer `Multiset` darf ein Schlüssel mehrmals vorkommen. Die Schlüssel werden in einer Baumstruktur sortiert gespeichert.



Die wichtigsten Methoden:

`insert()`	Einfügen eines Elementes
`back()`	gibt das letzte Element im Vector zurück
`clear()`	Map löschen
`swap()`	Inhalt zweier Map's austauschen
`erase()`	Löschen eines Elementes
`size()`	Anzahl Elemente in der Map

Um die Klasse im Code zu verwenden wird das folgende include Statement verwendet:
`#include <set>`

Beispiel:

```
1 #include <iostream>
2 #include <set>
3 using namespace std;
4
```

```
5  int main(int argc, char **argv){
6
7    // Erzeugen einer Set (int Werte)
8    set <int> s1;
9
10   // Abfragen der Groesse
11   cout << s1.size() << endl; // 0
12
13   // Element einfuegen
14   s1.insert(5);
15
16   // Abfragen der Groesse
17   cout << s1.size() << endl; // 2
18
19   // Erzeugen einer Set
20   set <int> s2;
21   s2.insert(23);
22
23   // Werte der beiden Sets austauschen
24   s1.swap(s2);
25
26   // Schluessel 23 aus Set1 loeschen
27   s1.erase(23);
28   s1.erase(3); // hier passiert nichts
29
30   // Abfragen der Groesse
31   cout << s1.size() << endl; // 0
32
33   // Set loeschen
34   s2.clear();
35
36   return 0;
37 }
```

12.2 Iteratoren

Bei einem Vector und einer Deque kann mit Hilfe eines Index auf die Elemente im Container zugegriffen werden. Doch bei allen anderen Container wird ein Interator benötigt um auf die Element zugreifen zu können. Werden Iteratoren für den Zugriff verwendet, so bietet sich auch die einfache Möglichkeit an, den Container austauschen zu können. Ein Iterator bietet eine passende Schnittstelle für den Datenzugriff.

Der folgende Code zeigt einen einfachen Iterator, welcher verwendet wird um auf die Elemente in einem Vector zugreifen zu können.

```
#include <vector>
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
  vector<int> container;

  // fill up container with random values
  for (int i=0; i<10; i++){
    container.push_back(rand());
  }

  // parse container with iterator
  vector<int>::iterator it;
  for (it=container.begin(); it!=container.end(); it++) {
    cout << *it << endl;
  }

  return 0;
}
```

Wird nun statt ein Vector eine List verwendet, so muss der Code, welcher verwendet wird um auf die Elemente im Container zugreifen zu können, nicht verändert werden.

```
#include <list>
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
  list<int> container;

  // fill up container with random values
  for (int i=0; i<10; i++){
    container.push_back(rand());
  }

  // parse container with iterator
  list<int>::iterator it;
  for (it=container.begin(); it!=container.end(); it++) {
    cout << *it << endl;
  }
```

```
18
19    return 0;
20  }
```

12.2.1 Map Iterator

Der folgende Code zeigt das Durchlaufen der Elemente in einer Map.

```
1  #include <map>
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main(int argc, char **argv){
7    map <string, int> phonebook;
8    phonebook["Herzig"] = 1234;
9    phonebook["Tanner"] = 4321;
10   phonebook["Scherrer"] = 2314;
11
12   map <string, int>::iterator it;
13   for (it=phonebook.begin(); it!=phonebook.end(); it++){
14     cout << it->first << " " << it->second << endl;
15   }
16
17   return 0;
18 }
```

12.3 Algorithmen

Die Algorithmen, welche in der Headerdatei `<algorithm>` implementiert sind, sind unabhängig vom verwendeten Container. Sie kennen nur Iteratoren, über welche auf die Elemente zugegriffen wird. Da sehr viele verschiedene Algorithmen zur Verfügung stehen, sollen in diesem Kapitel lediglich die wichtigsten betrachtet werden.
Es existieren Algorithmen, bei welchen die Daten im entsprechenden Container modifiziert werden, sowie solche Algorithmen, bei welchen die Daten nicht modifiziert werden.

12.3.1 find

Der Algorithmus `find()` wird verwendet um ein bestimmtes Element im Container zu suchen. Das Ergebnis ist ein Iterator, welcher auf die gefundene Stelle zeigt. Falls das Element nicht gefunden wurde, so zeigt das Ergebnis auf den `end()` Wert des Containers.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
```

```
5
6  int main(int argc, char **argv){
7    vector<int> container;
8    for (int i=0; i<10; i++){
9      container.push_back(i);
10   }
11   vector<int>::iterator result =
12     find(container.begin(), container.end(), 5);
13   if (result == container.end()) {
14     cout << "not found" << endl;
15   }
16   else {
17     cout << "found" << endl;
18   }
19   return 0;
20 }
```

12.3.2 count

Der Algorithmus `count()` gibt die Anzahl Elemente zurück, welche gleich einem bestimmten Wert sind.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main(int argc, char **argv){
7    vector<int> container;
8    for (int i=0; i<10; i++){
9      container.push_back(i);
10   }
11   cout <<
12     count(container.begin(), container.end(), 5)
13     << endl;
14
15   return 0;
16 }
```

12.3.3 search

Der Algorithmus `search()` sucht nach einer Sequenz innerhalb einer anderen Sequenz.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
```

```
5
6  int main(int argc, char **argv) {
7    vector<int> sequence;
8    vector<int> subsequence;
9
10   for (int i=0; i<10; i++) {
11     sequence.push_back(i);
12     if (i > 2 && i < 5) {
13       subsequence.push_back(i);
14     }
15   }
16
17   vector<int>::iterator result = search(
18     sequence.begin(), sequence.end(),
19     subsequence.begin(), subsequence.end());
20
21   if (result != sequence.end()) {
22     cout << "Sequence found" << endl;
23   }
24   else {
25     cout << "Sequence not found" << endl;
26   }
27   return 0;
28 }
```

12.3.4 replace

Der Algorithmus `replace()` ersetzt in einem Container alle Elemente, welche einen betimmten Wert besitzen mit einem anderen Element.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main(int argc, char **argv) {
7    vector<int> v1;
8    for (int i=0; i<10; i++) {
9      v1.push_back(i);
10   }
11
12   replace(v1.begin(), v1.end(), 1, 23);
13
14   vector<int>::iterator it;
15   for (it=v1.begin(); it!=v1.end(); it++) {
16     cout << *it << endl;
17   }
18
19   return 0;
```

```
}
```

12.3.5 reverse

Der Algorithmus `reverse()` kehrt die Reihenfolge der Elemente innerhalb eines Containers um.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char **argv) {
  vector<int> sequence;
  vector<int> subsequence;

  for (int i=0; i<10; i++) {
    sequence.push_back(i);
    if (i > 2 && i < 5) {
      subsequence.push_back(i);
    }
  }

  vector<int>::iterator result = search(
    sequence.begin(), sequence.end(),
    subsequence.begin(), subsequence.end());

  if (result != sequence.end()) {
    cout << "Sequence found" << endl;
  }
  else {
    cout << "Sequence not found" << endl;
  }
  return 0;
}
```

12.3.6 sort

Der Algorithmus `sort()` wird verwendet um die Elemente innerhalb eines Containers zu sortieren.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char **argv) {
  vector<int> v1;
```

```
8   for (int i=0; i<10; i++) {
9     v1.push_back(rand());
10  }
11
12  sort(v1.begin(), v1.end());
13
14  vector<int>::iterator it;
15  for (it=v1.begin(); it!=v1.end(); it++) {
16    cout << *it << endl;
17  }
18
19  return 0;
20 }
```

Kapitel 13

QT

Innovation besteht ganz einfach aus neuen Ideen,
die zu Produkten mit einem ungeahnten
neuen Mehrwertfür den Nutzer führen. Neuheiten
ohne Mehrwert sind nicht innovativ.
R.Sun

QT ist eine Klassenbibliothek, welche Klassen zur Erstellung von graphischen Benutzeroberflächen zur Verfügung stellt. QT steht für verschiedene Plattformen zur Verfügung. Darunter gehören alle gängigen Unix/Linux Plattformen, Windows Systeme und Mac OS X.
Im Laufe der Jahre hat sich QT so stark weiterentwickelt und bietet heute auch Klassen für diverse andere Dinge an. Dazu gehören Netzwerkkommunikation, Filehandling, XML Verarbeitung, u.s.w.
Aufgrund der riesigen Möglichkeiten, welche QT bietet, wird in diesem Kapitel nur eine kleine Einführung zur Programmierung mit QT gegeben.

13.1 Struktur eines QT Programms

Die Struktur eines QT Programmes sieht folgendermassen aus:

```
1  // my first program with QT
2
3  #include <QApplication>
4  #include <QPushButton>
5
6  int main(int argc, char *argv[]){
7    QApplication app(argc, argv);
8    QPushButton hello("Hello world!");
9    hello.show();
10   return app.exec();
11 }
```

Beschreibung:

```
1 #include <QApplication>
2 #include <QPushButton>
```

Jede Klasse die von QT zur Verfügung gestellt wird, muss über die `#include` *Anweisung eingebunden werden, damit diese verwendet werden kann.*

```
1 int main(int argc, char** argv)
```

Startpunkt der Applikation. Die Parameter `argc` *und* `argv` *werden für die Erstellung einer QT Applikation verwendet*

```
1 QApplication app(argc, argv);
```

Jede QT Applikation enthält genau ein Objekt dieser Klasse. Dieses Objekt ist für das komplette Eventhandling verantwortlich.

```
1 QPushButton hello("Hello world!");
```

Hier wird ein Objekt der Klasse `QPushButton` *erstellt. Dieses stellt einen graphischen Button dar.*

```
1 hello.show();
```

Die Methode `show` *wird verwendet um den Button anzuzeigen.*

```
1 return app.exec();
```

An dieser Stelle wird die Kontrolle von der main Methode auf die QT Applikation übertragen. Sobald die QT Applikation beendet wird, wird an diesem Punkt weitergefahren.

Abbildung 13.1: Graphische Oberfläche des QT Programms

13.2 QT Hilfe

Eine Übersicht über alle Klassen, welche von QT zur Verfügung gestellt werden, erhält man mit der QT Dokumentation, welche bei der Installation mitgeliefert wird. Alternativ kann auch die Dokumentation auf der Webseite von Trolltech angeschaut werden. Dort sind alle Klassen detailliert beschrieben. Wird mit QT gearbeitet, so ist diese Dokumentation unerlässlich.

http://doc.trolltech.com

13.3 Widgets

Alle Klassen welche in QT verwendet werden um Graphische Benutzerschnittstellen zu gestalten, sind Widgets. Dazu gehören Buttons, Dialogboxen, Laufbalken, ... u.s.w. Das Wort Widget stammt aus den beiden Begriffen *Window* und *Gadget*. Widgets selber können wiederum Widgets enthalten. Beispielsweise ein Button in einer Dialogbox. Eine Widget ist ein Objekt einer Klasse, welche von der Klasse `QWidget` abgeleitet ist.

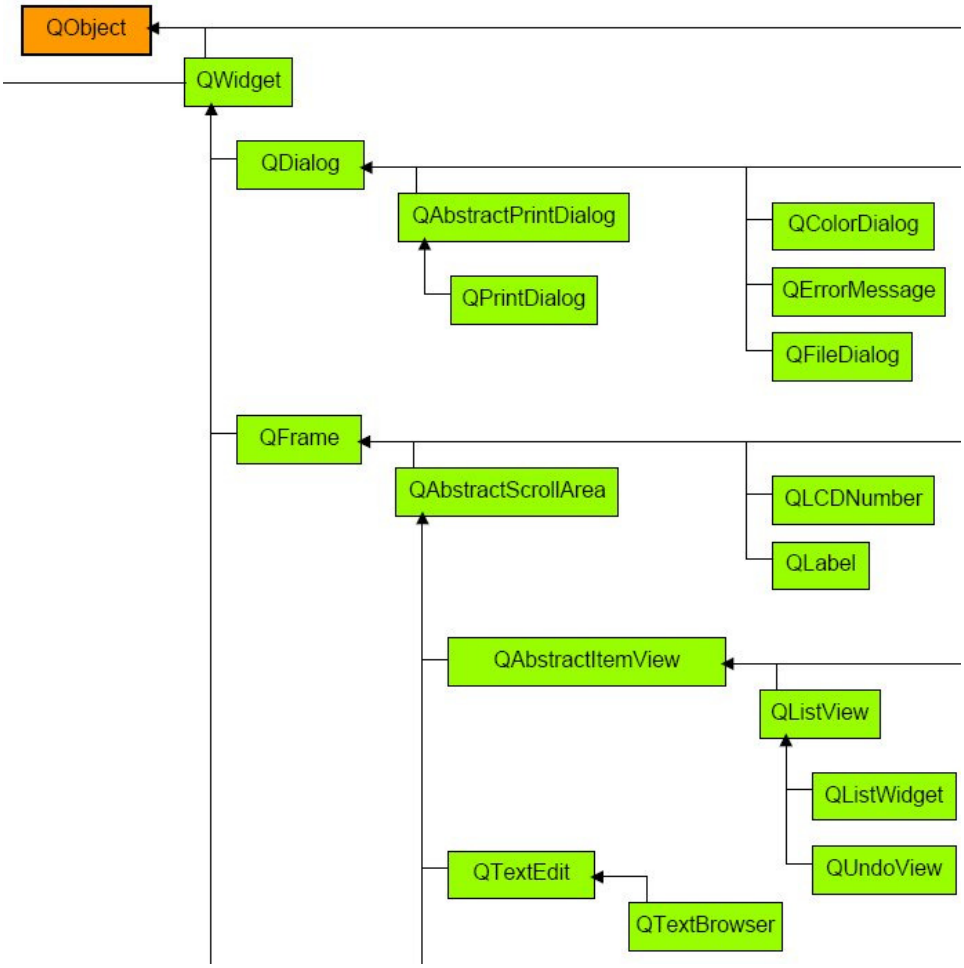


Abbildung 13.2: Auschnitt aus dem Klassendiagramm von QT

Wird ein eigenes Widget entwickelt, so muss dieses ebenfalls direkt oder indirekt von der Klasse `QWidget` abgeleitet werden.

13.3.1 MainWindow

Die Klasse `QMainWindow` stellt ein typisches Applikationsfenster zur Verfügung. Dieses bietet unter anderem eine Menubar und eine Statusbar. Die folgende Grafik zeigt den Aufbau dieser Klasse:

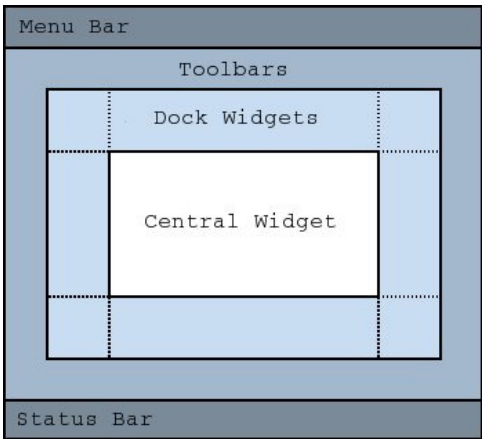


Abbildung 13.3: Aufbau der Klasse QMainWindow

Wird eine Applikation mit QT erstellt, so wird die MainWindow Klasse der Applikation von der Klasse `QMainWindow` abgeleitet.

Das folgende Beispiel zeigt das Grundgerüst einer mit QT erstellten Applikation.

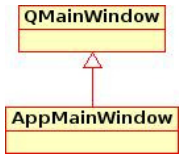


Abbildung 13.4: Grundgerüst für eine QT Applikation

```
1  #ifndef APPMAINWINDOW_H
2  #define APPMAINWINDOW_H
3
4  #include <QMainWindow>
5
6  class AppMainWindow : public QMainWindow {
7
8  };
9
10 #endif
```

```
#include <QApplication>
#include "appmainwindow.h"

int main(int argc, char *argv[]){
  QApplication app(argc, argv);
  AppMainWindow mainWindow;
  mainWindow.show();
  return app.exec();
}
```

Wird dieses Programm gestartet, so erscheint ein leeres Fenster.

Abbildung 13.5: QT Applikationsgrundgerüst

13.3.2 Layout Manager

Um Widget sinnvoll anzuordnen, stellt QT Layoutmanager zur Verfügung. Die wichtigsten sind

- `QHBoxLayout` und `QVBoxLayout`
- `QGridLayout`
- `XYLayout (kein Layout)`

QHBoxLayout und QVBoxLayout

Mit den Klassen `QHBoxLayout` und `QVBoxLayout` können Widgets horizontal und vertikal angeordnet werden. Die Klassen können auch miteinander verschachtelt werden.

Horizontale Anordnung

Der folgende Code zeigt ein Beispiel für die Verwendung der Klasse `QHBoxLayout`, um Widgets horizontal anzuordnen.

```
1  // QHBoxLayout
2
3  #include <QApplication>
4  #include <QPushButton>
5  #include <QHBoxLayout>
6
7  int main(int argc, char *argv[]){
8    QApplication app(argc, argv);
9
10   // Horizontal Layout Mananger
11   QWidget *mainWidget = new QWidget();
12
13   QPushButton *button1 = new QPushButton("Button 1");
14   QPushButton *button2 = new QPushButton("Button 2");
15   QPushButton *button3 = new QPushButton("Button 3");
16
17   QHBoxLayout *hBoxLayout = new QHBoxLayout();
18   hBoxLayout->addWidget(button1);
19   hBoxLayout->addWidget(button2);
20   hBoxLayout->addWidget(button3);
21
22   mainWidget->setLayout(hBoxLayout);
23
24   mainWidget->show();
25   return app.exec();
26 }
```

Abbildung 13.6: Layout mit QHBoxLayout

Vertikale Anordnung

Der folgende Code zeigt ein Beispiel für die Verwendung der Klasse `QVBoxLayout`, um Widgets vertikal anzuordnen.

```
1  // QVBoxLayout
2
3  #include <QApplication>
4  #include <QPushButton>
5  #include <QVBoxLayout>
6
7  int main(int argc, char *argv[]){
8    QApplication app(argc, argv);
9
10   // Vertikal Layout Mananger
11   QWidget *mainWidget = new QWidget();
12
13   QPushButton *button1 = new QPushButton("Button 1");
14   QPushButton *button2 = new QPushButton("Button 2");
15   QPushButton *button3 = new QPushButton("Button 3");
16
17   QVBoxLayout *vBoxLayout = new QVBoxLayout();
18   vBoxLayout->addWidget(button1);
19   vBoxLayout->addWidget(button2);
20   vBoxLayout->addWidget(button3);
21
22   mainWidget->setLayout(vBoxLayout);
23
24   mainWidget->show();
25   return app.exec();
26 }
```

Abbildung 13.7: Layout mit QVBoxLayout

QGridLayout

Mit der Klasse `QGridLayout` können Widgets in einem Gitter angeordnet werden.

Der folgende Code zeigt ein Beispiel für die Verwendung der Klasse `QGridLayout`, um Widgets in einem Gitter anzuordnen.

```
1  // QGridLayout
2
3  #include <QApplication>
4  #include <QPushButton>
5  #include <QLineEdit>
6  #include <QGridLayout>
7
8  int main(int argc, char *argv[]){
9    QApplication app(argc, argv);
10
11   // Grid Layout Mananger
12   QWidget *mainWidget = new QWidget();
13
14   QPushButton *button1 = new QPushButton("Button 1");
15   QPushButton *button2 = new QPushButton("Button 2");
16   QLineEdit *lineEdit1 = new QLineEdit();
17   QLineEdit *lineEdit2 = new QLineEdit();
18
19   QGridLayout *gridLayout = new QGridLayout();
20   gridLayout->addWidget(button1, 0, 0);
21   gridLayout->addWidget(lineEdit1, 0, 1);
22   gridLayout->addWidget(button2, 1, 0);
23   gridLayout->addWidget(lineEdit2);
24
25   mainWidget->setLayout(gridLayout);
26
27   mainWidget->show();
28   return app.exec();
29 }
```

Abbildung 13.8: Layout mit QGridLayout

XYLayout

Widgets können auch ohne Verwendung eines Layout Managers angeordnet werden. Dazu muss allen Widget die Position sowie die Höhe und Breite mitgegeben werden. Die erfolgt mit Hilfe der Methode `setGeometry`. So kann ein Widget an einem beliebigen Ort platziert werden.
Ohne Layoutmanager gibt es jedoch auch Nachteile. Wird die Fenstergrösse der Applikation geändert, so passsen sich die Widgets nicht der neuen Grösse an.

Der folgende Code zeigt ein Beispiel für die Anordnung von Widgets ohne Layout Manager.

```
1  // XY - Layout
2
3  #include <QApplication>
4  #include <QPushButton>
5
6  int main(int argc, char *argv[]){
7    QApplication app(argc, argv);
8
9    QWidget *mainWidget = new QWidget();
10
11   QPushButton *button1 = new QPushButton(
12     "Button 1", mainWidget);
13   QPushButton *button2 = new QPushButton(
14     "Button 2", mainWidget);
15   QPushButton *button3 = new QPushButton(
16     "Button 3", mainWidget);
17
18   button1->setGeometry(5, 5, 100, 20);
19   button2->setGeometry(30, 40, 60, 50);
20   button3->setGeometry(100, 40, 60, 30);
21
22   mainWidget->show();
23   return app.exec();
24 }
```

Abbildung 13.9: Layout mit XYLayout

13.4 Signals und Slots

Jedes Widget in QT kann einen Event auslösen. Dieser Event wird in Form eines Signal an ein anderes Widget gesendet. Empfangen wird das Signal mit einem Slot.
Welche Signales und Slots ein Widget zur Verfügung stellt kann in der QT API nachgelesen werden. Hier sollen nun die Signale der Klasse `QPushButton` betrachtet werden. Dies sind:

- `clicked(bool checked=false)`
- `pressed()`
- `released()`
- `toggled(bool checked)`

Wird nun der Button gedrückt, so wird das Signal *pressed* gesendet. Wird der Button losgelassen, so wird das Signal *released* gesendet. Beides zusammen, drücken und loslassen, sendet das Signal *clicked*.
Diese Signale können nun von Slots aufgefangen werden.

⋆ *Merke:*
Es können nur Signale und slot verbunden werden, welche die gleiche Parameterliste aufweisen. D.h. falls das Signal einen Parameter mitsendet, so muss auch der entsprechende Slot diesen Parameter aufnehmen können.

Ein Signal wird folgendermassen mit einem Slot verbunden:

```
QObject::connect(
  senderObject, SIGNAL(signal),
  receiverObject, SLOT(slot));
```

Beispiel:

Die folgende Applikation besteht aus einem Button. Wird dieser Button gedrückt, so soll die Applikation geschlossen werden. Das Signal `clicked` wird dabei mit dem Slot `closeAllWindows` verbunden. Der Sender ist das Button Objekt, der Empfänger ist das QApplications Objekt.

```
// Signal - Slot Example

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]){
  QApplication app(argc, argv);
  QPushButton *button = new QPushButton("Push Me!");
  QObject::connect(
    button, SIGNAL(clicked()),
    &app, SLOT(closeAllWindows()));
```

```
12   button->show();
13   return app.exec();
14 }
```

Das folgende Bild zeigt nochmal das Prinzip der Verbindung von Signals und Slots.

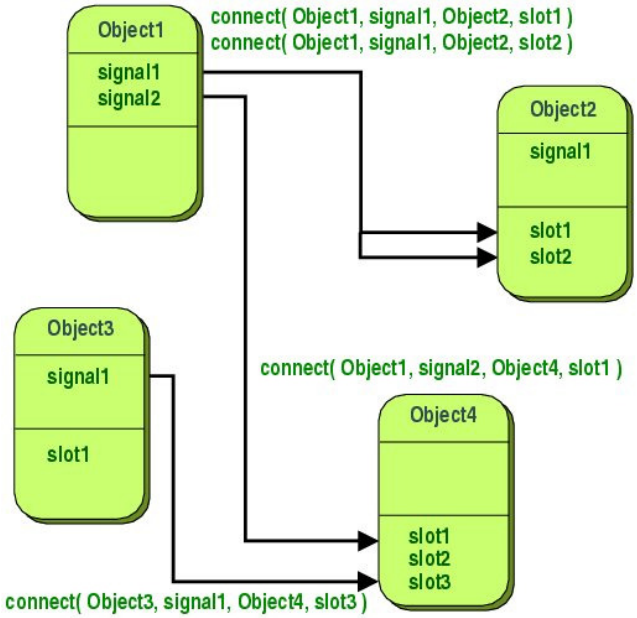


Abbildung 13.10: Signals und Slots

13.4.1 Eigene Slots

Wird eine Klasse implementiert, welche einen Slot zur Verfügung stellt, müssen die folgenden Bedingungen erfüllt sein:

- Die Klasse muss direkt oder indirekt von der Klasse `QObject` abgeleitet werden
- Die Klasse muss das Makro `Q_OBJECT` beinhalten
- Die Slots müssen im Bereich `public slots` implementiert werden

Der folgende Code zeigt das Grundgerüst einer Klasse, welche einen Slot zur Verfügung stellt.

```
1  class MySlot : public QObject {
2
3    Q_OBJECT
4
5  public:
6    ...
7  private:
8    ...
```

```
9  public slots:
10   void slotName();
11
12 };
```

Beispiel:

Die folgende Applikation besteht aus einer Klasse `Gui` und einer Klasse `Controller`. Die graphische Benutzerschnittstelle besteht aus einem Button und einem Eingabefeld. Wird der Button gedrückt, so soll der Inhalt im Eingabefeld gelöscht werden. Der Slot wird von der Klasse `Controller` zur Verfügung gestellt. Hier ist es wichtig, dass die Klasse `Controller` auf Methoden der Klasse `Gui` zugreifen kann, da der Slot ja eine Änderung in der Klasse `Gui` vornehmen soll.

```
1  // gui.h
2  #ifndef GUI_H
3  #define GUI_H
4
5  #include <QWidget>
6
7  class QPushButton;
8  class QLineEdit;
9  class Controller;
10
11 class Gui : public QWidget {
12 private:
13   QPushButton *button;
14   QLineEdit *input;
15   Controller *controller;
16
17 public:
18   Gui();
19   void clearInput();
20 };
21
22 #endif
```

```
1  // gui.cpp
2  #include "gui.h"
3  #include "controller.h"
4  #include <QPushButton>
5  #include <QLineEdit>
6  #include <QVBoxLayout>
7
8  Gui::Gui(){
9    button = new QPushButton("Clear");
10   input = new QLineEdit();
11   QVBoxLayout *layout = new QVBoxLayout();
```

```
12   layout->addWidget(button);
13   layout->addWidget(input);
14   setLayout(layout);
15   controller = new Controller(this);
16   QObject::connect(
17     button, SIGNAL(clicked()),
18     controller, SLOT(clearInput()));
19 }
20
21 void Gui::clearInput(){
22   input->clear();
23 }
```

```
1  // controller.h
2  #ifndef CONTROLLER_H
3  #define CONTROLLER_H
4
5  #include <QObject>
6
7  class Gui;
8
9  class Controller : public QObject {
10
11 Q_OBJECT
12
13 private:
14   Gui *parent;
15
16 public:
17   Controller(Gui *gui);
18
19 public slots:
20   void clearInput();
21 };
22
23 #endif
```

```
1  // controller.cpp
2  #include "controller.h"
3  #include "gui.h"
4
5  Controller::Controller(Gui *gui) {
6    parent = gui;
7  }
8
9  void Controller::clearInput() {
10   parent->clearInput();
11 }
```

13.4.2 Eigene Signals

13.5 Menus

13.6 Zeichnen

Ein Widget kann auch als Zeichnungsbrett verwendet werden. Jedesmal wenn ein Widget dargestellt wird, so wird die Methode `paintEvent` dieses Widgets aufgerufen. Diese Methode kann nun in einem eigenen Widget überschrieben werden. Danach können in dieser Methode mit Hilfe eines `QPainter` Objektes beliebige Dinge gezeichnet werden.

```
1 class PaintWidget : public QWidget {
2 public:
3   void paintEvent(QPaintEvent *event);
4 };
5
6 void PaintWidget::paintEvent(QPaintEvent *event){
7   QPainter painter(this);
8   ...
```

Mit dem `QPainter` Objekt können nun beliebige Dinge gezeichnet werden:

- `drawArc(...)`
- `drawLine(...)`
- `drawPoint(...)`
- `...`

Alle Methoden mit Parameter können in der QT Hilfe der Klasse `QPainter` nachgeschaut werden.

13.7 Aufgaben

13.7.1 Sierpinski Dreieck

Gegeben sind drei Punkte in der Ebene (diese Punkte werden nicht gezeichnet, nur als Variablen deklariert).

- A (10, 10)
- B (280, 100)
- C (130, 290)

Nun sollen 100000 Punkte gemäss dem folgenden Algorithmus gezeichnet werden.

1. Setze die Variablen `x` und `y` auf den Wert 0
2. Bestimme zufällig einen Punkt A, B oder C
3. Berechne den Mittelpunkt `xm, ym` zwischen `x, y` und dem in Schritt 2 gewählten Punkt
4. Setze `x` auf den Wert von `xm`
5. Setze `y` auf den Wert von `ym`
6. Zeichne einen Punkt an der Stelle `x, y`
7. Gehe zu Schritt 2

13.7.2 Moiree

Erstellen Sie mit QT ein Programm, welches in einem Widget das folgende Muster zeichnet.

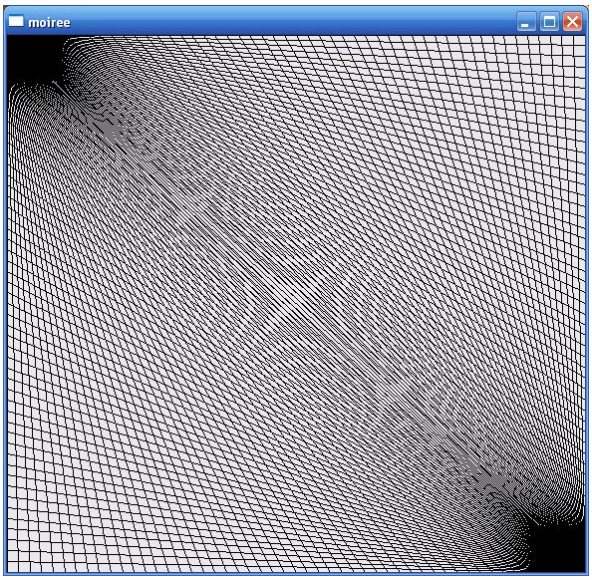


Abbildung 13.11: Moiree

Literaturverzeichnis

[1] **Nicolai Josuttis:** *Objektorientiertes Programmieren in C++*, Addison Wesley, 1994

[2] **Helmut Herold:** *Das QT Buch*, SuSE PRESS, 2001

[3] **Robert Sedgewick:** *Algorithmen in C++*, Addison Wesley, 1992

[4] **Dirk Louis:** *Easy C++*, Markt+Technik, 2001

[5] **Schader, Kuhlins:** *Programmieren in C++*, Springer, 1994

[6] **Liberty, Jesse:** *C++ in 21 Tagen*, Markt und Technik, 1999

[7] **Ottmann, Widmayer:** *Algorithmen und Datenstrukturen*, Spektrum, 2002

Abbildungsverzeichnis