The C++ Study Guide
KTSI

David Herzig

Juni 2005

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 FHBB und 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.
`d.herzig@bluewin.ch`

0.2 Danke

An dieser Stelle möchte ich mich bei allen bedanken, die mir beim Schreiben dieses Scriptes behilflich waren.

0.3 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 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

2.2 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

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

2.3 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. $log_a \sqrt[v]{u} = (\frac{1}{n}) \cdot log_a u$

2.4 Fakultät

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

Ergänzung: $0! = 1$

2.5 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 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 using namespace std Anweisung weggelassen, so muss das Kommando cout mit std::cout angesprochen werden.

```
1 int main()
```

Diese Zeile definiert den Startpunkt des Programms. Es spielt keine Rolle wo im Code es 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 VariablenName;
```

Beispiele:

```
1 float x; // Deklaration einer float Variablen mit dem Namen x
2 int m,n; // Deklaration von zwei int Variablen
3 double pi = 3.141, g; //Zwei double Variablen mit einer 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:

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

3.3 Operatoren

3.3.1 Zuweisung

Mit dem Zuweisungsoperator = können wir Werte zu Variablen zuweisen.

```
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:

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

Diese Anweisung ist äquivalent mit

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

3.3.2 Arithmetische Operatoren

In C++ existieren die folgenden arithmetischen Operatoren:

- `+` Addition
- `-` Subtraktion
- `*` Multiplikation
- `/` Division
- `%` Modulo

Beipiele:

```
a = b + c;
x = 3.141 * 5 - (m % n);
```

3.3.3 Vergleichsoperatoren

In C++ existieren die folgenden Vergleichsoperatoren:

- `==` Gleich
- `!=` Nicht gleich
- `>` 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	true

3.3.6 sizeof

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

```
int a,b;

// In a werden die Anzahl Bytes welche b
// verwendet gespeichert.
a = sizeof(b);

// In a werden die Anzahl Bytes einer
// double Variablen gespeichert.
a = sizeof(double);
```

Beispiel:

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

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

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

Wertebereich:

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

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

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.

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

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

```
1 cout << "C++" << " ist so toll" << endl;
```

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

```
1 cout << a << endl; // Gibt den Inhalt von a aus
2 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:

```
1 // Speicherung eines Wertes von der Konsole in der Variable a
2 cin >> a;
```

Beispiel:

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

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int i;
6      cout << "i= ";
7      cin >> i;
8      cout << "The value you entered is " << i << endl;
9      return 0;
10 }
```

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);

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

    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, dass die Speichergrösse der folgenden Variablen 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:

$$T = \sqrt{\frac{2h_0}{g}}$$

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

Schreiben Sie ein Programm, das die Höhe h_0 einliest und den Wert der Dauer des Falles ausgibt.

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 Sequnez
```

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.

```
1  int day;
2
3  switch(day){   // Abfragen der Integer Variable day
4
5    case 0:  cout << "Sonntag" << endl;
6             break;
7
8    case 1:  cout << "Montag" << endl;
9             break;
10
11   ...
12
13   default: cout << "Not a valid day" << endl;
14 }
```

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
- Bedingung
- Regel

```
for (init; condition; rule)
```

Ausführungsschritte:

1. Ausführung des Initial Statements
2. Überprüfen der Schleifenbedingung
3. Ausführen der Schleife
4. Ausführen der Regel
5. Bedingung prüfen
6. Gehe zu Schritt 3

Beispiel:

Implementierung eines Countdowns, welcher von 10 rückwärts zählt und die einzelnen Werte jeweils auf dem Bildschirm ausgibt. Bei dem Wert 0 soll das Wort `FIRE` ausgegeben werden.

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

int main(){
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!";
  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.

Beispiel:

Berechnung der harmonischen Reihe.

Harmonische Reihe:

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

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    float summe = 0, glied = 1.0;
6    int i = 1;
7
8    while (glied > 0.001){
9        summe = summe + glied;
10       i++;
11       glied = 1.0 / i;
12   }
13
14   cout << summe << endl;
15
16   return 0;
17 }
```

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.

Beispiel:

Der Benutzer soll ganzzahlige Werte eingeben. Diese sollen gleich wieder auf dem Bildschirm erscheinen. Gibt der Benutzer eine `0` ein, so wird das Programm beendet.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    unsigned long n;
6
7    do {
8      cout << "Enter number (0 to end): ";
9      cin >> n;
10     cout << "You entered: " << n << "\n";
11   } while (n != 0);
12
13   return 0;
14 }
```

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 sofort mit der nächsten fortgefahren. Auch diese Anweisung sollte nur in Sonderfällen eingesetzt werden.

Beispiel:

Implementierung eines Programmes, welches 10 Zahlen einliest, und die kleinste dieser 10 Zahlen auf dem Bildschirm ausgibt.

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5
6      int s, input;
7      cin >> s;
8
9      for (int i=0; i<9; i++){
10         cin >> input;
11         if (input < s) s = input;
12     }
13
14     cout << s << endl;
15
16     return 0;
17 }
```

4.2 Funktionen

Eine Funktion in C++ ist folgendermassen aufgebaut:

```
resultType functionName (parameter){

    // Code

    return returnValue;

}
```

Beispiel:

Berechnung der harmonischen Reihe mit einer Funktion.

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

float harmonischeReihe(int anzahlGlieder){
    float summe = 1;
    int i = 2;
    while (i <= anzahlGlieder){
        summe = summe + 1.0/i;
        i++;
    }
    return summe;
}
```

4.2.1 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 = harmonischeReihe(1000);
```

```
// Aufruf ohne Verwendung des Resultates
harmonische Reihe(1000);
```

4.2.2 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.

4.2.3 Return Statement

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

```
1 return returnValue;
```

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

4.2.4 Statische lokale Variablen

Wird ein Unterprogramm 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 des Unterprogrammes nicht gelöscht, sondern bleibt im Speicher und kann beim nächsten Aufruf wieder verwendet werden. Sie ist aber trotzdem nur im Unterprogramm bekannt.

```
1 static int i;
```

4.2.5 Funktionen ohne Resultat

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

```
1 void functionName (parameter){
2
3   // Code
4
5 }
```

Beispiel:

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

```
1  #include <iostream>
2  using namespace std;
3
4  int ggt(int a, int b){
5    while (a !=b){
6      if (a < b){
7          b = b - a;
8      }
9      else {
10         a = a - b;
11     }
12   }
13    return a;
14 }
```

4.2.6 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.

```
1  #include <iostream>
2  using namespace std;
3
4  void foo(int m){
5      // code
6  }
7
8  void foo(float x, int n){
9      // code
10 }
11
12 void foo(float x, float y){
13     // code
14 }
```

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

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;
```

4.3 Struktogramme

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

4.3.1 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.

Sequenz

Lineare Abfolgen werden durch lückenloses Untereinandersetzen von Anweisungen ausgedrückt.

Abbildung 4.1: Struktogramm: Einfache Anweisung

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

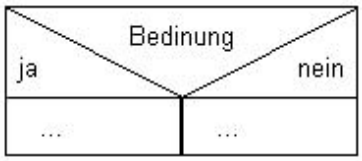


Abbildung 4.2: Struktogramm: Selektion

Mehrfach if Selektion, switch Statement

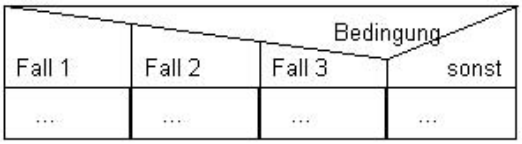


Abbildung 4.3: Struktogramm: Mehrfache Selektion

Beispiel:

Überprüfen ob ein Jahr ein Schaltjahr ist.

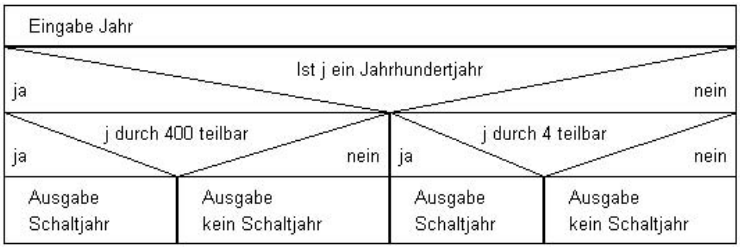


Abbildung 4.4: Struktogramm: Schaltjahrtest

Schleifen

Es existieren zwei Darstellungsarten für Schleifen:

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

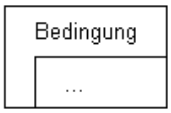


Abbildung 4.5: Struktogramm: Abweisende Schleife

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

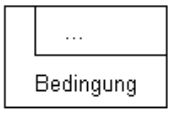


Abbildung 4.6: Struktogramm: Annehmende Scheife

Beispiel:

Überprüfung ob ein Rechteck ein Quadrat ist.

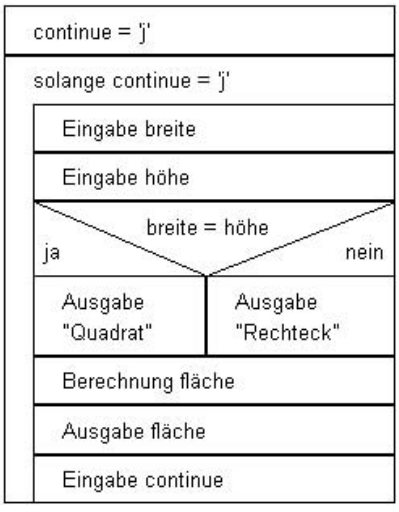


Abbildung 4.7: Struktogramm: Rechteck - Quadrat Test

4.4 Aufgaben

4.4.1 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:

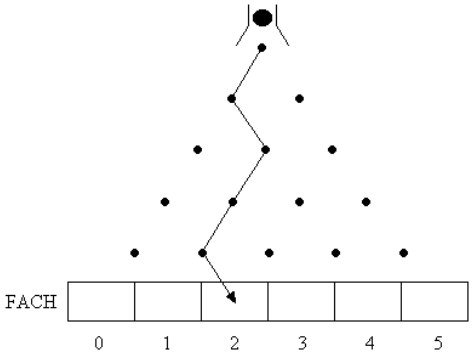
$$D = b^2 - 4 \cdot a \cdot c$$

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.

4.4.2 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.



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.

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 vektor[3];
```

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

`vektor[0],vektor[1],vektor[2]`

⋆ *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:

```
1 float vektor[3] = {1.2, 4.2, 9.4};
```

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

```
1 float vektor[] = {1.2, 4.2, 9.4};
```

⋆ *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.

```
1 float arrayName[1000];
2 for (int i=0; i<1000; i++){
3     arrayName[i] = ....
4 }
```

5.1.3 Zweidimensionale Array

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

```
1 dataType arrayName[zeilen][spalten];
```

Definition einer Float Matrix mit 3 Zeilen und 4 Spalten:

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

Elemente der Matrix:

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

Initialisierung bei der Definition

```
1 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[n_zeilen][n_spalten];
2
3 for (int i=0; i<n_zeilen; i++){
4     for (int j=0; j<n_spalten; j++){
5
6         matrix[i][j] = ...
7
8 }
```

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 `inclde` Anweisung im Source Code ergänzt werden:

```
1 #include <vector>
```

Die Anweisung

```
1 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.

```
1 v[0] = 12;
2 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.

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

Ein Vektor kann auch nach seiner Grösse abgefragt werden.

```
1 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 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 }
```

String zeichenweise ausgeben (mit Indexprüfung)

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

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 int n = output.length(); // In n befindet sich die Anzahl Zeichen
```

5.3.3 Zusammenfügen von Strings

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

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

5.3.4 String kopieren

Ein String kann folgendermassen kopiert werden:

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

5.3.5 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.

```
int stringMatch(string text, string pattern){
    int tlen = text.size();
    int plen = pattern.size();
    int i, j, skip[1024];

    for(i=0; i<1024; i++)  skip[i]=plen;
    for(i=0; i<plen; i++) skip[pattern[i]]=plen-1-i;

    for(i=j=plen-1; j>=0; i--, j--)
        while(text[i] != pattern[j]) {
            i += max(plen-j,skip[text[i]]);
            if(i >= tlen) return -1;
            j = plen-1;
        }

    return i;
}
```

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

5.4 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).

5.4.1 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 *variablenname;
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
```

5.4.2 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
```

5.4.3 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.

5.4.4 Die Speicheradresse NULL

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

```
1 float *p;
2 p = NULL;
3 ...
4 if (p == NULL) ...
```

5.4.5 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 *float_ptr;
2 char *char_ptr;
3 char_ptr = (char *)(float_ptr);
```

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

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

5.4.7 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_wert);  // 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!
```

5.4.8 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 << "FEHLER" << endl;
14 }
15
16 int main(){
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.

5.5 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
```

5.5.1 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.

```
1  #include <iostream>
2  using namespace std;
3
4  void func(int &n , int m){
5
6      n = 4;
7      m = 5;
8
9  }
10
11 int main(){
12
13     int a = 2, b = 3;
14
15     func(a,b);
16
17     cout << a << endl;
18     cout << b << endl;
19
20     return 0;
21 }
```

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.

5.6 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.

5.6.1 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 `NULL` zurück.

Beispiel:

Dynamische Erzeugung einer float Variablen

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

int main(){

    float *xp;
    xp = new float;

    if (xp == NULL){
        // 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.

5.6.2 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.

5.6.3 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` aufrufen.

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

5.6.4 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;
```

5.6.5 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 }
```

5.7 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};
```

5.7.1 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
```

5.8 Benutzerdefinierte Datentypen

5.8.1 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(){

    // 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;
}
```

5.9 Aufgaben

5.9.1 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 }
```

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.

```
1  class Product{
2
3  public:  // Nun sind alle folgenden Deklarationen public
4
5      string name;
6      float price;
7
8  private:  // Nun sind alle folgenden Deklarationen private
9
10     int number;
11 };
```

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:

```
1  class Product{
2
3  public:  // Nun sind alle folgenden Deklarationen public
4
5      string name;
6      float price;
7
8      // Deklaration einer Funktion in einer Klasse
9      void displayNumber(){
10       cout << "Number = " << number << endl;
11     }
12
13 private:  // Nun sind alle folgenden Deklarationen private
14
15     int number;
16 };
```

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 ein Objekt der Klasse `Products` angelegt werden, damit diese Methode aufgerufen werden kann.

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

Zugriff auf die Methode 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
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.

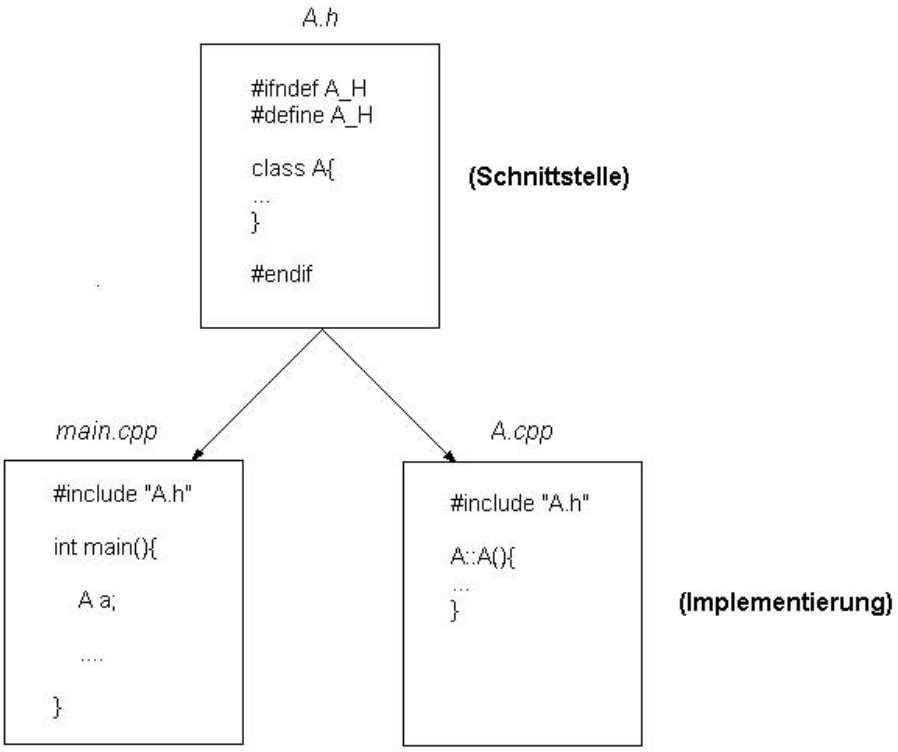


Abbildung 6.1: Aufteilung Deklaration und Implementierung

Beispiel:

Implementierung einer Klasse für Rechtecke.

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

class Rectangle {
private:
    int x, y;
public:
    void set_values(int,int);
    int area();
};

int Rectangle::area(){
    return x*y;
}

void Rectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main(){
    Rectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area() << endl;
    return 0;
}
```

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:

Klasse Rectangle

```
#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    int const 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    int const c;
5  public:
6    Klasse(int i);
7  };
8
9  A::A(int i)
10   : r(i), c(i){
11
12 }
```

▷ *Merke:*
Die Reihenfolge der Initialisierungen hängt von der Reihenfolge der Deklarationen (in der Klasse) ab. Wenn wir also `A(int i) : c(i), r(i) {}` schreiben würden, 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(){
8      A einObjekt;                // Aufruf default Konstruktor
9      A neuesObjekt(einObjekt); // Copy Konstruktor
10     return 0;
11 }
```

Der Copy Konstruktor kopiert nun die Werte aller Elemente in das neu erzeugte Objekt.

Was passiert wenn die Klasse dynamisch Speicher alloziert hat?

```
1  class A{
2  public:
3      int *m; // Irgendwo steht m = new int;
4      ...
5  };
6
7  int main(){
8      A einObjekt;                // Aufruf default Konstruktor
9      A neuesObjekt(einObjekt); // Copy Konstruktor
10     return 0;
11 }
```

Jetzt besitzen wir zwei Objekte der Klasse `A`. Jedes Objekt besitzt einen Pointer `*m`. Da der Copy Konstruktor die Werte aller Elemente 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 Integer Variable. Dies ist natürlich schlecht! Wenn ein Objekt den Wert dieser Variablen ändert, ist er gleich für alle Objekte der selben Klasse geändert.

Also müssen wir den Copy Konstruktor überladen und ihn so implementieren, das er 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(){
14     A einObjekt;                // Aufruf default Konstruktor
15     A neuesObjekt(einObjekt); // Copy Konstruktor
16     return 0;
17 }
```

Jetzt wird beim Kopieren eines Objektes zuerst eine eigene dynamische Integer 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.
Graphische Darstellung des Speichers eines falschen und eines richtigen Copykonstruktor.

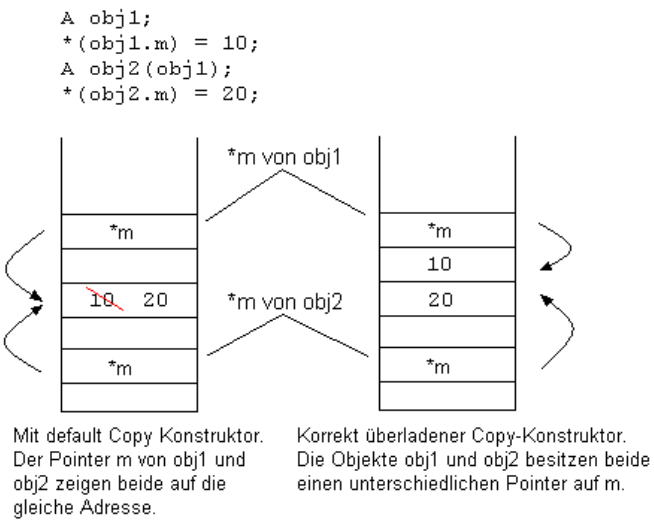


Abbildung 6.2: Copy Konstruktor

6.2.3 Destruktor

Der Destruktor ist eine Methode, welche die folgenden Eigenschaften besitzt:

- Keinen Rückgabewert
- Methodenname = ~Klassenname
- Keine Parameter

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

Was kann nun im Destruktor implementiert werden? 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 }
```

Wenn wir nun ein Objekt der Klasse `A` anlegen, haben wir also im Speicher auch eine dynamisch erzeugte float Variable. 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 angesprochen werden. Also müssen wir jedesmal wenn wir ein `A` Objekt zerstören, auch alle dynamisch erzeugten Elemente innerhalb der Klasse zerstören.

```
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:*
Wurde in der Klasse dynamisch Speicher alloziert, so müssen der Copy Konstruktor und der Destruktor implementiert werden.

6.3 Pointer auf Klassen

Es können ohne weiteres auch Pointers auf Klassen deklariert werden. Der folgende Code zeigt ein Beispiel:

```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle{
5  private:
6      int width, height;
7  public:
8      void set_values(int, int);
9      int area();
10
11 void Rectangle::set_values (int a, int b) {
12     width = a;
13     height = b;
14 }
15
16 int Rectangle::area(){
17     return width * height;
18 }
19
20 int main () {
21     Rectangle a, *b, *c;
22     Rectangle * d = new Rectangle[2];
23     b= new Rectangle;
24     c= &a;
25     a.set_values(1,2);
26     b->set_values(3,4);
27     d->set_values(5,6);
28     d[1].set_values(7,8);
29     cout << "a area: " << a.area() << endl;
30     cout << "*b area: " << b->area() << endl;
31     cout << "*c area: " << c->area() << endl;
32     cout << "d[0] area: " << d[0].area() << endl;
33     cout << "d[1] area: " << d[1].area() << endl;
34     return 0;
35 }
```

Dieser Code erzeugt den Output

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

6.4 Konstante Methoden

In C++ können auch Methoden einer Klasse das const-Schlüsselwert haben. Dies bedeutet, daß diese Methode das Objekt, über das sie aufgerufen wird, nicht verändert.

```
1  class MyClass{
2  private:
3    int value;
4  public:
5    // die Methode getValue veraendert keine Daten in MyClass
6    int getValue() const;
7  };
8
9  int MyClass::getValue(){
10   return value;
11 }
```

6.5 Datenkapselung

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

Weiter soll ein Element, welches von aussen zugänglich sein muss, auch nicht als `public` deklariert werden. Es sollen entsprechende Methoden implementiert werden um das Element zu lesen und zu schreiben. Dies ermöglicht einen kontrollierten Zugriff auf die Elemente über eine definierte Schnittstelle. Andernfalls kann ein Objekt in einen ungültigen Zustand gelangen.

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

Get Methoden:

```
datentyp getElementname();
```

Set Methoden:

```
void setElementname(datentyp m);
```

Beispiel: Klasse Kreis

```
class Kreis{
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 Elemente einer Klasse mit get und set Methoden.

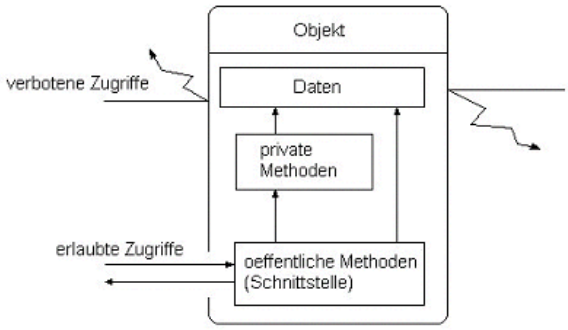


Abbildung 6.3: Datenkapselung

6.6 Überladen von Operatoren

Betrachten Sie die folgenden Klasse Bruch.

```
1  class Bruch{
2  private:
3      int nenner;
4      int zaehler;
5  public:
6      Bruch(int,int);
7      void displayBruch();
8  };
9
10 Bruch::Bruch(int z, int n)
11   :nenner(n), zaehler(z){
12 }
13
14 void Bruch::displayBruch(){
15     cout << zaehler << "/" << nenner << endl;
16 }
```

Mit dieser Klasse können Bruch-Objekte mit Zähler und Nenner erzeugt werden. Nun möchten Sie eine Methode entwicklen, die zwei Brüche miteinander addiert. Nichts einfacher als das:

```
1  class Bruch{
2  private:
3      int nenner;
4      int zaehler;
5  public:
6      Bruch(int,int);
7      void displayBruch();
8      Bruch addition(Bruch);
9  };
10
11 Bruch::Bruch(int z, int n)
12   :nenner(n), zaehler(z){
13 }
14
15 void Bruch::displayBruch(){
16     cout << zaehler << "/" << nenner << endl;
17 }
18
19 Bruch Bruch::addition(Bruch b){
20     int z = nenner * b.zaehler + zaehler * b.nenner;
21     int n = nenner * b.nenner;
22     Bruch result(z,n);
23     return result;
24 }
```

Jetzt wäre es allerdings schöner, wenn wir statt

```
b3 = b1.addition(b2);
b3 = b1 + b2;
```

schreiben könnten. Allerdings gibt dies eine Fehlermeldung des Compilers. Klar, denn der Computer weiss ja auch nicht wie er zwei Brüche miteinaner addieren sollen. Dies müssen wir ihm mitteilen. Dazu überladen wir den Operator `+`. Der neue Code sieht dann folgendermassen aus:

```
class Bruch{
private:
    int nenner;
    int zaehler;

public:
    Bruch(int,int);
    void displayBruch();
    Bruch operator +(Bruch);
};

Bruch::Bruch(int z, int n)
  :nenner(n), zaehler(z){
}

void Bruch::displayBruch(){
    cout << zaehler << "/" << nenner << endl;
}

Bruch Bruch::operator +(Bruch b){
    int z = nenner * b.zaehler + zaehler * b.nenner;
    int n = nenner * b.nenner;
    Bruch result(z,n);
    return result;
}
```

Das einzige was ausgetauscht wurde, war der Methodenname `addition` mit `operator +`. Äquivalent können so auch der Operator `-`, `*` und `/` überladen werden.

6.6.1 Der Gleichheitsoperator

Das Überladen des Gleichheitsoperators erfolgt sehr ähnlich wie das Überladen des Copy Konstruktors. Für Klassen, welche keinen dynamischen Speicher alloziert hat, funktioniert der default Gleichheitsoperator einwandfrei. Sobald jedoch dynamisch Speicher alloziert wurde, muss zuerst der dynamische Speicher des Objektes, welches überschrieben wird, freigegeben werden, danach muss neuer Speicherplatz reserviert werden und zum Schluss müssen dann noch die Werte kopiert werden.

```
1  #include <iostream>
2  using namespace std;
3
4  class Vector{
5  private:
6      int dimension;
7      float *values;
8  public:
9      Vector(int);
10     Vector operator = (const Vector&);
11 };
12
13 Vector::Vector(int d)
14   :dimension(d){
15 }
16
17 Vector Vector::operator = (const Vector &v){
18     delete [] values;              // Alter Speicher loeschen
19     dimension = v.dimension;
20     values = new float[dimension]; // Neuer Speicher reservieren
21
22     // Werte kopieren
23     for (int i=0; i<dimension; i++){
24         values[i] = v.values[i];
25     }
26
27     return *this;
28 }
```

In dieser Klasse wird ein Vector mit Hilfe eines Arrays implementiert. Der Array hat die Grösse wie die Dimension des Vectors. Wird nun der Gleichheitsoperator verwendet

```
Vector v1(2),v2(5);
...
v1 = v2
```

heisst das, dass die Werte in `v1` überschrieben werden. Wir müssen davon ausgehen, das `v1` und `v2` nicht die gleiche Dimension haben. Das heisst, wir müssen den dynamisch erzeugten Array von `v1` erstmal löschen. Um nun die Werte von `v2` zu kopieren, müssen wir wieder ein Array anlegen. Weiter müssen die Werte von `v2` nach `v1` kopiert werden, und zum Schluss muss die Methode ein Objekt auf den Bruch zurückgeben,

welchen den Gleichheitsoperator verwendet hat.

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

6.7 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 n;
4  public:
5      setN(int n){
6          this->n = n;
7          // n bezieht sich auf den Parameter
8          // this->n bezieht sich auf das Attribut der Klasse
9      }
10 };
```

6.8 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 ich also bei einem Objekt den Wert von `a` ändere, ist der Wert auch bei allen anderen Objekte 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(){
12     A a1, a2;  // Erzeugen von 2 A Objekten
13     a1.a = 20;
14     a2.a = 10;
15
16     cout << a1.a << endl;   // Da nur ein a existiert und dieses
17                             // von a2 zuletzt geaendert wurde.
18
19     A::a = 3; // Zugriff ohne Objekt
20
21     return 0;
22 }
```

`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.

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5  public:
6      static void display();
7  };
8
9  void A::display(){
10     cout << "Hello World" << endl;
11 }
12
13 int main(){
14
15     A::display(); // Aufruf ohne Objekt
16
17     A a;
18     a.display(); // Aufruf mit Objekt
19
20     return 0;
21 }
```

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

6.9 Friend

6.9.1 Friend Funktionen

Von ausserhalb einer Klasse besitzt man kein Zugriff auf Elemente innerhalb der Klasse, welche als `private` oder `protected` deklariert sind.
Jetzt können wir eine ganz normale Funktion definieren, welcher wir dann innerhalb eine Klasse als `friend` deklarieren. Das heisst, diese Funktion hat dann Zugriff auf alle Elemente innerhalb dieser Klasse.

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5  private:
6      int n;
7  public:
8      A(int);
9      friend void display(A); // Funktion als friend deklarieren
10 };
11
12 A::A(int n){
13     this->n = n;
14 }
15
16 // Diese Funktion hat Zugriff auf die privaten Elemente
17 // von A, da sie in der Klasse als friend deklariert wurde
18 void display(A obj){
19     cout << obj.n << endl;
20 }
```

Beispiel:

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

```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle{
5  private:
6      int width, height;
7  public:
8      void set_values (int, int);
9      int area () {return (width * height);}
10     friend Rectangle duplicate(Rectangle);
11 };
12
13 void Rectangle::set_values (int a, int b){
14     width = a;
15     height = b;
16 }
17
18 // Diese Funktion wurde als friend in der Klasse Rectangle deklariert
19 Rectangle duplicate (Rectangle rectparam){
20     Rectangle rectres;
21     rectres.width = rectparam.width*2;
22     rectres.height = rectparam.height*2;
23     return (rectres);
24 }
25
26 int main () {
27     Rectangle rect, rectb;
28     rect.set_values (2,3);
29     rectb = duplicate (rect);
30     cout << rectb.area();
31 }
```

6.9.2 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.

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5  private:
6      int n;
7  public:
8      A(int);
9      friend class B; // Klasse als friend deklarieren
10 };
11
12 class B{
13     // Die Klasse B hat nun Zugriff auf die privaten
14     // Elemente der Klasse A
15 };
```

Beispiel:

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

```
1  #include <iostream>
2  using namespace std;
3
4  class Square;
5
6  class Rectangle{
7  private:
8      int width, height;
9  public:
10     int area () {return (width * height);}
11     void convert (Square a);
12 };
13
14 class Square {
15 private:
16     int side;
17 public:
18     void set_side (int a) {side=a;}
19     friend class Rectangle;
20 };
21
22 void Rectangle::convert(Square a) {
23     width = a.side;
24     height = a.side;
25 }
26
27 int main () {
28     Square sqr;
29     Rectangle rect;
30     sqr.set_side(4);
31     rect.convert(sqr);
32     cout << rect.area();
33     return 0;
34 }
```

6.10 Vererbung

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

Die folgende Abbildung zeigt eine Vererbungshirarchie.

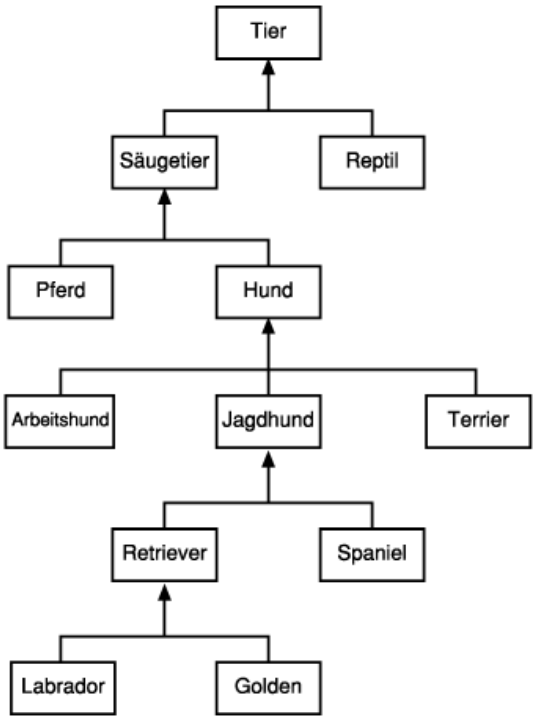


Abbildung 6.4: Vererbungshierarchie von Tieren

Hier kann in der Klasse `Tier` als Beispiel eine Methode `fressen()` im `public` Bereich deklariert werden. Diese Methode ist dann in allen Subklassen verfügbar.

Eine Vererbung wird folgendermassen deklariert:

```
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(){
    Man m;
    m.age = 12;  // Kann auf das Attribut age von Superklasse zugreifen
    m.displayAge();  // Kann auf die Methoden der Superklasse zugreifen

    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
```

6.10.1 Begriffe

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

6.10.2 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 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){
}
```

6.12 Klassendiagramme

Um Klassen graphisch darstellen zu können, existieren Klassendiagramme von UML (Unified Modelling Language).
Eine Klasse wird dabei einfach als Rechteck dargestellt und mit dem Klassenname beschriftet.
Es existieren drei verschiedene Detailgrade: Nur der Klassenname, Klassenname mit Attributen oder Klassenname mit Attributen und Methoden.

Die folgende Abbildung zeigt die drei möglichen Darstellungsarten.

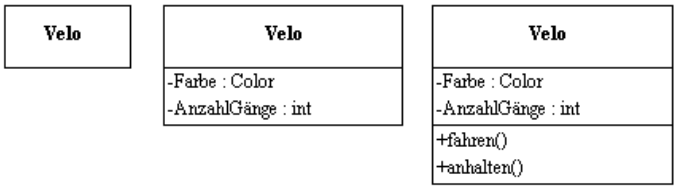


Abbildung 6.5: UML - Klassendiagramme

Eine Vererbung kann mit einem Pfeil dargstellt werden. Am Pfeilanfang befindet sich die Subklasse, am Pfeilende die Superklasse.
In der folgenden Abbildung sind die Klassen `Leiter` und `Kunde` von der Klasse `Person` abgeleitet.

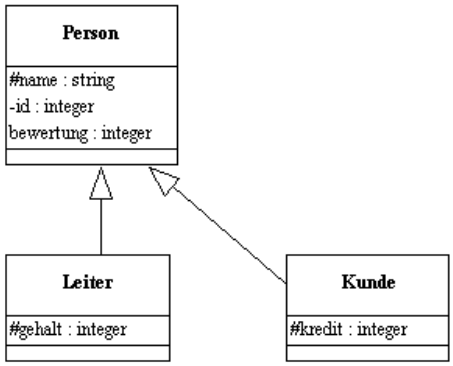


Abbildung 6.6: Darstellung der Vererbung in einem Klassendiagramm

Um die Zugriffsart darzustellen, kann jeweils vor die Methode oder das Attribut eines der folgenden Zeichen geschrieben werden:

- `-` `:private`
- `#` `:protected`
- `+` `:public`

6.13 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()`.

```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle{
5  protected:
6      int width, height;
7  public:
8      Rectangle(int, int);
9      int area();
10 };
11
12 Rectangle::Rectangle(int w, int h){
13     width = w;
14     height = h;
15 }
16
17 int Rectangle::area(){
18     return width * height;
19 }
20
21 class Square : public Rectangle{
22 public:
23     Square(int);
24     int area();
25 };
26
27 Square::Square(int side) : Rectangle(side, side){
28 }
29
30 int Square::area(){
31     return width * width;
32 }
```

6.14 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  #include <iostream>
2  using namespace std;
3
4  class A{
5  public:
6      void xy(){
7          cout << "xy from class A" << endl;
8      }
9  };
10
11 class B : public A{
12 public:
13     void xy(){
14         cout << "xy from class B" << endl;
15     }
16 };
```

Nun möchten Sie 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 dann die `xy` Methode der Superklasse aufgerufen.

6.15 Virtuelle Methoden

Betrachten Sie 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;
}
```

6.15.1 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.
Jetzt rufe ich auf diesem neu erzeugten Objekt die Methode `display()` auf.

```
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.

6.15.2 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:

```
1  #include <iostream>
2  using namespace std;
3
4  class A{
5  public:
6      virtual void display();  // Wird neu als virtuelle Methode deklariert
7  };
8
9  class B : public A{
10 public:
11     void display();
12 };
13
14 void A::display(){
15     cout << "A" << endl;
16 }
17
18 void B::display(){
19     cout << "B" << endl;
20 }
```

6.15.3 Virtuelle Destruktoren

Das gleiche wie für Methoden gilt auch für Destruktoren.

```
1  A *ptr = new B();
2  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.

6.16 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.

6.17 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.

6.17.1 Namenskonflikte

Betrachten Sie das folgende Beispiel:

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

class B{
protected:
    int n;
};

class C : public A, public B{
public:
    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 ich nun in der Klasse `C` auf das `n` zugreife, 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         A::n = 5;   // n von A
15         B::n = 5;   // n von B
16     }
17 };
```

6.17.2 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 {...};
```

Kapitel 7

Fortgeschrittene Themen

7.1 Templates

Oft kommt es vor, dass Funktionen mehrmals definiert werden müssen, da diese für verschiedene Datentypen verwenden werden können. Oder bei der Implementierung eines Stacks soll dieser sowohl für `char`, `int`, `double` oder andere Datentypen verwendet werden können.
Aus diesem Grund wurden in C++ Templates eingeführt.

7.1.1 Funkitonen Templates

Betrachten Sie 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` Variablen brauchbar. Nun soll diese Funktion so umgeschrieben werden, dass sie für beliebige Datentypen verwendet werden kann.

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

`GenericTyp` ist kein bestimmter Datentyp. An seiner Stelle kann ein beliebiger Datentyp verwendet werden. Wenn ich also diese Funktion aufrufe, muss bestimmt werden, was `GenericType` ist.

```
1 // Fuer GenericType wird double eingesetzt
2 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>
T GetMin (T a, U b) {
    return (a<b?a:b);
}
```

7.1.2 Klassen Templates

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

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

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

oder

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

Da es sich bei Template Klassen nur um Muster für die Erzeugung wirklicher Klassen handelt, muss für den Compiler ber der Übersetzung eines Programmteiles, das Templates 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

Eine Funktion endeckt einen Fehler, den sie selbst nicht behandeln kann, da die Funktion nur eine Aufgabe für eine höhere Instanz ausführt. Die Funktion wirft (`throw`) deshalb eine Exception (Ausnahme) und unterbricht die Abarbeitung der Aufgabe sofort.
Der Aufrufer kann die Funktion innerhalb eines `try` Blocks aufrufen, so dass er in der LAge ist eine von der Funktion geworfene Exception zu fangen (`catch` Block) und den Fehler zu behandeln.

7.2.1 Ausnahmen

C++ bietet das folgende Schema für Ausnahmebehandlung an:

```
1  try{
2      // critical code
3      throw exception;
4  }
5  catch (dataType1 dt1){
6      // Errorhandling
7  }
8  catch (dataType dt2){
9      // Errorhandling
10 }
11
12 // eventuell weitere catch-Bloecke
```

Betrachten Sie die folgende Funktion zur Division zweier `float` Werte:

```
1  float division(float a, float b){
2    return a/b;
3  }
```

Bei dieser Funktion tritt ein Fehler auf, falls `b` den Wert 0 hat (Division durch 0 ist nicht erlaubt).
Nun können wir diesen Fall abfangen:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  float division(float a, float b){
6
7      try{
8          if (b == 0) throw "Division durch 0";
9          return a/b;
10     }
11     catch (char *str){
12         cout << "Exception: " << str << endl;
13         return 0;
14     }
15 }
16
17 int main(){
18
19     cout << division(1,0) << endl;;
20     return 0;
21 }
```

7.2.2 Spezifikation

Man kann genau spezifizieren, welche Exceptions eine Funktion werfen kann. Damit wird das Programm in die Funktion `unexcepted()` geleitet, falls eine Ausnahme geworfen wird, die nicht der Spezifikation entspricht. Dadurch geschieht in der Regel ein sofortiger Programmabbruch.

```
1  void f1() throw(int); // darf nur Exceptions vom Typ int werfen
2  void f2() throw(); // darf keine Exceptions werfen
```

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

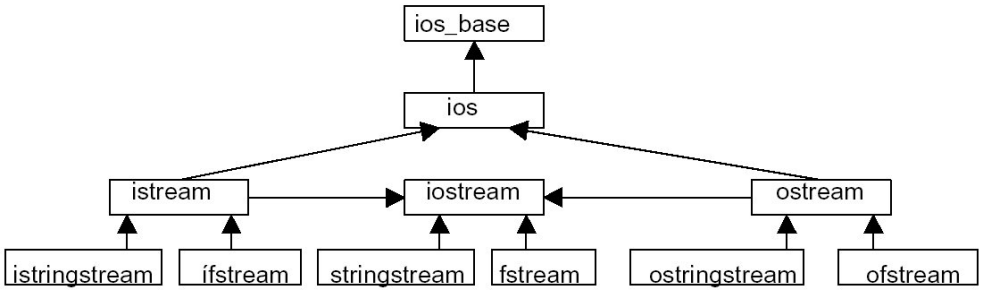


Abbildung 7.1: Klassen für die Ein- und Ausgabe mit Files

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.

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

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

```
1 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.

```
1  #include <string>
2  #include <fstream>
3  using namespace std;
4
5  int main(){
6
7      ifstream file;
8      file.open("xy.txt");
9
10     if (!file.is_open()) exit(1);
11
12     string line;
13
14     while (!file.eof()){
15         file >> line;
16         cout << line << endl;
17     }
18
19     file.close();
20
21     return 0;
22 }
```

Beispiel:

Implementierung des Unix Programmes `cat`.

```
1  #include <fstream>
2  using namespace std;
3
4  int main(int argc, char **argv){
5
6      ifstream file(argv[1]);
7
8      if (!file.is_open()){
9          cout << "File not found!" << endl;
10         exit(1);
11     }
12
13     char ch;
14     while (file.get(ch)){
15         cout << ch;
16     }
17
18     file.close();
19
20     return 0;
21 }
```

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.

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

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

```
1 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.

```
1 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.
Ein Unterprogramm 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 des Unterprogrammes ruft sich dieses 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 }
```

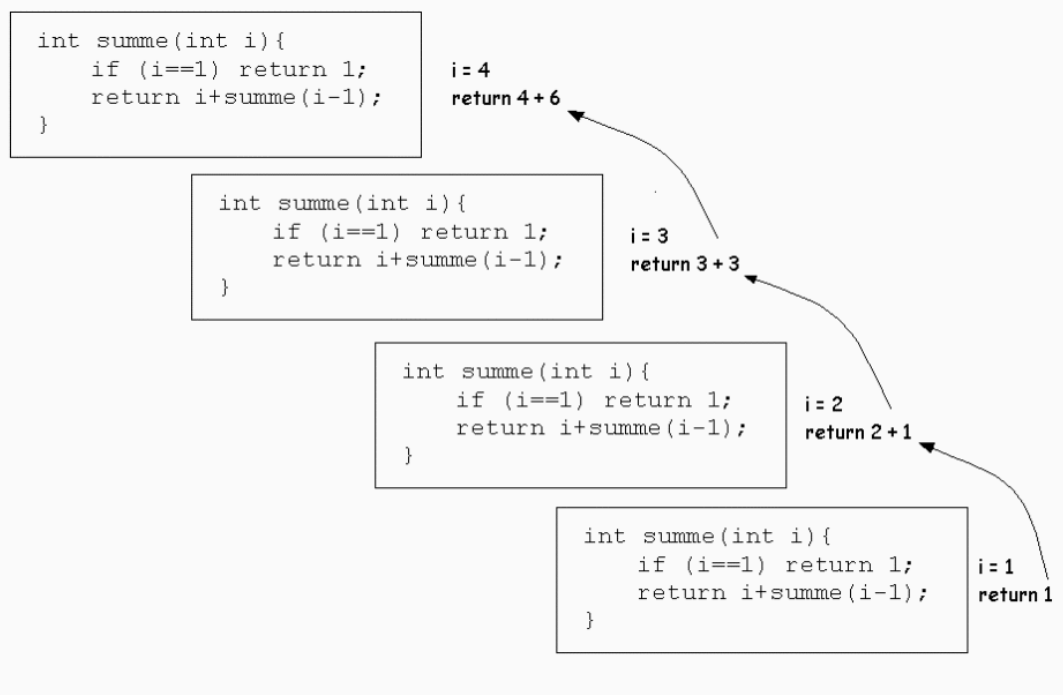


Abbildung 8.1: Rekursiver Aufruf einer Funktion

Kapitel 9

Dynamische Datenstrukturen

Ein Geiger zerreisst viele Saiten
ehe er Meister ist.
Sprichwort

9.1 Verkettete Liste

Eine verkettete Liste ist eine dynamische Datenstruktur, d.h. alle Datenelemente werden während dem Programmablauf dynamisch erzeugt. Jedes Datenelement enthält neben den eigentlichen Daten einen Pointer auf ein weiteres Element.

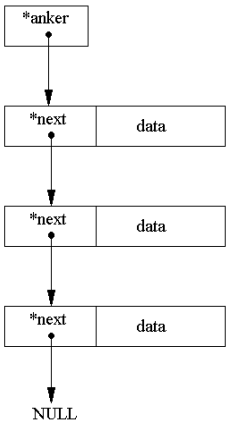


Abbildung 9.1: Verkettete Liste

In einer verketteten Liste können Elemente nach Bedarf eingefügt oder entfernt werden. Hingegen kann (im Gegensatz zum Array) nicht direkt mit einem Index auf das i-te Element zugegriffen werden.

Beim Einfügen und Entfernen von Elementen müssen lediglich die Pointers der entsprechenden Elemente neu gesetzt werden.

9.1.1 Einfügen

Ein Element kann in die verkettete Liste an einer beliebigen Stelle eingefügt werden. Dabei muss der `next` Pointer des vorherigen Elementes auf das neue Element zeigen und der `next` Pointer des neuen Elementes muss auf das Element zeigen, auf das das vorherige gezeigt hat.

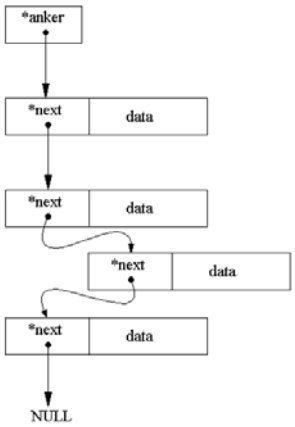


Abbildung 9.2: Einfügen eines Elementes in eine verkettete Liste

9.1.2 Entfernen

Beim Entfernen muss der `next` Pointer des Elementes, welches vor dem Elemente steht, welches Entfernt werden soll, auf das Element zeigen, auf welches der `next` Pointer des zu entfernenden Elementes zeigt.

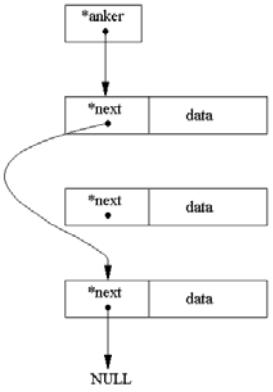


Abbildung 9.3: Enternen eines Elementes aus einer verketteten Liste

Ein Element einer verketteten Liste kann wiefolgt aussehen:

```
1 class Element {
2
3   public:
4     ElementType data;
5     Element *nextAdress;
6
7 };
```

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

9.2 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.

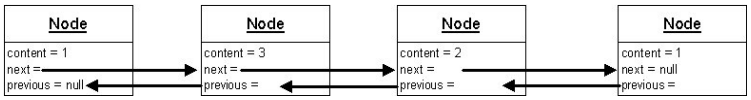


Abbildung 9.4: Doppelt verkettete Liste

Ein Element einer doppelt verketteten Liste kann wiefolgt aussehen:

```
1 class Element {
2
3   public:
4     ElementType data;
5     Element *nextAdress;
6     Element *prevAdress;
7
8 };
```

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

9.3 Stacks

Die Bezeichnung Stack (Stapel) 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`.

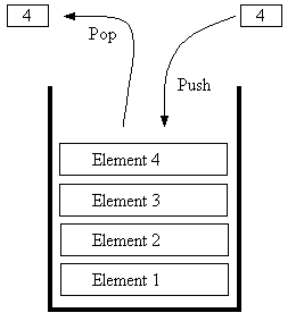


Abbildung 9.5: Stack

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

Weiter sollten auf einem Stack die folgenden Operationen ausgeführt werden können:

- `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.
- `int size()`
  Gibt die aktuelle Grösse des Stacks zurück.

9.3.1 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.

```
1  template <class T>
2  class Stack{
3          int actual, max;
4          T *elements;
5  public:
6          Stack(int);
7          ~Stack();
8          bool push(T);
9          bool pop(T&);
10 };
11
12 template <class T>
13 Stack<T>::Stack(int max){
14         actual = 0;
15         this->max = max;
16         elements = new T[max];
17 }
18
19 template <class T>
20 Stack<T>::~Stack(){
21         delete [] elements;
22 }
23
24 template <class T>
25 bool Stack<T>::push(T element){
26         if (actual >= max) return false;
27         elements[actual] = element;
28         actual++;
29         return true;
30 }
31
32 template <class T>
33 bool Stack<T>::pop(T &element){
34         if (actual <= 0) return false;
35         actual--;
36         element = elements[actual];
37         return true;
38 }
```

9.4 Binäre Bäume

Jedes Element erhält zwei Pointers left und right:

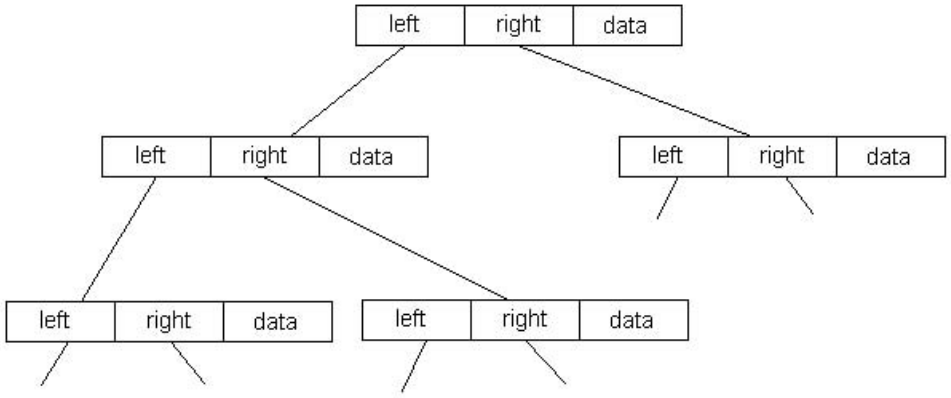


Abbildung 9.6: Binärer Baum

9.4.1 Rekursive Definition eines binären Baumes

Ein binärer Baum ist entweder eine leere Datenstruktur oder eine Wurzel mit einem linken und einem rechten Teilbaum, welche beide wieder binäre Bäume sind. Die Rekursion wird im nächsten Kapitel behandelt.

9.4.2 Definition eines sortierten Baumes (Search Tree)

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 $R.key > N.key$.

9.4.3 Implementierung

Bei einem binären Baum kann ein Element folgendermassen implementiert werden:

```
1 class Node {
2
3   public:
4     ElementType data;
5     Node *nextAddressLeft;
6     Node *nextAddressRight;
7
8 };
```

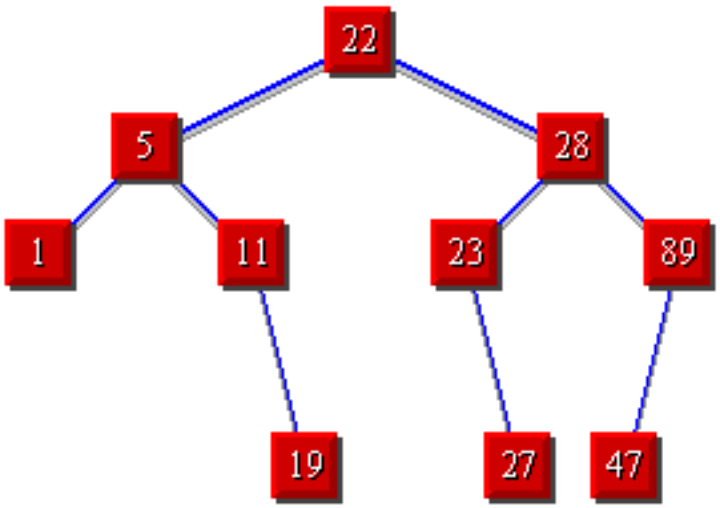
9.4.4 Traversierung

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



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

9.4.5 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 $log_2 N$.

Kapitel 10

Sortieren

Wissen ist Macht.
F.Bacon

Die im folgenden Kapitel gezeigten Algorithmen haben alle das gleiche Ziel: Die Reihenfolge der Elemente soll so geändert werden, dass sie nachher in auf- oder absteigender Reihenfolge sortiert sind.

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.

Es gibt viele verschiedene Sortieralgorithmen. Die in diesem Kapitel vorgestellten Algorithmen sind *einfache Algorithmen*. Sie erreichen alle das geforderte Ziel und arbeiten nach verschiedenen Methoden.

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

10.1 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
```

10.1.1 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.

10.1.2 Source Code

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

```
1  void swap(int &a,int &b){
2     int tmp = a;
3     a = b;;
4     b =tmp;
5  }
6
7  void bubbleSort(int *array,int size){
8
9     for(int obergrenze=size-1; obergrenze>0; obergrenze--){
10       for(int pos=0; pos<obergrenze; pos++){
11          if(array[pos] > array[pos+1])
12             swap(array[pos], array[pos+1]);
13       }
14    }
15 }
```

10.2 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
```

10.2.1 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.

10.2.2 Source Code

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

```
1  void selectionSort(int *array, int size){
2      int small, temp;
3      for (int i=size-1; i>0; i--){
4          small = 0;
5          for (int j=1; j<=i; j++){
6              if (array[j] > array[small]) small = j;
7          }
8          temp = array[small];
9          array[small] = array[i];
10         array[i] = temp;
11         }
12     return;
13 }
```

10.3 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
```

10.3.1 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.

10.3.2 Source Code

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

```
1  void insertionSort(int *array, int size){
2      int key;
3      for(int j=1; j<size; j++){
4          key = array[j];
5          for(int i=j-1; (i>=0) && (array[i]>key); i--){
6              array[i+1] = array[i];
7                  }
8          array[i+1] = key;
9      }
10     return;
11 }
```

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

Abbildungsverzeichnis