

Skript zum Vorkurs Informatik

23. September 2010

Inhaltsverzeichnis

1. Einleitung	4
2. Grundlagen	5
2.1. Hello-World Programm	5
2.2. Rechenoperationen	6
2.3. Kommentare	7
2.4. Variablen	8
2.5. Eingaben	9
2.6. Entscheidungen	10
2.7. Schleifen	11
2.7.1. for-Schleifen	11
2.7.2. do-while-Schleifen	12
2.8. Funktionen	13
2.9. Felder (Arrays)	15
2.9.1. Arrays fester Größe	15
2.9.2. Arrays variabler Größe	16
3. Rekursion	16
3.1. Fibonacci-Folge	16
3.2. Fakultätsfunktion	18
4. Bibliotheken	19
4.1. Math-Bibliothek	19
4.2. Strings	20
5. Sortierproblem	22
5.1. Definition des Sortierproblems	22
5.2. Bubble-Sort Algorithmus	22
6. Zeiger	24
6.1. Call-by-value und call-by-reference	24
6.2. Zeiger und Felder	26
6.3. Dynamische Speicherverwaltung	26
6.4. Einfach verkettete Listen	27
7. Übungsaufgaben	31
7.1. Primzahltest	31
7.2. Übung zu Feldern	32
7.3. Größter gemeinsamer Teiler	33
7.4. Euklidischer Algorithmus	34

Inhaltsverzeichnis

A. Kurze Einführung in Linux	35
A.1. Wichtige Befehle in der Konsole	35
A.2. Wichtige Programme	35
B. C++ unter Windows	35
B.1. Installation	35
B.2. Deinstallation	36
B.3. Erstellen eines Konsolenprogramms	36
B.4. Mehrere Projektdateien mit Dev-CPP	37
C. Makefiles	37
D. Einige nützliche Links	39

1. Einleitung

1. Einleitung

Dieses Skript ist im Oktober 2007 begleitend zum Vorkurs Informatik der Fachschaft *MathPhys* entstanden. Es soll Studierenden den Umgang mit Linuxsystemen und eine kleine Einführung in die Programmierung mit *C++* geben. Das Skript enthält die Lösungen zu den im Kurs bearbeiteten Übungsaufgaben. Diese Aufgaben können einerseits wie im Skript erklärt durch Verwendung von *C++* unter Linux, als auch mit der freien Entwicklungsumgebung *Dev-CPP* (siehe Anhang A) nachvollzogen werden. Korrekturen und Verbesserungsvorschläge dürfen gerne an daniel.jungblut@iwr.uni-heidelberg.de geschickt werden.

Verändert im September 2008 durch Daniel Gerecht

Verändert im September 2010 durch Max Menges

2. Grundlagen

2.1. Hello-World Programm

Aus dem Startmenü wird zunächst das *Microsoft Visual Studio 2008* gestartet. Mit einem Klick auf `Datei` → `Neu` → `Projekt` ein „Neues Projekt“ angelegt. Um nun auch in *C++* programmieren zu können, muss noch eine neue *C++ Datei* (.cpp) hinzugefügt werden, wieder mit einem Klick auf `Projekt` → `neues Element hinzufügen`.

```
#include <iostream>

using namespace std;

int main() {

    cout << "Hallo Welt!" << endl;

    return 0;
}
```

Dieses in *C++* geschriebene Programm muss nun zunächst vom dem Computer in Maschinensprache übersetzt werden. Diesen Vorgang bezeichnet man als *kompilieren*. Anschließend kann das Programm von dem Computer ausgeführt werden. Das *Visual Studio* führt beides gleichzeitig durch Klicken auf *Debuggen*, oder durch Drücken der *F5* Taste. Da die *Konsole*, die für die Ausgabe vom *Visual Studio* geöffnet wird, wieder geschlossen wird sobald das Programm durchgelaufen ist, fügen wir *vor* dem Befehl `return 0;` noch die Zeile `system("PAUSE");` ein. Dies führt dazu, dass das Programm solange angehalten wird, bis man `<enter>` drückt.¹ Wir bekommen in der Konsole die Ausgabe:

```
Hallo Welt!
```

Anhand des Hello-World Programms können wir uns nun den grundlegenden Aufbau eines *C++* Programms anschauen:

Die Zeile `#include <iostream>` sagt dem Compiler, dass er den Inhalt der Datei *iostream* in das Programm einfügen soll. Diese wird immer benötigt, wenn das Programm Ein- oder Ausgaben machen soll.

¹Da dieses Skript für das Programmieren in einer Linux Konsole ausgelegt war, fehlt der `system("PAUSE")` in allen nachfolgenden Programmen, da dieser unter Linux nicht notwendig ist.

2. Grundlagen

Die Zeile `using namespace std;` sagt dem Programm, dass wenn keine weiteren Angaben gemacht werden, der Namensraum *std* (Standard) genutzt werden soll. Mit verschiedenen *Namespaces* lassen sich z.B. Variablen mit dem gleichen Namen in einem Programm verwenden, was normalerweise zu Konflikten führen würde. Dies ist aber erst einmal für unsere einfachen Anwendungen nicht nötig.

Wir kommen nun zum wichtigsten Teil eines C++ Programms, der *main* Funktion. Jedes C++ Programm muss genau *eine main* Funktion enthalten. Diese wird später als erstes vom Programm aufgerufen. Die *main* Funktion ist wie folgt aufgebaut:

Wir sagen zuerst, dass die Funktion vom Typ `int` sein soll (s. Kap. 3.4) und geben ihr dann einen Namen (hier: `main`). In der Paranthese (also zwischen den runden Klammern) definieren wir Parameter die der Funktion übergeben werden müssen wenn sie aufgerufen wird. Die *main* Funktion braucht keine Parameter, die Klammern müssen trotzdem vorhanden sein.

Der wichtige Teil des Programms befindet sich zwischen den beiden geschweiften Klammern. Hier können wir Befehle eingeben, die beim Ausführen des Programms abgearbeitet werden. Befehle werden durch ein Semikolon (;) von einander getrennt. Den ersten Befehl den wir hierbei gerade gelernt haben ist der Befehl `cout`. Mit Hilfe von `cout` können wir Text oder Ergebnisse unseres Programms in der Konsole als Text ausgeben lassen. Hierbei können wir durch das Zeichen `<<` verknüpft mehrere Ausgaben auf einmal erledigen. Beispiel:

```
cout << "Hallo" << endl << "Welt!" << endl;
```

Der Befehl `endl` erzeugt hierbei jeweils eine neue Zeile. Wenn wir Text ausgeben möchten, so ist dieser immer in Anführungszeichen zu schreiben.

Eine Funktion wird immer mit dem Befehl `return` beendet. Der Wert `0` (Null) hier sagt, dass das Programm normal ausgeführt wurde und keine Fehler aufgetreten sind.

2.2. Rechenoperationen

Unser Computer kann aber nicht nur Text ausgeben, sondern auch rechnen. Wir ändern unser obiges Programm wie folgt:

```
#include <iostream>

using namespace std;
```

2. Grundlagen

```
int main() {  
  
    cout << "3+5" << endl;  
    cout << 3+5 << endl;  
  
    return 0;  
}
```

Wir kompilieren unser Programm wie oben beschrieben und führen es aus. Die Ausgabe lautet:

```
3+5  
8
```

Wie eben gelernt, wird durch Anführungszeichen immer ein Text dargestellt. `3+5` ohne Anführungszeichen symbolisiert allerdings eine Rechnung. Das Ergebnis dieser einfachen Rechnung ist offensichtlich 8, was auch ausgegeben wird. Weitere Rechenoperatoren sind:

+ (Addition)

- (Subtraktion)

* (Multiplikation)

/ (Division)

% (Modulo)

2.3. Kommentare

Wir können in unseren Programmtext beliebigen Text einbauen, wenn wir am Anfang der entsprechenden Zeile das Kommentarzeichen `//` einfügen oder einen Kommentarabschnitt mit `/*` beginnen und mit `*/` beenden.

```
// Mit Kommentaren kann man sinnvolle  
// Erklärungen in das Programm einbauen.  
  
// Zudem kann man Programmzeilen, die  
// man gerade nicht benötigt auskommentieren:  
// cout << 3+5 << endl;  
  
/*
```

2. Grundlagen

```
So  
kann  
man  
ein  
Kommentar  
über  
mehrere  
Zeilen  
ausdehnen.  
*/
```

2.4. Variablen

Es ist notwendig in Programmen mit Variablen zu rechnen. Diese Variablen sind vergleichbar mit Variablen die aus der Mathematik bekannt sind. Wir müssen dem Programm nur genau mitteilen, welchen *Typ* welche Variable haben soll. Solche Typen sind beispielsweise:

int: Ganze Zahl (1 | -2 | 3 ...)

float: Fließkommazahl (1,23 | -2,43)

char: Zeichen (a | b | @)

string: Buchstabenkette (benötigt <string>)

bool: Wahrheitswert. Kann die Werte **true** oder **false** annehmen.

Um eine Variable in einem Programm zu nutzen, muss sie und ihr Typ durch folgenden Befehl deklariert, d.h. festgelegt werden:

```
int x = 5;
```

Zunächst müssen wir den Typ angeben, den unsere Variable haben soll (hier: **int**). Anschließend wird der Name der Variablen festgelegt (hier: x). Nun können wir der Variable mit einem = getrennt einen Wert zuweisen. Es ist nicht unbedingt nötig einer Variablen einen Wert zuzuweisen, es ist aber üblich dies zu tun. Weitere Beispiele:

```
// Beispiel wie eben:  
int x = 5;  
// Deklarieren mehrerer Variablen auf einmal:  
int i, j, k;  
// Deklaration einer Fließkommazahl:
```

2. Grundlagen

```
float f = 2.5;
// Buchstaben werden im Gegensatz zu Text durch einzelne
// Anführungszeichen ' angegeben:
char c = 'e';
// Ein boolscher Wert:
bool b = true;
```

Da durch den Operator = den Variablen Werte zugewiesen werden, wird = in diesem Zusammenhang als *Zuweisungsoperator* bezeichnet. Wir können die Werte einer Variablen auf dem Bildschirm ausgeben lassen:

```
int x = 5;
cout << x << endl;
```

Befehle werden in der Reihenfolge abgearbeitet, in der sie notiert sind. Um eine Variable verwenden zu können, muss sie also weiter oben deklariert worden sein.

2.5. Eingaben

Bisher mussten alle Variablen oder Rechnungen die unsere Programme ausführen sollen zur *Compilerzeit* (d.h. dem Zeitpunkt zu dem das Programm Compiliert wird) bekannt sein. Mithilfe des `cin` Befehls können wir nun Eingaben zur Laufzeit des Programmes machen.

Wir können z.B. ein Programm schreiben, das den Benutzer auffordert zwei Zahlen einzugeben und diese dann addiert:

```
#include <iostream>

using namespace std;

int main(){
    int a,b;
    cout<<"Erste Zahl eingeben: ";
    cin>>a;
    cout<<"Zweite Zahl eingeben: ";
    cin>>b;
    cout<<a<<"+"<<b<<"="<<a+b<<endl;
    return 0;
}
```

Die Ausgabe könnte z.B. so aussehen:

2. Grundlagen

```
Erste Zahl eingeben: 40
Zweite Zahl eingeben: 2
40+2=42
```

Wir sehen, dass `cout` nicht unbedingt ein `endl` benötigt, so kann man die Eingabe (z.B. 40 und 2) in der gleichen Zeile erfolgen lassen. Da die Eingabe mit der Enter-Taste bestätigt wird, findet trotzdem ein Zeilenumbruch statt.

Der Befehl `cin>>a;` ist so zu interpretieren, dass die Eingabe in `a` gespeichert werden soll. Auch so kann man Variablen Werte zuweisen. Man muss allerdings darauf achten, dass die Eingabe auch mit dem Typ der Variablen übereinstimmt. Die Pfeile (`>>`) zeigen beim `cin` in die andere Richtung als beim `cout`.

2.6. Entscheidungen

Wir möchten unser Programm Entscheidungen der Form *wenn A, dann B, sonst C* ausführen lassen. Hierfür verwenden wir ein *if-else-Statement*:

```
int main(){
    int x;
    cout<<"Zahl Eingeben: ";
    cin>>x;
    if (x%2==0) cout<<x<<"ist gerade"<<endl;
    else cout<<x<<"ist ungerade"<<endl;
    return 0;
}
```

Nach dem Befehl `if` folgt in runden Klammern die Bedingung nach der entschieden werden soll. Ist diese wahr wird der danach folgende Befehl ausgeführt. Ist die Bedingung falsch, so wird der Befehl nach dem `else` ausgeführt. Die Alternative `else` ist nicht obligatorisch. Lässt man sie weg, so passiert einfach nichts, wenn die Bedingung falsch ist. Zu beachten ist, dass bei einer Bedingung, die eine Gleichheit beschreibt, der zu verwendende Operator aus zwei Gleichheitszeichen besteht. Weitere dieser *Vergleichsoperatoren* sind:

`==` gleich

`!=` ungleich

`>` größer als

`<` kleiner als

2. Grundlagen

$>=$ größer als oder gleich

$<=$ kleiner als oder gleich

Mit den sogenannten *logischen Operatoren*

$\&\&$ und

$\|\|$ oder

$!$ nicht

können Bedingungen auch verknüpft werden. Beispiel:

```
if(((x < 5) && (y != 2)) || (x < 10))
```

Bei der Konstruktion solcher Ausdrücke ist auf korrekte Klammersetzung zu achten. Möchte man mehrere Befehle bedingt ausführen lassen, so kann man nach dem *if-Statement* einen Befehlsblock durch geschweifte Klammern abgrenzen. Beispiel:

```
if(x == 5) {  
    // hier können nun beliebige Befehle stehen  
} else {  
    // so ein Befehlsblock ist natürlich auch nach  
    // else möglich  
}
```

Es empfiehlt sich, auch bei nur einem Befehl die Klammern $\{ \}$ zu setzen. Dies verhindert später Fehler wenn man weitere Befehle einfügen will und dient der Übersichtlichkeit.

2.7. Schleifen

2.7.1. for-Schleifen

Um den Computer einzelne Befehle oder eine Abfolge von Befehlen wiederholt ausführen zu lassen bedienen wir uns einer *Zählschleife*, auch *for-Schleife* genannt.

```
for(int i = 0; i < 10; i++) {  
    cout << i << endl;  
}
```

Die Ausgabe dieses Programmes ist:

2. Grundlagen

```
0
1
2
3
4
5
6
7
8
9
```

Offenbar zählt das Programm von 0 bis 9 und gibt diese Zahlen aus. Die *for-Schleife* besteht aus drei Befehlen die durch Semikolons getrennt und in runde Klammern hinter dem Schlüsselwort `for` geschrieben werden. Anschließend wird durch geschweifte Klammern ein Befehlsblock definiert. Durch die *for-Schleife* werden nun alle Befehle innerhalb dieses Befehlsblock wiederholt ausgeführt.

1. Im ersten dieser drei Befehle wird eine Zählvariable (normalerweise vom Typ `int`) deklariert und initialisiert: `int i = 0;`
2. Nun folgt die Bedingung, wie lange gezählt werden soll. Hierbei wird die *for-Schleife* wiederholt solange diese Bedingung erfüllt ist: `i < 10;`
3. Zuletzt wird definiert, in welchen Schritten die Zählvariable verändert werden soll: `i++`

`i++` bedeutet hierbei, dass die Zählvariable `i` in Einerschritten hochgezählt wird. Durch `i = i + n` kann man durch Festlegen von `n` auch andere Schrittgrößen einstellen. Der Befehl `i+=n` bewirkt das selbe, spart aber bei längeren Variablennamen ein wenig Schreibarbeit.

Ist bereits zur Compilerzeit bekannt wie oft die Schleife durchlaufen werden soll eignet sich die *for-Schleife*. Oftmals ist aber zur Compilerzeit nicht bekannt wie oft die Schleife durchlaufen werden soll (z.B. wenn auf eine bestimmte Eingabe gewartet wird).

2.7.2. do-while-Schleifen

Für den Fall gibt es die *while-* und *do-while-*Schleifen. Die *while-Schleife* wird solange ausgeführt, wie eine bestimmte Bedingung erfüllt ist.

```
bool a=true;
int x;
```

2. Grundlagen

```
while (a==true){
    cout<<"Zahl eingeben: "<<endl;
    cin>>x;
    if (x==0){
        a=false;
    }
}
```

Wie man sieht fordert uns das Programm auf, solange eine Zahl einzugeben, bis wir 0 eingeben. Dies wäre mit einer *for*-Schleife nur schwer machbar gewesen. Man kann allerdings jede *for*-Schleife durch eine *while*-Schleife ersetzen.

Bei *while*-Schleifen muss man allerdings darauf achten, dass die Abbruchbedingung auch erreicht werden kann. Dass dies nicht unbedingt immer der Fall ist, zeigt das folgende Beispiel:

```
int i=1;
while (i!=2){
    i=i+2;
}
```

Da die Bedingung *vor* der Ausführung des { }-Blocks überprüft wird, kann es passieren, dass die Anweisung gar nicht ausgeführt wird. Will man die Anweisung aber mindestens einmal ausführen verwendet man die *do-while*-Schleife. Diese überprüft die Bedingung erst nach der Ausführung:

```
do {
    cout<<"Diese Anweisung würde in einer while-Schleife
        nicht ausgeführt werden."<<endl;
} while (1==2);
```

2.8. Funktionen

Bei größeren Programmen wird es recht schnell unübersichtlich, wenn man Befehl für Befehl hintereinander schreibt. Eine Lösung hierfür bieten Funktionen. Möchte man in einem Programm eine Berechnung mehrfach, aber an unterschiedlichen Stellen durchführen, bietet es sich an, eine Funktion für diese Berechnung zu entwerfen. Die entsprechende Funktion kann an jeder Stelle des Programms aufgerufen werden und macht somit den zu schreibenden Quelltext kürzer und wesentlich übersichtlicher. Wir betrachten hierzu folgendes Programm, das eine Funktion zur Berechnung der Quadrate von ganzen Zahlen enthält:

```
#include <iostream>
```

2. Grundlagen

```
using namespace std;

int quadrat(int n) {
    return n * n;
}

int main() {
    cout << quadrat(3) << endl;
    int x = 5;
    int y = quadrat(x);
    cout << y << endl;

    return 0;
}
```

Bestandteile einer Funktion:

1. Funktionskopf:

int: Gibt den Variablentyp des Rückgabewertes der Funktion an.

quadrat: Name der Funktion.

(int n): Eingabeparameter der Funktion.

Als Typ des Rückgabewertes können wir jeden Variablentyp verwenden, den wir bisher kennengelernt haben (*int*, *float*, *char*, *bool* usw.). Ebenso sind bei der Namensgebung der Funktionen kaum Grenzen gesetzt. Einzig Umlaute, die meisten Sonderzeichen und Wörter, die bereits in *C++* eine andere Bedeutung haben, dürfen hier nicht verwendet werden. Die Anzahl der Parameter ist völlig frei. Hier ein Beispiel mit vier Parametern:

```
int TestFunktion(int n, char c, float f, bool b)
```

2. Funktionsrumpf:

Dies ist der Teil der Funktion, der zwischen den geschweiften Klammern steht, also die eigentlichen Befehle, die beim Funktionsaufruf ausgeführt werden. Hier dürfen wir nach Herzenslust Variablen deklarieren und benutzen. Die als Parameter übergebenen Variablen dürfen (und sollten, sonst bräuchte man sie ja nicht zu übergeben) natürlich auch verwendet werden. Wenn die Funktion ihr Ziel erfüllt hat, können wir mit dem *return* Befehl wieder aus der Funktion herausspringen. Durch den Parameter nach dem *return* Befehl wird das Ergebnis der Funktion angegeben.

2. Grundlagen

Der Aufruf der Funktion im Hauptprogramm sollte durch das obige Beispiel klar sein. Wir haben bereits (in Kap. 3.1) gelernt, dass jedes C++ Programm genau *eine main-Funktion* enthält. Diese wird auch *Hauptprogramm* genannt und dient als Startpunkt wenn das Programm ausgeführt wird. In der *main-Funktion* (und natürlich auch in anderen Funktionen) können nun unsere selbst geschriebenen Funktionen aufgerufen werden, die dann der Reihe nach abgearbeitet werden.

Dabei muss nicht für jede Funktion selbst geschrieben werden. C++ stellt in sog. *Bibliotheken* (z.B. `<iostream>`, `<math>`, `<string>`) Funktionen für verschiedene Anwendungen bereit. Einige davon werden wir später noch kennen lernen.

2.9. Felder (Arrays)

Oft benötigt man in einem Programm mehrere zusammengehörende Variablen oder ganze Gruppen von Variablen, von denen man nicht weiß, wie viele benötigt werden. Hierfür gibt es sogenannte *Arrays* oder *Felder*. Dabei gibt es nur einen Variablennamen, die einzelnen Variablen werden einfach durchnummeriert:

```
a[0], a[1], a[2] ...
```

2.9.1. Arrays fester Größe

Man deklariert etwa:

```
int i[17];
```

Nun legt der Computer gleich 17 Ganzzahlwerte auf einmal an. So können wir auf die einzelnen Werte des Feldes zugreifen:

```
i[11] = 4;
i[7] = 10;
i[3] = i[11] + i[7];
cout << i[3] << endl;
```

Dies liefert z. B. die Ausgabe 14.

Die Zahl in den Klammern wird auch Index genannt. Zu beachten ist noch, dass der erste Wert des Arrays mit dem Index [0] zu erreichen ist, der höchste mit dem Index [n-1] (hier [16]).

3. Rekursion

2.9.2. Arrays variabler Größe

Es können auch Felder mit variabler Größe angelegt werden:

```
int n = 12;
float x[n];
```

Selbstverständlich ist es auch möglich, Felder von einem anderem Variablentyp (welche kennst du bis jetzt?) zu erzeugen.

Die große Stärke von Feldern besteht darin, dass man auf die einzelnen Einträge auch mit Variablen zugreifen kann. Hier ein kleines Beispiel:

```
int q[26];

for (int n = 1; n <= 25; n++) {
    q[n] = n * n;
}
```

Jetzt hätte z. B. `q[12]` den Wert 144.

Achtung: Man kann bei solchen Zugriffen mit Variablen auch auf Felder zugreifen, die vorher *nicht* deklariert wurden. Gibt man diese mit `cout` aus, erhält man meistens nur zufällige Zahlen. Man kann allerdings auch nicht deklarierte Speicheradressen überschreiben, was dann passiert kann man nur raten.²

3. Rekursion

„To understand recursion one must first understand recursion“

Rekursion ist eine nützliche Programmiertechnik, mit der man einfache Probleme sehr elegant und mit wenig Programmieraufwand lösen kann. Unter einer rekursiven Funktion versteht man eine Funktion, die sich selbst aufruft. Wir wollen dies an zwei Beispielen veranschaulichen:

3.1. Fibonacci-Folge

Ein klassisches Beispiel ist die sog. Fibonacci-Folge:

1, 1, 2, 3, 5, 8, 13, 21, ...

²cf. Wiki über Pufferüberlauf

3. Rekursion

Die nachfolgende Zahl wird berechnet, indem man ihre beiden Vorgänger addiert. Die ersten beiden *Fibonacci-Zahlen* sind hierbei mit dem Wert 1 festgelegt. Wir wollen zunächst die Fibonacci-Zahlen *iterativ* berechnen:

```
int fib (int n){

    int fib1=1, fib2=1, erg;
    // Wir definieren die ersten beiden Fib. Zahlen und
    // ein "Ergebnis"

    if (n==1 || n==2)

        return 1;
        // Nach Definition sind die ersten beiden
        // Zahlen 1

    else {

        for (int i=3; i<=n; i++){
            // Da die ersten beiden Zahlen 1 sind fängt
            // unsere schleife bei 3 an

            erg=fib1+fib2;
            // Das Ergebnis wird aus den letzten
            // beiden Fibonacci-Zahlen berechnet

            fib1=fib2;

            fib2=erg;
            // Die letzten beiden Fib.-Zahlen werden
            // ein "weitergerückt".
        }

    }

    return erg;
    // Die n.-Fibonacci-Zahl wird zurückgegeben
}
```

Wie man sieht ist die *iterative* Programmierung eher unübersichtlich und nichtintuitiv. Wir wollen das gleiche Programm nun *rekursiv* formulieren.

Dafür definieren wir zunächst allgemein die Fibonacci Folge rekursiv:

$$\text{fib}(n) = \begin{cases} 1, & \text{falls } n \in \{1, 2\} \\ \text{fib}(n-2) + \text{fib}(n-1), & \text{sonst} \end{cases}$$

3. Rekursion

Wir wollen nun die fib-Funktion von oben umschreiben:

```
int fib(int n) {  
    if((n == 1) || (n == 2))  
        return 1;  
    else  
        return fib(n-2) + fib(n-1);  
}
```

Man sieht, dass die Funktion nun deutlich kürzer und übersichtlicher geworden ist.

Schauen wir uns einmal an, was der Computer rechnet, wenn wir ihn $fib(5)$ ausrechnen lassen:

```
fib(5)=fib(4)+fib(3)  
=(fib(3)+fib(2))+fib(2)+fib(1)  
=((fib(2)+fib(1))+1)+(1+1)  
=((1+1)+1)+(2)  
=(2+1)+2=5
```

Man spricht hierbei von *kaskadierender* Rekursion, da jeder Funktionsaufruf zwei neue Funktionen aufruft. Dieser ist zwar einfacher zu programmieren, jedoch nicht unbedingt schneller als die *iterative* Programmierung, da z.B. $fib(3)$ mehrmals aufgerufen wird. Bei der *linearen* Rekursion, ein Beispiel ist die Fakultätsfunktion im nächsten Abschnitt, tritt dieses Problem nicht auf.

3.2. Fakultätsfunktion

Die Aufgabe ist es eine Funktion zu schreiben, die die Fakultät einer ganzen Zahl (genauer: einer natürlichen Zahl) ausrechnet. Mathematisch kann die Fakultätsfunktion auf zwei Arten definiert werden:

1.

$$n! = \begin{cases} 1, & \text{falls } n = 1 \\ n(n-1)!, & \text{sonst} \end{cases}$$

2.

$$n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

4. Bibliotheken

Hierbei enthält wieder die erste Variante die Definition der Funktion die zu definieren ist, die Funktion selbst. Die *iterative* Definition ist das Produkt der natürlichen Zahlen von 1 bis n. Wir können nun beide Varianten programmieren:

```
//rekursiv
int fakrec(int n) {
    if(n == 1) return 1;
    else return n * fakrec(n-1);
}

//iterativ
int fakit(int n) {
    int ergebnis = 1;

    for(int i = 1; i <= n; i++)
        ergebnis = ergebnis * i;

    return ergebnis;
}
```

4. Bibliotheken

Wir wollen uns nun die bereits in früheren Kapiteln angesprochenen *C++-Bibliotheken* anschauen. Bisher haben wir nur die *iostream*-Bibliothek verwendet. Die *C++*-Bibliotheken stellen uns jedoch eine Reihe nützlicher Funktionen zur Verfügung. Die Bibliotheken werden wie folgt eingebunden:

```
# include <name>
```

Das `#` ist kein *C++-Code* sondern eine Anweisung für den Präprozessor, der die Bibliotheken einbindet. Mit dem *include* können wir aber auch andere Dateien, z.B. Klassen oder Funktionen, importieren.

4.1. Math-Bibliothek

Die Mathematik-Bibliothek wird mit `# include <math.h>` eingebunden. Wir haben nun Zugriff auf alle Funktionen die in dieser enthalten sind. Das sind z.B.:

- `pow(x,a) = xa`: Potenzen
- `sqrt(x) = √x`: Quadratwurzel
- `cos(x)`, `sin(x)`, `tan(x)`: Trigonometrische Funktionen

4. Bibliotheken

- `exp(x) = ex`: Exponentialfunktion
- `ceil(x)`, `floor(x)`: Rundet auf bzw. ab

4.2. Strings

Wir haben *strings* (Zeichenketten) bereits kennen gelernt. Alle Texte die zwischen „ ... “ stehen werden als `string` verwendet. Diese mussten aber alle bereits zur Compilerzeit bekannt sein. Mithilfe der *string*-Bibliothek können wir nun mit *strings* als Variablen arbeiten. Die *string*-Bibliothek wird mit `# include <string>` eingebunden.

Wir können nun *strings* wie alle anderen Variablentypen deklarieren:

```
string wort="Hello World!";  
  
cout<<wort<<endl;
```

Dies erzeugt wieder die Ausgabe:

```
Hello World!
```

Die *string*-Bibliothek stellt uns nun eine Reihe von Funktionen bereit, die uns erlauben *strings* zu manipulieren. Hier sind einige Beispiele:

operator = : Wert Zuweisung

operator [i]: Gibt den *i*. *char* im *string* zurück

operator +=: Hängt ein *string* an einen anderen

size: Gibt die Länge des Strings zurück

insert: In *string* einfügen

erase: Aus *string* löschen

replace: In *string* ersetzen

find: Im *string* suchen

Im Zusammenhang mit der *string*-Bibliothek ist noch der `getline()` Befehl zu erwähnen, mit dem man ganze *strings* einlesen kann:

4. Bibliotheken

```
cout << "Dein Name?" << endl;
string str;
getline (cin, str);
cout << "Hallo" << str << endl;
```

Dies liefert z.B. die Ausgabe:

```
Dein Name?
Max
Hallo Max
```

Da es sich bei der *string*-Bibliothek in *C++* um eine Klasse handelt (dazu später mehr), ist die Syntax für *string*-Befehle wie folgt:

```
string.befehl();
```

Diese Punktnotation liest sich so, dass der Befehl auf den *string* angewendet wird.

Um das zu verdeutlichen, wollen wir nun ein Programm schreiben, welches ein eingegebenes Wort rückwärts wieder ausgibt:

```
int main (){
    string wort;
    cout << "Bitte Wort eingeben: " << endl;
    getline (cin, wort);
    //bis hier wird nur ein "wort" eingelesen
    string buffer;
    //der buffer nimmt rückwärts die Buchstaben vom "
    wort" auf
    for (int i=0; i<wort.size(); i++){
        //dies geschieht solange bis das Ende des Wortes
        erreicht ist
        //size() gibt die Länge an
        buffer+=wort[word.size()-(i+1)];
        //allerdings beginnt die Numerierung der Zeichen
        bei
        //daher die etwas komische Modifizierung im []-
        Block
    }
    cout << buffer << endl;
    return 0;
}
```

5. Sortierproblem

5.1. Definition des Sortierproblems

Es sei eine Folge von n Zahlen gegeben. Aufgabe ist es diese Folge in aufsteigender Reihenfolge zu sortieren. Beispiel:

5, 8, 3, 11, 1

soll sortiert werden. Das Ergebnis ist offensichtlich

1, 3, 5, 8, 11.

5.2. Bubble-Sort Algorithmus

Mögliche Vorgehensweise: Wir vergleichen die ersten beiden Elemente der Folge, falls das erste größer als das zweite ist vertauschen wir sie. Nun vergleichen wir zweites und drittes Element und vertauschen wenn dies notwendig ist. So gehen wir sukzessive durch die Folge durch und erhalten

5, 3, 8, 1, 11.

Hierbei fällt auf, dass das größte Element bereits an die richtige Stelle gewandert ist. Wir wiederholen dieses Vorgehen $n - 1$ mal und sortieren so die Folge in aufsteigender Reihenfolge. Hierbei können wir berücksichtigen, dass beim j -ten Durchlauf die hinteren j Elemente bereits richtig sortiert sind.

```
#include <iostream>

using namespace std;

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

    int n = 5;
    int x[n];
    x[0] = 5;
    x[1] = 8;
    x[2] = 3;
    x[3] = 11;
    x[4] = 1;

    for(int j = 0; j < (n-1); j++) {
        for(int i = 0; i < (n-1-j); i++) {
```

5. Sortierproblem

```
        if(x[i] > x[i+1]) {
            int temp = x[i];
            x[i] = x[i+1];
            x[i+1] = temp;
        }
    }
}

for(int i = 0; i < n; i++)
    cout << x[i] << endl;

return 0;
}
```

Dieser Algorithmus wird als *Bubblesort-Algorithmus* bezeichnet. Es gibt einige weitere Sortieralgorithmen, die teilweise deutlich schneller sind als *Bubblesort*.

6. Zeiger

Bisher haben wir Variablen immer in der Form

```
int x = 5;
float f = 3.0;
```

angelegt. Hierbei haben die Variablen einen Namen (z.B. x) und man kann ihnen Werte (z.B. 5) zuweisen. Es ist wünschenswert ein Konstrukt zu haben, das auf den Inhalt von Variablen zeigt und diese manipulieren kann ohne den Namen der Variablen selbst zu kennen. Dieses Konstrukt wird als *Zeiger* bezeichnet und kann wie folgt verwendet werden:

```
// Deklarieren einer ganzzahligen Variablen:
int x = 5;

// Deklarieren eines Zeigers auf eine
// ganzzahlige Variable:
int* p;

// p soll auf den Wert von x zeigen:
p = &x;

// Den Wert von der Variable auf die p zeigt
// manipulieren:
*p = 8;

// Den Wert der Variable auf die p zeigt ausgeben:
cout << *p << endl;
```

Die Ausgabe dieses Programmes ist logischerweise 8.

6.1. Call-by-value und call-by-reference

Wir möchten ein Programm zur Verwaltung von Konten programmieren. Der Kontostand jedes Konto wird durch eine ganzzahlige Variable programmiert. Es soll eine Funktion geschrieben werden, die als Übergabeparameter ein Konto und einen abzuhebenden Betrag erhält. Die Funktion soll dann den Kontostand entsprechend anpassen. Betrachten wir folgenden Ansatz:

```
void abheben(int konto, int betrag) {
    konto = konto - betrag;
}
```

6. Zeiger

Wir rufen die Funktion im Hauptprogramm auf:

```
int konto1 = 500;
abheben(konto1, 200);
cout << konto1 << endl;
```

Die Ausgabe ist 500, da der Wert der Variable *konto1* beim Aufruf der Funktion *abheben* in die Variable *konto* der Funktion kopiert wird. Ändern wir nun den Wert der Variable *konto* innerhalb der Funktion *abheben* hat das keinen Einfluß auf die ursprüngliche Variable *konto1*. Parameterübergaben dieser Form werden als *call-by-value* bezeichnet. Bisher haben wir keine Möglichkeit Variablen innerhalb von Funktionen zu manipulieren, die wir als Parameter in diese Funktionen übergeben. Mit Hilfe von Zeigern läßt sich dies realisieren.

```
// Wir übergeben einen Zeiger auf eine
// ganzzahlige Variable (konto). So
// können wir diese Variable innerhalb
// der Funktion manipulieren:
void abheben(int* konto, int betrag) {
    // mit *konto greifen wir auf den Wert
    // der Variable zu, auf die der Zeiger
    // konto zeigt:
    *konto = *konto - betrag;
}
```

Im Hauptprogramm rufen wir die Funktion wie folgt auf:

```
int konto1 = 500;
// konto1 ist eine ganzzahlige Variable, die Funktion
// abheben verlangt als Übergabeparameter aber einen
// Zeiger auf eine ganze Zahl. Daher muss vor dem
// Variablennamen konto1 der sog. Adressoperator &
// stehen:
abheben(&konto1, 200);
cout << konto1 << endl;
```

Nun wurde wirklich von der Variable *konto1* der Wert 200 abgezogen und die Ausgabe des Programms ist 300. Mit der Übergabe von Zeigern lassen sich also die Werte von Variablen in Funktionen manipulieren. Eine Parameterübergabe dieser Form bezeichnet man als *call-by-reference*.

6.2. Zeiger und Felder

Wir betrachten folgendes Programm:

```
int f[5];
int* p;
p = f;
p[3] = 17;
cout << p[3] << endl;
```

Die Ausgabe ist 17. Wir weisen der Variable p , die ein Zeiger auf eine ganze Zahl ist, den Wert der Variable f zu, die ein Feld aus ganzen Zahlen symbolisiert. Da der Compiler bei diesem Vorgang keinen Fehler ausgibt, war die Zuweisung offenbar gültig. Somit müssen die Variablen p und f vom gleichen Typ sein. Der Typ von p ist bekannt, es handelt sich um einen Zeiger auf eine ganze Zahl. Somit ist auch f ein Zeiger auf eine ganze Zahl. f zeigt nämlich auf das erste Element des Feldes. Dies können wir uns zu Nutze machen, wenn wir Felder an Funktionen übergeben möchten. Wir können beispielsweise den *Bubble-Sort* Algorithmus aus dem letzten Kapitel als Funktion programmieren. Wichtig ist hierbei, dass die Dimension des Feldes als weitere Variable übergeben wird, damit sie in der entsprechenden Funktion bekannt ist.

```
void bubble_sort(int* x, int n) {
    for(int j = 0; j < (n-1); j++)
        for(int i = 0; i < (n-1-j); i++)
            if(x[i] > x[i+1]) swap(x[i], x[i+1]);
}
```

6.3. Dynamische Speicherverwaltung

Mit Hilfe von Zeigern ist es möglich Variablen anzulegen, ohne dass diese einen Namen besitzen. Dies bezeichnet man als *dynamische Speicherallokierung*.

```
// Anlegen einer Variable vom Typ int
// mit Hilfe eines Zeigers:
int* p = new int;

// Wert zuweisen (wie oben):
*p = 5;

// Variable wieder löschen:
delete p;
```

6. Zeiger

```
// Es können auch Felder auf diese Weise  
// erzeugt werden:  
int n = 25;  
int* f = new int[n];  
  
// Feld löschen:  
delete [] f;
```

Wichtig ist, dass man auf diese Weise angelegte Variablen auch wieder löscht. Betrachten wir folgende Funktion:

```
int funktion(int n) {  
    int i = n;  
    return i;  
}
```

Die Variablen n und i werden innerhalb bzw. als Übergabeparamter in der Funktion deklariert. Sie werden beim Beenden der Funktion automatisch gelöscht. Betrachten wir eine weitere Funktion:

```
int funktion2(int n) {  
    int* p = new int;  
    *p = n;  
    return *p;  
}
```

Die Variablen n und p sind analog zu oben innerhalb der Funktion deklariert. Sie werden somit beim Beenden der Funktion gelöscht. Über den Zeiger p erzeugen wir aber eine weitere Variable. Diese Variable überlebt das Beenden der Funktion. Da der Zeiger auf die Variable aber mit dem Beenden der Funktion gelöscht wird, haben wir eine Variable erzeugt, die Speicherplatz blockiert, aber auf die wir nicht mehr zugreifen können. Es ist daher wichtig, dass dynamisch erzeugte Variablen gelöscht werden, wenn sie nicht mehr gebraucht werden oder bevor der letzte Zeiger auf eine solche Variable gelöscht wird.

6.4. Einfach verkettete Listen

Angenommen man möchte ein Verwaltungssystem für Bankkonten programmieren. Ein Konto sei durch eine ganzzahlige Variable, den Kontostand, definiert. Für eine große Bank könnte man durch ein Feld viele Konten auf einmal anlegen. Man müsste zur Entwicklung des Programms allerdings die Größe dieses Feldes festlegen. Wenn nun mehr Konten eröffnet werden sollen, als das Programm vorsieht, bekommt man ein Problem. Dies könnte man umgehen, in dem man das

6. Zeiger

Feld so groß macht, dass es fast sicher ist, dass niemals so viele Kunden Konten eröffnen möchten. Dann würde man aber viel Speicherplatz verschwenden, da man wahrscheinlich viel weniger Konten benutzen würde, als in der Software vorgesehen sind. Mit Hilfe von Zeigern läßt sich eine Datenstruktur definieren, die je nach Bedarf ein weiteres Konto hinzufügen oder Konten auch löschen kann:

```
class Konto {  
  
    public:  
        int kontostand;  
        Konto* naechstes;  
};
```

Ein Konto besteht also aus dem Kontostand und einem Zeiger auf das nächste Konto. Hat man nun mehreren Konten innerhalb der Software, so zeigt jedes Konto auf seinen Nachfolger. Der Zeiger des letzten Kontos zeigt zunächst nirgendwohin. Soll ein neues Konto angelegt werden, so wird dieser Zeiger dazu verwendet ein neues Konto zu erzeugen und zeigt fortan auf dieses neue Konto. Der Zeiger des neuen Kontos zeigt dann wiederum ins Leere. Zur Definition der Klasse *Konto* haben wir wie in Kapitel 5 gelernt, eine Klasse konzipiert. Diese Klasse besitzt allerdings nur Eigenschaften und keine Methoden. Eine solche Zusammenfassung von mehreren Variablen in eine *Struktur* kann auch folgendermaßen programmiert werden:

```
struct Konto {  
    int kontostand;  
    Konto* naechstes;  
};
```

Alle Variablen innerhalb eines solchen *structs* sind im Sinne der Objektorientierung öffentlich und können somit überall im Programm manipuliert werden. Um eine Liste zu generieren benötigen wir zwei Variablen:

- Einen Zeiger auf das erste Listenelement.
- Die Anzahl der Listeneinträge.

Wir möchten folgende Operationen auf unserer Liste programmieren:

- Element am Ende der Liste einfügen.
- Eintrag des *i*-ten Elements ausfindig machen.
- Suchen, ob Element mit Wert *w* vorhanden ist.

Eine mögliche Lösung wäre folgendes Programm:

6. Zeiger

```
#include <iostream>

using namespace std;

struct ListElement {
    int value;
    ListElement *next;
};

// Funktion um ein neues Element ans Ende der Liste zu h
// ängen.
// l ist ein Zeiger auf das erste Element der Liste.
// p_count ist ein Zeiger auf die Variable, die
// die Anzahl der Listenelemente enthält. Diese
// wird als Referenz übergeben, weil sie innerhalb
// der Funktion modifiziert werden soll.
// value ist der Wert des neuen Listenelements.
void new_element(ListElement* l, int *p_count, int value
) {

    // Gehe ans Ende der Liste:
    for(int i = 0; i < (*p_count - 1); i++)
        l = l->next;

    // Erzeuge neues Listenelement:
    l->next = new ListElement;
    // Weise übergebenen Wert zu:
    l->next->value = value;
    // next-Zeiger des neuen Elements ist NULL:
    l->next->next = NULL;

    // Anzahl Listenelemente erhöhen:
    *p_count = *p_count + 1;
}

// Eintrag des n-ten Listenelements herausfinden.
// Wir fangen bei 0 an zu zählen.
// l: Zeiger auf erstes Listenelement.
// count: Anzahl Listenelemente.
// n: Index des gesuchten Listenelements.
int get_value(ListElement* l, int count, int n) {

    // 0 zurückgeben, wenn gesuchtes Element nicht
    // vorhanden ist:
```

6. Zeiger

```
if(n > (count - 1)) return 0;

// Gehe an richtige Stelle der Liste:
for(int i = 0; i < (count - 1); i++)
    l = l->next;

return l->value;
}

// Suche ob Element mit Wert value in Liste
// vorhanden ist:
// l: Zeiger auf erstes Listenelement.
// count: Anzahl Listenelemente.
// value: Gesuchter Wert.
bool search_value(ListElement* l, int count, int value)
{
    // ganze Liste durchsuchen:
    for(int i = 0; i < count; i++) {
        // schauen ob aktuelles Element den Wert
        // value enthält:
        if(l->value == value) return 1;
        l = l->next;
    }
    // false zurückgeben, falls value nicht in
    // der Liste gefunden wurde:
    return false;
}

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

    // Zeiger auf erstes Listenelement,
    // dass zunächst erzeugt wird:
    ListElement *top = new ListElement;
    top->next = NULL;
    top->value = 0;
    // Anzahl Listenelemente:
    int count = 1;

    // Hier testen wir die Funktionen::

    for(int i = 1; i < 200; i++)
        new_element(top, &count, i);
}
```

7. Übungsaufgaben

```
cout << "17. Listeneintrag ausgeben:" << endl;
cout << get_value(top, count, 17) << endl << endl;

cout << "Suche nach 0:" << endl;
cout << search_value(top, count, 0) << endl;

cout << "Suche nach 35:" << endl;
cout << search_value(top, count, 35) << endl;

cout << "Suche nach 199:" << endl;
cout << search_value(top, count, 199) << endl;

cout << "Suche nach 200:" << endl;
cout << search_value(top, count, 200) << endl;

return 0;
}
```

7. Übungsaufgaben

7.1. Primzahltest

Wir möchten eine Funktion schreiben, die als Ergebnis liefert ob eine Zahl eine Primzahl ist oder nicht. Zunächst definieren wir uns also eine Funktion *primzahl*, die wir dann programmieren:

$$\text{primzahl}(n) = \begin{cases} 1, & \text{falls } n \text{ prim} \\ 0, & \text{sonst} \end{cases}$$

Die Idee ist nun, zu testen ob die Zahl n durch eine Zahl zwischen 2 und $n - 1$ teilbar ist. Dazu versuchen wir n durch all diese Zahlen zu teilen. Finden wir einen Teiler darunter, ist n offenbar nicht prim. Finden wir keine Teiler handelt es sich um eine Primzahl. Der Rückgabewert der Funktion ist entweder 0 oder 1, in anderen Worten *false* oder *true*. Wir können als Datentyp des Rückgabewertes also *bool* verwenden. Eine fertige Implementierung könnte so aussehen:

```
bool primzahl(int n) {
```

7. Übungsaufgaben

```
// Wenn n kleiner 1 ist, kann n nicht prim sein:
if(n <= 1) return false;

// Suche einen Teiler zwischen 2 und n-1:
for(int i = 2; i < n; i++)
    if((n % i) == 0) return false;

// Wenn kein Teiler gefunden wurde, ist n prim:
return true;
}
```

7.2. Übung zu Feldern

Aufgabe ist es ein Feld der Größe n zu erzeugen, in dem im i -ten Eintrag gespeichert wird, ob die Zahl i eine Primzahl ist oder nicht. Anschließend sollen alle Primzahlen die kleiner sind als n mit Hilfe dieses Feldes ausgegeben werden. Als Hilfe verwenden wir die Primzahltestfunktion von oben. Das komplette Programm würde dann so aussehen:

```
#include <iostream>

using namespace std;

bool primzahl(int n) {

    // Wenn n kleiner 1 ist, kann n nicht prim sein:
    if(n <= 1) return false;

    // Suche einen Teiler zwischen 2 und n-1:
    for(int i = 2; i < n; i++)
        if((n % i) == 0) return false;

    // Wenn kein Teiler gefunden wurde, ist n prim:
    return true;
}

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

    // Variable n gibt die Größe des Feldes an:
    int n = 50;
    // Feld mit n Elementen vom Typ bool:
    bool ist_primzahl[n];

    // Speichere in i-ten Eintrag ob i
```

7. Übungsaufgaben

```
// eine Primzahl ist, oder nicht:
for(int i = 1; i < n; i++)
    ist_primzahl[i] = primzahl(i);

// Gebe nur Zahlen aus, die prim sind:
for(int i = 1; i < n; i++)
    if(ist_primzahl[i] == true) cout << i << endl;

return 0;
}
```

7.3. Größter gemeinsamer Teiler

Aufgabe ist es eine Funktion zu schreiben, die den größten gemeinsamen Teiler (ggT) zweier natürlicher Zahlen $a \geq b$ berechnet. Ein naiver Ansatz ist, für alle Zahlen $i \leq b$ auszuprobieren ob sie a und b teilen und hiervon die größte als Ergebnis zu nehmen. Die zugehörige Funktion kann folgendermaßen aussehen:

```
int ggT(int a, int b) {

    // Wenn a < b ist, werden die beiden Variablen
    vertauscht:
    if(a < b) {
        int temp = a;
        a = b;
        b = temp;
    }

    // Ansonsten gehe alle Zahlen von b bis 1 durch und
    überprüfe die Teilbarkeit. Da von oben angefangen
    wird, ist die erste Zahl i, die sowohl a als auch b
    teilt, der ggT von a und b.
    for(int i = b; i > 0; i--)
        if((a % i == 0) && (b % i == 0))
            return i;
}
```

Bei großen teilerfremden Zahlen ist dieses Vorgehen wenig effizient. Allgemein sind $2 \cdot (b - \text{ggT}(a, b))$ viele Teilbarkeitstests notwendig um den größten gemeinsamen Teiler auf diese Weise zu berechnen. Effizienter ist die Verwendung des euklidischen Algorithmus.

7. Übungsaufgaben

7.4. Euklidischer Algorithmus

Es sei der größte gemeinsame Teiler von $a \geq b$ zu berechnen. Mit $r_0 = b$ betrachten wir folgende Rechenschritte:

$$\begin{aligned}a &= q_1 r_0 + r_1 \text{ mit } r_0 > r_1 \\r_0 &= q_2 r_1 + r_2 \text{ mit } r_1 > r_2 \\r_1 &= q_3 r_2 + r_3 \text{ mit } r_2 > r_3 \\&\dots \\&\dots \\r_{n-1} &= q_{n+1} r_n + 0\end{aligned}$$

Hierbei gilt jeweils

$$r_{i+2} = r_i \pmod{r_{i+1}}.$$

Dann ist r_n der größte gemeinsame Teiler von a und b . Setzt man r_n sukzessive in die vorherigen Gleichungen ein, ist schnell einzusehen, dass r_n Teiler von a und b (und allen anderen r_i) ist. Es lässt sich zudem zeigen, dass r_n der größte dieser Teiler ist, was wir hier aber nicht genauer beleuchten möchten. Aus den obigen Gleichungen lässt sich aber ableiten, dass das Problem den ggT von a und b zu berechnen auf das Problem den ggT von b und $a \pmod{b}$ zu berechnen vereinfacht werden kann. Ist $b = 0$, so haben wir mit a den größten gemeinsamen Teiler gefunden. Diese Beobachtung lässt sich auf einfache Weise rekursiv programmieren:

```
int euklid(int a, int b) {  
    if(b == 0) return a;  
    else return euklid(b, a % b);  
}
```

Ein Beispiel des euklidischen Algorithmus. Es sei $ggT(1071, 1029)$ zu bestimmen.

$$1071 = 1 \cdot 1029 + 42$$

$$1029 = 24 \cdot 42 + 21$$

$$42 = 2 \cdot 21 + 0$$

Somit gilt: $ggT(1071, 1029) = 21$. Bei dem Beispiel zeigt sich, dass der euklidische Algorithmus deutlich schneller als obige naive Ansatz zur Berechnung des größten gemeinsamen Teilers ist.

A. Kurze Einführung in Linux

A.1. Wichtige Befehle in der Konsole

- `cd` Wechseln der Verzeichnisse
- `cd ..` Ein Verzeichnis aufwärts
- `mkdir` Erzeugen eines neuen Verzeichnisses
- `rmdir` Löschen eines Verzeichnisses
- `ls` Anzeigen des Verzeichnisinhalts
- `ls -a` Anzeigen des Verzeichnisinhalts mit versteckten Dateien
- `ls -l` Anzeigen des Verzeichnisinhalts mit zusätzlichen Informationen
- `ls -al` Kombination von `ls -l` und `ls -a`
- `clear` Löschen des Bildschirminhalts
- `passwd` Ändern des Passworts

A.2. Wichtige Programme

Um weitere Befehle in der Konsole ausführen zu können, sollten die folgenden Programme mit dem Suffix `&` gestartet werden.

- `gedit`: Editor zum schreiben von Textdateien.
- `mozilla`: Internetbrowser.
- `g++`: C++ Compiler. Beispiel zur Benutzung: `g++ program.cpp -o program-name`.

B. C++ unter Windows

In diesem Anhang wird kurz die Installation und Benutzung der freien Entwicklungsumgebung *DevCPP* für Windows erklärt. *DevCPP* verwendet den *g++-Compiler*, den wir unter Linux ebenfalls verwendet haben. Hierdurch können wir das gelernte C++ eins zu eins mit *DevCPP* unter Windows verwenden.

B.1. Installation

Lade die Installationsdatei "devcpp-4.9.9.2_setup.exe" von www.bloodshed.net herunter, starte sie und folge den Anweisungen. Die Vorgaben können übernommen

B. C++ unter Windows

werden. Der Standardinstallationspfad ist "C:\Dev-Cpp". Im weiteren wird angenommen, dass als Installations Sprache "Deutsch" gewählt wurde.

B.2. Deinstallation

Solltest Du den Dev-Cpp deinstallieren wollen, führe unbedingt die Datei "uninstall.exe" im Verzeichnis "C:\Dev-Cpp" aus und lösche dann anschließend den gesamten Ordner "C:\Dev-Cpp". Wenn Du auf andere Weisen versuchst, Dev-Cpp von Deinem Computer zu entfernen, kann es sein, dass er sich hinterher nicht wieder installieren lässt!

B.3. Erstellen eines Konsolenprogramms

1. Starte *Dev-CPP*.
2. Wähle aus der oberen Menüleiste "Datei→Neu→Projekt".
3. Klicke auf "Console Application" und gib links unten einen Namen für das Projekt ein (dies wird später der Name der *.exe Datei). Klicke dann auf "Okay".
4. Nun musst Du das neue Projekt speichern. Du erstellt dazu am besten einen neuen Ordner, und zwar außerhalb des zunächst angezeigten Pfades "C:\Dev-Cpp" (z. B. in den "Eigenen Dateien"). Speichere dann in den neuen Ordner das Projekt.
5. Es ist sehr zu empfehlen, für jedes neue Projekt auch einen neuen Ordner zu verwenden, da es sonst Konflikte mit den Dateinamen geben kann.
6. Wähle anschließend "Datei→Speichern", um die Hauptdatei Deines Projekts zu speichern. Hier kann man den vorgeschlagenen Dateinamen "main.cpp" beibehalten oder auch beliebig abändern.
7. Projekt kompilieren: Wähle "Ausführen→Kompilieren und Ausführen" (Shortcut: F9). Hierbei werden alle Projektdateien automatisch gespeichert.
8. Jetzt kannst Du Deinen eigenen Code eingeben. Das Kompilieren geht wie oben beschrieben.

B.4. Mehrere Projektdateien mit Dev-CPP

Möchte man mit der *Dev-CPP* Entwicklungsumgebung mehrere Dateien in einem Projekt erzeugen, so geht man folgendermaßen vor:

- Man erzeuge ein Projekt wie in Anhang A beschrieben.
- Nun klickt man auf die Menüleiste "Projekt → Neue Datei".
- Diese Datei speichert man unter dem gewünschten Namen, sie gehört nun zum aktuellen Projekt.
- Diesen Vorgang muss man wiederholen, wenn man sowohl eine "*.cpp" als auch eine "*.h" Datei zum Projekt hinzufügen möchte.
- Compiliert man nun das Projekt wie oben beschrieben erzeugt *Dev-CPP* automatisch ein Makefile in dem die nötigen Informationen zur Compilierung aller Projektdateien reingeschrieben werden.
- Dieses Makefile wird im Projektordner unter dem Namen "Makefile.win" abgelegt. Es ist sehr lehrreich dieses Makefile zu lesen.

C. Makefiles

Um größere Programme übersichtlich zu gestalten bietet es sich an den Programmcode auf mehrere Dateien zu verteilen. Um aus diesen einzelnen Dateien später ein ausführbares Programm zu erzeugen verwenden wir ein sogenanntes *Makefile*. Dieses Makefile wird die einzelnen Projektdateien compilieren und zu einem einzigen Programm zusammenbauen. Als Beispiel betrachten wir folgendes Szenario:

```
Datei main.cpp:  
  
#include <iostream>  
#include "quadrat.h"  
  
using namespace std;  
  
int main(int argc, char **argv) {  
  
    cout << quadrat(3) << endl;  
    return 0;  
}
```

C. Makefiles

```
}  
  
Datei quadrat.h:  
  
#ifndef _QUADRAT_H_  
#define _QUADRAT_H_  
  
int quadrat(int n);  
  
#endif
```

```
Datei quadrat.cpp:  
  
#include "quadrat.h"  
  
int quadrat(int n) {  
    return n * n;  
}
```

Wir haben unsere Quadratfunktion von oben also in die Datei "quadrat.cpp" ausgelagert. Die Datei "quadrat.h" wird als *Header-Datei* bezeichnet und dient als Schnittstelle zwischen den einzelnen Dateien. In ihr wird lediglich definiert, was es für Funktionen in der Datei "quadrat.cpp" gibt. Die wirkliche Implementierung der einzelnen Funktionen findet dann in "quadrat.cpp" statt. Über den Befehl

```
#include "quadrat.h"
```

können wir dann in der Datei "main.cpp" alle in "quadrat.h" definierten Funktionen einbinden und verwenden. Betrachten wir nun ein Makefile, das dieses Projekt vollständig compilieren kann. Kommentare in der Sprache *make* werden mit *#* eingeleitet.

```
# Festlegen des Compilers:  
CPP = g++  
  
# Das Programm quadrat wird neu erzeugt  
# wenn sich main.o oder quadrat.o geändert haben:  
quadrat: main.o quadrat.o  
    $(CPP) main.o quadrat.o -o quadrat  
  
# Durch "make clean" können wir alle durch  
# das Compilieren erzeugten Dateien löschen:  
clean:  
    -rm -f *.o quadrat  
  
# main.o wird neu erzeugt, wenn sich
```

D. Einige nützliche Links

```
# main.cpp oder quadrat.h geändert haben:
main.o: main.cpp quadrat.h
    $(CPP) -c main.cpp -o main.o

# quadrat.o wird neu erzeugt, wenn sich
# quadrat.cpp oder quadrat.h geändert haben:
quadrat.o: quadrat.cpp quadrat.h
    $(CPP) -c quadrat.cpp -o quadrat.o
```

Hierbei ist zu beachten, dass die Zeilen, in der die Anweisungen zur Compilierung von Dateien stehen, durch ein Tabulator eingeleitet werden. Wir speichern diese Datei unter dem Namen "Makefile" in das selbe Verzeichnis wie die übrigen Projektdateien. Nun können wir in der Konsole mit den Befehlen

```
make
make clean
```

das Projekt compilieren, bzw. alle compilierten Dateien löschen. Ändern wir nun eine der Quelldateien werden nur die entsprechend veränderten, bzw. von der Veränderung abhängigen Dateien neu compiliert. Zu beachten gilt zudem, dass nur die "*.cpp" Dateien compiliert werden. Die "*.h" Dateien dienen wie oben beschrieben lediglich als Schnittstelle zwischen den einzelnen "*.cpp" Dateien, werden selbst aber nicht compiliert.

D. Einige nützliche Links

<http://www.cplusplus.com> : C++-Referenz

<http://www.cppreference.com> : C++-Referenz

<http://www.bloodshed.net> : Dev-CPP-Compiler