Werner-von-Siemens-Gymnasium Weißenburg / Bay.

Kollegstufe Abiturjahrgang 2003

Facharbeit

aus **Mathematik**

Thema: Exakte und näherungsweise Berechnung von Fakultäten und Binomialkoeffizienten

Verfasser:	Schuh Andreas
Leistungskurs:	3M2
Kursleiter/in:	Ernst Reinhard
Bearbeitungszeitraum:	November 2002 – Januar 2003
Abgabetermin:	03. Februar 2003

Inhaltsverzeichnis

1 Vorwort

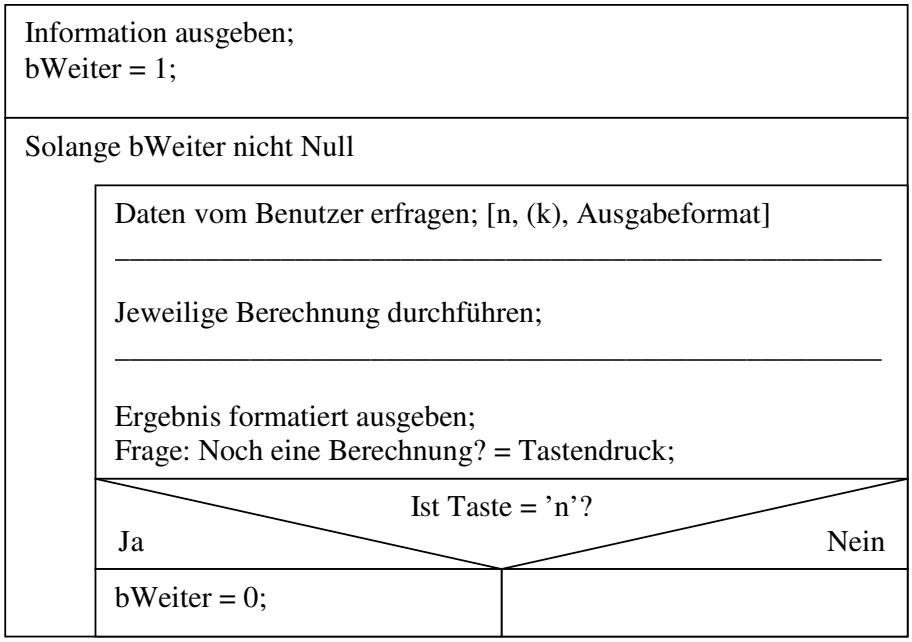
1.1 Allgemeines zu den Programmen

Auf der beiliegenden CD dieser Arbeit befinden sich etliche Programme und deren Quellcodes. Alle Programme sind mit Borland C++ 5.02 erstellt. Eine Kenntnis dieser oder ähnlicher Programmiersprachen wird vorausgesetzt. Falls Sie vorhaben diese Sprache zu erlernen, kann ich das Buch *„Borland C++ 5 – Das Kompendium“* von Dirk Louis empfehlen, welches auch mir zum Selbststudium diente. Im Internet sind ebenfalls viele interessante Seiten zu diesem Thema vorhanden und zuletzt ist mir persönlich noch die Online-Hilfe von Borland C++ immer eine nützliche Hilfe gewesen. Ein Großteil der Programme sind einfache DOS-Programme, die lediglich dazu dienen die einzelnen Berechnungsmethoden auf schlichte Weise zu demonstrieren. Wobei ich gestehen muss, dass es mir nicht möglich war die DOS-Programme zur exakten Berechnung auch für größere Berechnungen zum Laufen zu bringen. Sie dienen somit lediglich zur überschaubaren Erläuterung der Programmiertechnik. Bei der exakten Berechnung von Fakultäten und Binomialkoeffizienten rate ich zu dem Windows-Programm **FakBinomial**, dem eigentlichen Hauptprogramm dieser Arbeit. Des Weiteren findet sich noch ein Windows-Programm zum Rechnen mit langen Zahlen genannt **CalcLongInt**. Beide Windows-Programme sind für 32-Bit Systeme programmiert, also ab Windows 95 und Windows NT 4.0. Der Programmieraufwand für diese Programme ist um einiges größer und die Quellcodes nicht unbedingt für jedermann verständlich. Hierfür sollte man sich zunächst in die Windows-Programmierung einarbeiten. Zu diesem Zweck möchte ich auf das wohl mit Abstand wichtigste Buch zur Windows-Programmierung mit der Windows-API (Anwendungs-programmierschnittstelle von Windows) hinweisen, nämlich Charles Petzold´s Buch *„Windows-Programmierung“*. Allerdings muss man sagen, dass der Preis für dieses Buch sehr hoch ist. Die Investition lohnt sich jedoch, wenn man sich für die Windows-Programmierung interessiert. Genauere Angaben zu den beiden erwähnten Titeln finden sich im Literaturverzeichnis.

Möchte man die beiden Win32-Programme von der Festplatte aus starten, muss man sicherstellen, dass sich die auf der CD befindliche Dynamische Linkbibliothek (**D**ynamic **L**ink **L**ibrary) **cw3230.dll**, die mit Borland C++ mitgeliefert wird, entweder im selben Verzeichnis wie die jeweilige Programmdatei befindet oder im System-Verzeichnis von Windows (normalerweise: C:\WINDOWS\SYSTEM).

1.2 Grundgerüst aller DOS-Programme

Alle im Rahmen dieser Arbeit erstellten DOS-Programme haben das gleiche Grundgerüst. Lediglich die Berechnungen sind von Programm zu Programm verschieden, je nachdem welche Methode aufgezeigt werden soll bzw. ob eine Fakultät oder ein Binomialkoeffizient berechnet werden soll. Letzteres zeigt auch einen geringfügigen Unterschied im Grundgerüst, da man bei Binomialkoeffizienten natürlich nicht nur *n* sondern auch *k* vom Benutzer erfragen muss.
Zunächst einmal wird eine Information ausgegeben, die dem Benutzer mitteilt, um welche Berechnung es sich in dem Programm handelt. Danach tritt das Programm in eine Schleife ein, die so lange ausgeführt wird, bis der Benutzer bei der Frage, ob er noch eine Berechnung durchführen möchte mit Nein antwortet und somit das Programm beendet. Innerhalb dieser Schleife werden zunächst die für die Berechnung notwendigen Daten erfragt, unter anderem auch, wie das Ergebnis ausgegeben werden soll. Daraufhin folgt die Berechnung und anschließende formatierte Ausgabe des Ergebnisses. Veranschaulicht wird dies in Struktogramm 1.1 dargestellt.



Struktogramm 1.1: Grundgerüst aller DOS-Programme

2 Näherungsweise Berechnung

2.1 Einführung

Da man sehr große Zahlen nicht auf herkömmliche Weise berechnen kann, bedient man sich eines kleinen Tricks. Man berechnet zunächst den dekadischen Logarithmus dieser Zahl und erhält so das Ergebnis in der Form $\lg(z) = EE,mmm$, wobei z die zu berechnende große Zahl ist.

Nun gilt: $z = 10^{\lg(z)} = 10^{EE,mmm}$

Nach den Rechengesetzen der Potenzen erhält man nun durch Abspalten des ganzzahligen Teils *EE* den Exponenten und durch $10^{0,mmm}$ die Mantisse.

$$\Rightarrow \qquad z = 10^{0,mmm} \cdot 10^{EE}$$

Natürlich kann man auf diese Art die große Zahl auch nur gerundet auf eine bestimmte maximale Anzahl geltender Ziffern angeben, aber immerhin hat man auf schnelle und einfache Art ein Ergebnis, das den meisten Ansprüchen genügt. Möchte man große Zahlen bis auf die letzte Stelle genau wissen, sieht die Sache schon etwas komplizierter aus. Doch dies heben wir uns für Kapitel 3 zur exakten Berechnung auf.

2.2 Fakultäten

2.2.1 Sterlingsche Näherungsformel

Die Sterlingsche Näherungsformel für große Fakultäten aus dem Jahr 1730 lautet:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Der Beweis dieser Formel ist im Anhang aufgeführt.

Zusammenhang zwischen *n*! und *e*

In der Sterlingschen Näherungsformel kommt die Eulersche Zahl *e* vor. Dem Zusammenhang, der zwischen der Fakultät von *n* und der Eulerschen Zahl *e* besteht, wollen wir nun auf den Grund gehen.

Fakultäten sind Mehrfachprodukte der Form: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.

Ersetzen wir die Faktoren durch das geometrische Mittel, so erhalten wir:

$$\overline{n} = \sqrt[n]{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n}$$

Somit gilt: $n! = \overline{n}^{\,n}$

Den Zusammenhang zwischen *n* und $\overline{n}$ zeigt Tabelle 2.2.1.

n	$\frac{n}{\overline{n}}$
10	2,208
20	2,408
50	2,566
10^2	2,6321
10^3	2,7002
10^4	2,7162
10^5	2,7180
10^6	2,7182

Tabelle 2.2.1

Aus Tabelle 2.2.1 lässt sich erkennen, dass offensichtlich gilt: $\lim_{n\to\infty} \frac{n}{\overline{n}} = e = 2{,}718282$

Oder: $\overline{n} \approx \frac{n}{e}$

Daraus folgt: $n! \approx \left(\frac{n}{e}\right)^n$

Ein Vergleich mit der Sterlingschen Näherungsformel zeigt, dass diese Beziehung den wesentlichen Term der Formel wiedergibt.

Das Programm FAKSTIR

Aus Tabelle 5.1 des Beweises geht hervor, dass die rechte Seite der Ungleichung (5) eine besonders gute Näherung für die Fakultät von *n* liefert. Deshalb wird nun folgende Approximation verwendet:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

Zunächst wird der natürliche Logarithmus berechnet und das Ergebnis durch Multiplikation mit $\frac{1}{\ln 10}$ in den dekadischen Logarithmus überführt.

$$\ln n! = \frac{1}{2}\ln(2\pi) + \left(n+\frac{1}{2}\right)\ln n - n + \frac{1}{12n}$$

$$\lg n! = \left(\frac{1}{2}\ln(2\pi) + \left(n+\frac{1}{2}\right)\ln n - n + \frac{1}{12n}\right)\frac{1}{\ln 10}$$

Daraus erhält man nun das Ergebnis, indem man wie unter Punkt 2.1 beschrieben verfährt. Im Folgenden ist der wesentliche vom Grundgerüst abweichende Quellcode vorzufinden.

```
29    if((n == 0) || (n == 1)) printf("\nn! = 1");
30    else
31    {
32       dF = 0.5*log(2*M_PI)+(n+0.5)*log(n)-n+1/(12*n);
33       dF /= log(10);
34       ma = modf(dF, &ex);
35       ma = exp(ma*log(10));
36
37       gcvt(ma, iZiffern, szMa);
38       itoa(ex, szEx, 10);
39
40       printf("\nn! = %s E %s", szMa, szEx);
41    }
```

In Zeile 32 und 33 des Quellcodes findet die Berechnung nach der oben hergeleiteten Formel statt. Daraufhin wird der ganzzahlige Teil mittels der Funktion *modf()* abgespalten und in der Variablen *ex* gespeichert. Anschließend wird die Mantisse berechnet. Da es keine Funktion für Zehnerpotenzen gibt, bei denen der Exponent keine natürliche Zahl ist, behelfen wir uns mit der Exponentialfunktion.

Nämlich: $10^x = e^{x\cdot\ln 10}$

Nun müssen die Ergebnisse noch in Strings umgewandelt werden, welche schließlich auf dem Bildschirm ausgegeben werden.

2.2.2 Logarithmische Berechnung

Unter Punkt 2.1 wurde beschrieben, wie man lange Zahlen anhand des dekadischen Logarithmus berechnet und anschließend die Mantisse und den Exponenten aus dem Ergebnis ermittelt. So lässt sich leicht eine einfache und genauere Näherung für große Fakultäten als mit der Sterlingschen Formel berechnen, es sei denn man hat keinen Computer zur Hand.

Es gilt: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n;$

Oder: $\lg(n!) = \lg(1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n);$

$\lg(n!) = \lg 1 + \lg 2 + \lg 3 + \ldots + \lg(n-1) + \lg n;$

Dies lässt sich mühelos mit einer Schleife programmieren. Daraus erkennt man allerdings schon, dass der Preis für diese genauere Näherung eine zeitaufwendigere Berechnung ist, was sich jedoch erst bei sehr großen Fakultäten – und das auch nur bei älteren Rechnern – bemerkbar macht. Bei der Geschwindigkeit, mit der heutige Computer rechnen ist dieser Zeitaufwand nahezu vernachlässigbar.

Das Programm FAKLOG

Die Umsetzung der logarithmischen Berechnung von Fakultäten in ein Computerprogramm zeigt das Programm FAKLOG, aus dessen Quellcode der folgende Ausschnitt stammt:

```
27    if((n == 0) || (n == 1)) printf("\nn! = 1");
28    else
29    {
30       dF = log10(2);
31       for(j = 3; j < n+1; j++) dF += log10(j);
32       ma = modf(dF, &ex);
33       ma = exp(ma*log(10));
34
35       gcvt(ma, iZiffern, szMa);
36       itoa(ex, szEx, 10);
37
38       printf("\nn! = %s E %s", szMa, szEx);
39    }
```

Die Berechnung erfolgt ausschließlich in der Schleife in Zeile 31, die, da $\lg 1 = 0$ und der Anfangswert lg 2 ist, (*n*-2)-mal durchlaufen wird. Innerhalb dieser Schleife wird zum Ergebnis *dF* der dekadische Logarithmus der Schleifenvariablen *j* addiert.
Zur Erläuterung der Umwandlung des Ergebnisses und dessen Ausgabe siehe Beschreibung des Programms FAKSTIR.

2.3 Binomialkoeffizienten

2.3.1 Stirlingsche Näherungsformel

Nun wenden wir die Sterlingsche Formel auf Binomialkoeffizienten an.
Da diese folgendermaßen definiert sind:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

kann man $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, $k! \approx \sqrt{2\pi k}\left(\frac{k}{e}\right)^k$ und $(n-k)! \approx \sqrt{2\pi(n-k)}\left(\frac{n-k}{e}\right)^{n-k}$ ersetzen und erhält somit folgende Näherung:

$$\binom{n}{k} \approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k \sqrt{2\pi(n-k)}\left(\frac{n-k}{e}\right)^{n-k}} =$$

$$= \sqrt{\frac{n}{2\pi(n-k)k}} \cdot \frac{n^n}{k^k (n-k)^{n-k}}$$

Das Programm BINOMSTI

Genauso wie beim Programm FAKSTIR wird zunächst der natürliche Logarithmus berechnet:

$$\ln\binom{n}{k} = -\frac{1}{2}\ln(2\pi) + \left(n+\frac{1}{2}\right)\ln n - \left(k+\frac{1}{2}\right)\ln k +$$

$$-\left(n-k+\frac{1}{2}\right)\ln(n-k) + \frac{1}{12}\left(\frac{1}{n} - \frac{1}{k} - \frac{1}{n-k}\right)$$

Daraus erhält man wiederum den dekadischen Logarithmus:

$$\lg\binom{n}{k} = \frac{\ln\binom{n}{k}}{\ln 10}$$

Der wichtigste Teil des Quellcodes sieht dementsprechend wie folgt aus:

```
32  if(k > n) printf("\nk muss kleiner als n sein!\n\n");
33  else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
34  else if(k == 0) printf("\nk aus n = 1\n\n");
35  else
36  {
37     dF = -0.5*log(2*M_PI)+(0.5+n)*log(n)-(0.5+k)*log(k)
38            -(n-k+0.5)*log(n-k)+(1/12)*(1/n-1/k-1/(n-k));
39     dF /= log(10);
40     ma = modf(dF, &ex);
41     ma = exp(ma*log(10));
42
43     gcvt(ma, iZiffern, szMa);
44     itoa(ex, szEx, 10);
45
46     printf("\nk aus n = %s E %s\n\n", szMa, szEx);
47  }
```

Als Erstes wird geprüft, ob tatsächlich *k* kleiner als *n* ist, andernfalls wird eine Fehlermeldung ausgegeben. Kann man das Ergebnis nicht sofort angeben, wird der Binomialkoeffizient nach obiger Formel berechnet, welche man in den Zeilen 37 bis 39 des Quellcodes wiedererkennt. Die darauffolgende Umwandlung und Ausgabe wurde bereits in der Beschreibung des Programms FAKSTIR erwähnt.

2.3.2 Logarithmische Berechnung

Binomialkoeffizienten kann man aber auch wie bei der logarithmischen Berechnung von Fakultäten beschrieben über den dekadischen Logarithmus näherungsweise Berechnen. Der Vorteil gegenüber der Sterlingschen Formel ist derselbe wie bei Fakultäten, nämlich die genauere Näherung. Natürlich bleibt noch immer der Nachteil der zeitaufwendigeren Berechnung.

Es gilt: $$\lg\binom{n}{k} = \frac{\lg n!}{\lg k! \cdot \lg(n-k)!} =$$

$$= \lg 1 + \lg 2 + \lg 3 + \ldots + \lg n - \lg 1 - \lg 2 - \lg 3 - \ldots$$

$$\ldots - \lg k - \lg 1 - \lg 2 - \lg 3 - \ldots - \lg(n-k) =$$

$$= \lg(k+1) + \ldots + \lg n - \lg 2 - \ldots - \lg(n-k)$$

Diesmal umfasst der passende Algorithmus also zwei Schleifen.

Das Programm BINOMLOG

Zunächst wieder ein Ausschnitt des Quellcodes zum Programm BINOMLOG:

```
30   if(k > n) printf("\nk muss kleiner als n sein!\n\n");
31   else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
32   else if(k == 0) printf("\nk aus n = 1\n\n");
33   else
34   {
35      printf("\nBitte warten...\n\n");
36      if(k < n-k) k = n-k;
37      dF = log10(k+1);
38      for(j = k+2; j < n+1; j++) dF += log10(j);
39      for(j = 2; j < n-k+1; j++) dF -= log10(j);
40      ma = modf(dF, &ex);
41      ma = exp(ma*log(10));
42
43      gcvt(ma, iZiffern, szMa);
44      itoa(ex, szEx, 10);
45
46      printf("\nk aus n = %s E %s\n\n", szMa, szEx);
47   }
```

Die beiden erwähnten Schleifen zur Berechnung des Binomialkoeffizienten findet man in Zeile 38 und 39 des Quellcodes. Zuvor jedoch wird in Zeile 36 eventuell *k* durch (*n*-*k*) ersetzt, um die Berechnung zu beschleunigen. Dies folgt aus dem Symmetriegesetz für Binomialkoeffizienten: $\binom{n}{k} = \binom{n}{n-k}$. Ansonsten hat sich zum Programm BINOMSTI nichts verändert.

3 Exakte Berechnung

3.1 Rechnen mit langen Zahlen

Da Fakultäten nicht mit den normalen Variablentypen, die maximal 4 Byte umfassen, darstellbar sind, deren Wertebereich sich nur von 0 bis 4 294 967 295 erstreckt, aber schon die Fakultät von 13 diesen Bereich überschreitet, ist es notwendig, diese großen Zahlen in kleinere Einheiten zu zerlegen und diese am Besten in einer Liste zu verwalten. Der Vorteil gegenüber einem Array liegt darin, dass meist die Länge der Zahl im Voraus nicht bekannt ist und somit nicht abschätzbar ist, wie viel Speicher für die Zahl reserviert werden muss. Mit einer Liste lässt sich der Speicher dynamisch verwalten und es wird nur soviel wie benötigt belegt.
Im Folgenden wird eine doppelt verkettete Liste verwendet, obwohl natürlich eine einfach verkettete Liste ausreichen würde. Hierfür benutze ich die von mir schon vor längerer Zeit programmierte **template** Klasse *gldlist*.

Die lange Zahl wird in jeweils drei Ziffern aufgespalten, diese Ziffernfolgen werden wir ab sofort als einen Block bzw. Dreierblock bezeichnen. Den Anfang der Liste bildet der niederwertigste Teil der Zahl. Ein Beispiel stellt die Tabelle 3.1 anschaulich dar. In ihr sind die Speicherinhalte der Listenelemente angegeben, wobei *Pos* die Position des Elements in der Liste bezeichnet. Die Position 0 kennzeichnet den Anfang der Liste.

Pos	5	4	3	2	1	0
Inhalt	12	154	850	1	45	0

Tabelle 3.1: Die Zahl 12 154 850 001 045 000

Die Multiplikation beinhaltet eine Addition, die Division eine Subtraktion von langen Zahlen, deshalb werden auch diese beiden Grundrechenarten kurz erläutert. Die Division wird später für die exakte Berechnung von Binomialkoeffizienten benötigt. Als Grundgedanke für alle vier Rechenarten dient das schriftliche Rechnen, wie es schon in der Grundschule erlernt wird. Alle vier Routinen rechnen mit natürlichen Zahlen und der Null. In der Klasse TLongInt wird auch noch das Vorzeichen der langen Zahl gespeichert, weshalb man dann mit ganzen Zahlen und der Null rechnen kann, auch wenn dies zur Berechnung von Fakultäten bzw. Binomialkoeffizienten nicht nötig ist. Da es aber keinen sonderlichen Aufwand darstellt und die Klasse TLongInt universell einsetzbaren Anspruch erhebt, wurde das Vorzeichen mit in die Klasse aufgenommen. So sind folgende Funktionen als betragsweise Rechenoperationen zu verstehen. Zu allen vier Rechenoperationen ist im Anhang jeweils ein Beispiel aufgeführt, das den Ablauf demonstrieren soll. Um die Beispiele besser nachvollziehen zu können, ist es ratsam das zugehörige Struktogramm zur Hand zu nehmen.

3.1.1 Addition

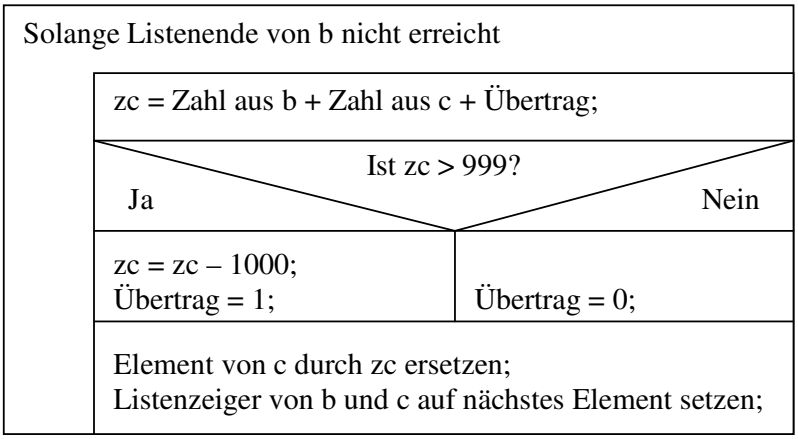
Die Addition ist die einfachste Rechenart. Wir betrachten natürliche Zahlen und die Null. Außerdem ist es für die Schleifenabbruchbedingung wichtig, dass *a* größer oder gleich *b* ist. Das Ergebnis der Addition ist eine natürliche Zahl *c* oder Null.

$$\text{Also:} \qquad a + b = c; \qquad a \geq b; \qquad (a, b, c \in N_0)$$

Um unnötige Schleifendurchgänge zu vermeiden, wird zunächst *c* gleich *a* gesetzt. Im folgenden Teilstruktogramm 1 der Addition sind alle benötigten Schritte aufgeführt, die zunächst durchgeführt werden müssen.

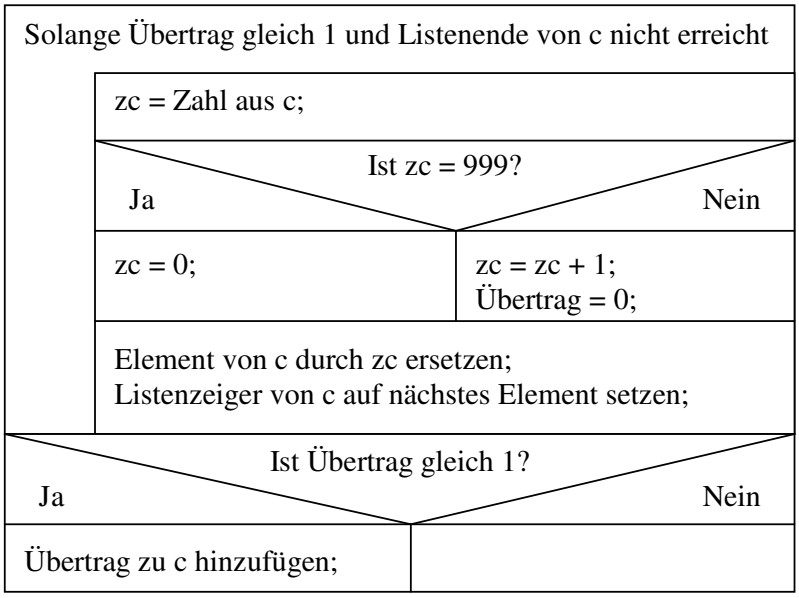
Teilstruktogramm 1 der Addition: Initialisierung

Man addiert nun in einer Schleife alle sich entsprechenden Blöcke von *b* und *c* miteinander, bis das Ende der kleineren Zahl *b* erreicht ist. Ist das Ergebnis der Addition der Zahl aus *b* und der entsprechenden aus *c* größer als 999, so wird von diesem 1000 abgezogen und der Übertrag gleich 1 gesetzt. Das aktuelle Element von *c* wird durch das Ergebnis ersetzt. Dies veranschaulicht das zweite Teilstruktogramm der Addition.



Teilstruktogramm 2 der Addition: Hauptschleife

Nun wird noch eine weitere Schleife so lange durchlaufen, bis kein Übertrag mehr vorhanden oder das Ende von *c* erreicht ist. Ist die Zahl, die im aktuellen Element gespeichert ist, gleich 999, so wird diese durch Null ersetzt und der Übertrag bleibt bestehen. Erst wenn diese Zahl mit eins addiert maximal drei Ziffern hat, wird der Übertrag gleich Null gesetzt und somit die Schleife beendet.
Wurde die Schleife allerdings beendet, weil das Ende von *c* erreicht war, so wird der übrig gebliebene Übertrag an das Ende von *c* angehängt, wie es auch Teil-struktogramm 3 der Addition entnommen werden kann.



Teilstruktogramm 3 der Addition: Übertragsschleife

3.1.2 Subtraktion

Wir gehen wieder davon aus, dass *a* größer oder gleich *b* ist. Das Ergebnis der Subtraktion sei *c*, wobei *a*, *b* und *c* wieder Elemente der Menge der natürlichen Zahlen mit der Null sind.

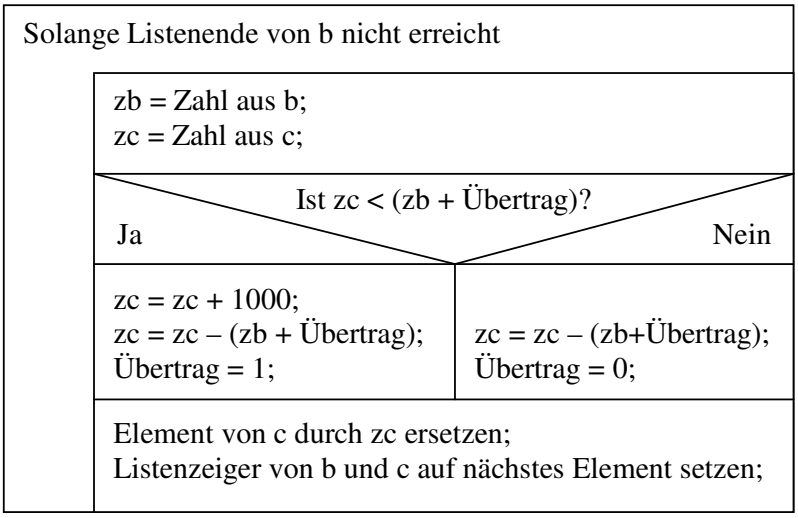
$$\text{Also:} \qquad a - b = c; \qquad a \geq b; \qquad (a, b, c \in N_0)$$

Die Subtraktion ist nicht sehr viel anders als die Addition zu handhaben. Nur dass diesmal ein Übertrag gesetzt wird, wenn vom nächst höheren Block 1000 „geborgt“ wird. Dies ist notwendig, wenn die Zahl des zu subtrahierenden Blocks größer ist als die, von der diese Zahl subtrahiert werden soll.

Wieder müssen zunächst alle Werte auf ihren Anfangswert gesetzt werden.

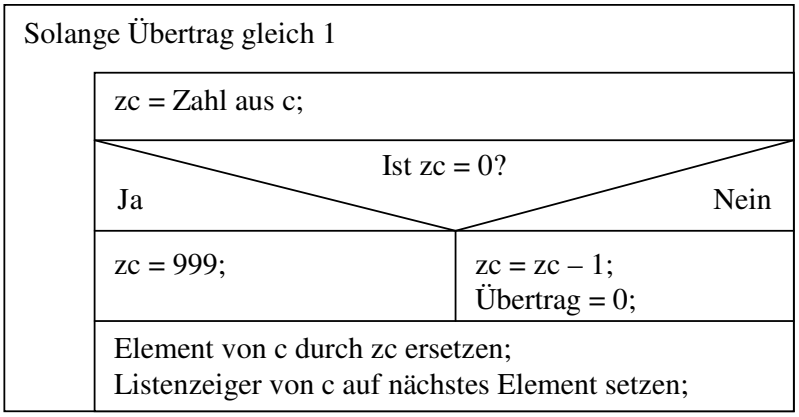
Teilstruktogramm 1 der Subtraktion: Initialisierung

Als Nächstes wird parallel zur Addition eine Schleife durchlaufen, bis das Ende der kleineren Zahl *b* erreicht ist. Innerhalb dieser Schleife werden die entsprechenden Blöcke unter Berücksichtigung des geborgten Übertrags voneinander subtrahiert (siehe Teilstruktogramm 2 der Subtraktion).



Teilstruktogramm 2 der Subtraktion: Hauptschleife

Zuletzt ist noch eine Schleife notwendig, die so lange den Übertrag subtrahiert, bis die Zahl, von der der Übertrag 1 subtrahiert werden soll größer als Null ist, so dass nichts vom höherwertigen Block geborgt werden muss. Dies ist der Fall noch bevor das Listenende von *c* erreicht wird, da $a \geq b$ ist. Deshalb taucht diese Abbruchbedingung im Vergleich zur Addition im Kopf der Schleife – in Teilstruktogramm 3 der Subtraktion dargestellt – nicht auf.



Teilstruktogramm 3 der Subtraktion: Übertragsschleife

3.1.3 Multiplikation

Auch die Multiplikation stellt kein größeres Problem dar, wenn man sich an die Methode des schriftlichen Multiplizierens erinnert. Diesmal spielt es auch keine Rolle, welcher der beiden Faktoren der Größere ist.

Also: $a \cdot b = c; \qquad a, b, c \in N_0$

Jeder Block aus *a* wird der Reihe nach mit jedem einzelnen Block aus *b* multipliziert. Dies wird mit einer Schleife innerhalb einer äußeren Schleife erledigt. Dabei wird jeweils ein Übertrag gebildet, der gleich dem Quotienten aus der Division des Ergebnisses der Multiplikation der entsprechenden Blöcke, zu dem der vorherige Übertrag addiert wurde, durch 1000 gesetzt wird. Der Rest dieser Division wird an das Ende einer Zwischenergebnisliste angehängt. Die innere Schleife ist in Teilstruktogramm 1 der Multiplikation dargestellt.

za = Zahl aus a; Listenzeiger von b auf Anfang setzen;
Solange Listenende von b nicht erreicht
zb = Zahl aus b; zc = za * zb + Übertrag; Übertrag = Quotient aus Division von zc durch 1000; zc = Rest der Division von zc durch 1000; zc in Zwischenergebnisliste einfügen; Listenzeiger von b auf nächstes Element setzen;

Teilstruktogramm 1 der Multiplikation: Innere Schleife

Nun muss noch der übrig gebliebene Übertrag in die Zwischenergebnisliste eingefügt werden und anschließend die benötigte Anzahl an Nullblöcken am Anfang der Zwischenergebnisliste angehängt werden, was einer Multiplikation mit 1000 entspricht. Jetzt addiert man dieses Zwischenergebnis zu *c* hinzu, löscht die Zwischenergebnisliste wieder für den nächsten Durchlauf der äußeren Schleife und setzt den Zeiger von *a* auf den nächsten Block, der nun ebenfalls wie gerade beschrieben mit *b* multipliziert wird. Durch Aufaddieren der Zwischenergebnisse erhält man so das Ergebnis *c* der Multiplikation der beiden langen natürlichen Zahlen *a* und *b*.

Ist Übertrag > 0?	
Ja	Nein
Übertrag in Zwischenergebnisliste einfügen; Übertrag = 0;	
Nullblöcke am Anfang der Zwischenergebnisliste einfügen; c = c + Zwischenergebnis; (Addition von langen Zahlen) Zwischenergebnisliste löschen; Listenzeiger von a auf nächstes Element setzen;	

Teilstruktogramm 2 der Multiplikation: Aufaddieren der Zwischenergebnisse

Wie die beiden Teilstruktogramme in die äußere Schleife integriert werden, kann man dem kompletten Struktogramm der Multiplikation, das im Anhang aufzufinden ist, entnehmen.

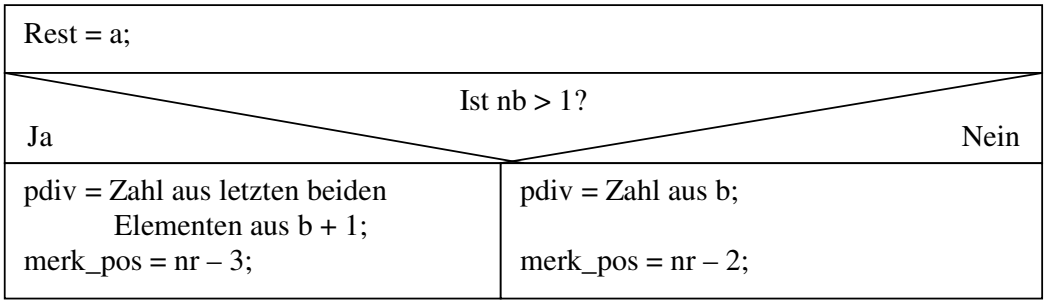
3.1.4 Division

Die am schwierigsten zu programmierende Rechenart ist die Division. Deshalb fällt der Quellcode auch relativ lang aus. Dementsprechend ist auch das Struktogramm etwas aufwendiger als die der anderen drei Rechenarten. Wiederum setzen wir voraus, dass *a* größer oder gleich *b* ist. Das Ergebnis ist diesmal nicht nur eine natürliche Zahl *c* oder Null, sondern auch ein Rest aus der Menge der natürlichen Zahlen mit Null.

$$\text{Also:} \qquad a : b = c + \frac{\text{Rest}}{b}; \qquad a \geq b; \qquad (a, b, c, \text{Rest} \in N_0)$$

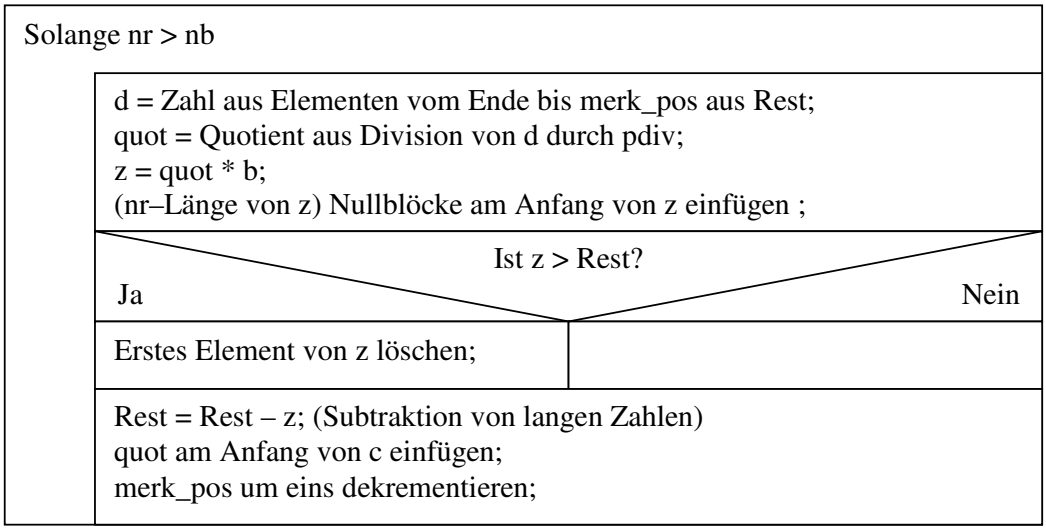
Als Erstes wird der Rest gleich *a* gesetzt. Daraufhin wird, je nachdem ob *b* mehr als ein Element enthält, *pdiv* aus den beiden höchstwertigen Dreierblöcken plus eins bzw. dem einzigen vorhandenen Dreierblock gebildet. Die Variable *pdiv* dient später dazu, den Quotienten zu ermitteln, der mit Sicherheit kleiner ist als der eigentliche Quotient, da zu *pdiv* eins addiert wurde. In der Variablen *merk_pos* wird schließlich die Position des Blocks in Rest gespeichert, bis zu der die Zahl ausgelesen wird, die später durch *pdiv* geteilt wird. Normalerweise ist das eine aus drei Blöcken zusammengesetzte Zahl. Die Variable *merk_pos* ist notwendig, um sicher zu stellen, dass immer nur ein Dreierblock „heruntergeholt“ wird (vgl. schriftliches Dividieren).

nr: Anzahl der Listenelemente des Rests
nb: Anzahl der Listenelemente von b;



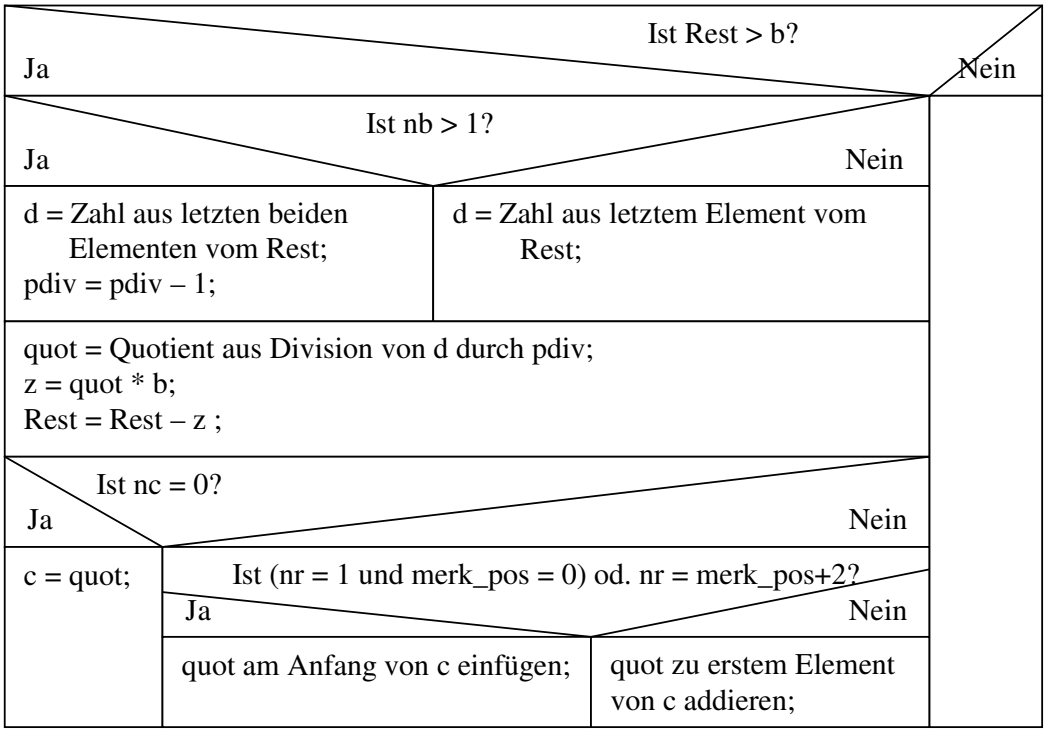
Teilstruktogramm 1 der Division: Initialisierung

Die Hauptschleife wird so lange ausgeführt, bis die Liste des Rests genauso viele bzw. weniger Elemente besitzt als der Divisor. Zunächst wird aus den Blöcken vom Ende des Rests bis zur Position *merk_pos* eine Zahl gebildet. Diese wird anschließend durch *pdiv* geteilt. Der Quotient aus dieser Division wird am Anfang der Ergebnisliste *c* eingefügt. Nun muss noch vom Rest das Ergebnis der Multiplikation des Quotienten mit dem Divisor unter Berücksichtigung der anzuhängenden Nullblöcke subtrahiert werden. Zuletzt wird *merk_pos* um eins nach „rechts“ verschoben, also um eins erniedrigt.



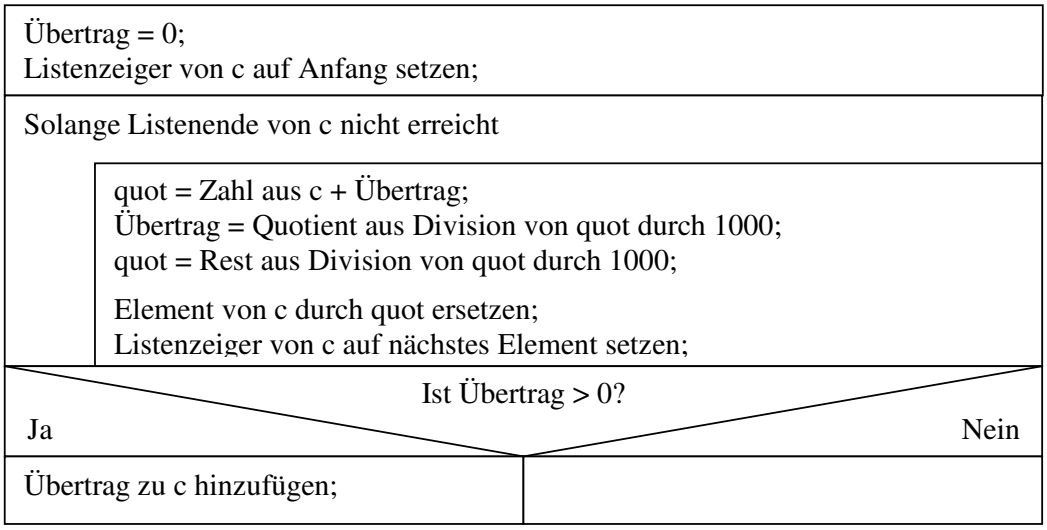
Teilstruktogramm 2 der Division: Hauptschleife

Anschließend muss überprüft werden, ob noch eine letzte Division möglich ist. Dies ist der Fall, wenn der Rest der Hauptschleife größer als der Divisor *b* ist. Bei dieser letzten Division ist es notwendig, dass die zur Ermittlung des Quotienten gebildete Zahl, die durch *pdiv* geteilt wird, aus der gleichen Anzahl an Elementen gebildet wird wie zuvor *pdiv*, ansonsten erhält man einen viel zu großen Quotienten. Der übrig gebliebene Quotient wird abhängig von der letzten Division zu dem zuletzt zu *c* hinzugefügten Quotienten addiert oder an den Anfang von *c* angehängt.



Teilstruktogramm 3 der Division: Abschließende Division

Da die ermittelten Quotienten auch mehr als drei Ziffern enthalten können, muss das Ergebnis noch in die vereinbarte Form gebracht werden. Hierfür wird die Ergebnisliste vom Anfang bis zum Ende durchgegangen und jeweils der Übertrag zum nächst höheren Block addiert.



Teilstruktogramm 4 der Division: Formatierung des Ergebnisses

So hat man nun den natürlichen Quotienten aus der Division von *a* durch *b* ermittelt und in der Liste von *c* gespeichert. Den natürlichen Rest der Division erhält man im selben Format.

3.1.5 Die Klasse TLongInt

Die Klasse TLongInt bietet eine Vielzahl an Funktionen zum Rechnen mit langen Zahlen aus der Menge der ganzen Zahlen mit Null (Z_0) bzw. zum Vergleichen solcher. Die Klasse speichert von der langen Zahl das Vorzeichen und eine doppelt verkettete Liste, die die Dreierblöcke enthält. Mit den jeweiligen Elementfunktionen bzw. überladenen Operatoren, kann man nun diese lange Zahl verändern (Bsp.: $a \mathrel{+}= b \Rightarrow a = a + b$). Möchte man als Ergebnis eine neue lange Zahl, so finden sich in der Header-Datei der Klasse TLongInt **longint.h** auch noch einzelne Funktionen, die das Ergebnis einer neuen langen Zahl zuordnen und diese als Rückgabewert zurückliefern (Bsp.: $c = a + b$). Beim Entwickeln der Klasse TLongInt spielte v. a. die Operatorenüberladung eine entscheidende Rolle. Hierdurch wird es dem Anwender der Klasse kaum bewusst, dass er nicht mehr mit normalen Zahlentypen sondern mit langen Zahlen rechnet. Wenn man sich den Quellcode der Header-Datei ansieht wird einem die Funktionsweise der Klasse leicht zugänglich sein und dem fröhlichen Rechnen mit langen Zahlen nichts mehr im Wege stehen. Die Implementation der Elementfunktionen der Klasse TLongInt sowie der anderen in der Header-Datei definierten Funktionen findet sich in der Datei **longint.cpp**.

Auf Grundlage der Klasse TLongInt ist es einfach das Rechnen mit langen Zahlen mit Dezimalstellen zu programmieren. Hierfür kann man entweder die Position des Kommas oder gleich jeweils eine Liste für den ganzzahligen Teil und eine weitere für die Nachkommastellen in die Klasse aufnehmen. Auch das Rechnen mit rationalen Zahlen ist möglich, wenn man den Zähler und den Nenner der Zahl speichert. Verwendet man für beide den Typ TLongInt, muss man lediglich nach jeder Neuberechnung des Zählers und des Nenners den Bruch kürzen. Hierin liegt die eigentliche Schwierigkeit: im Auffinden des größten gemeinsamen Teilers. In diesem Zusammenhang möchte ich auf das ebenfalls im Literaturverzeichnis erwähnte Buch *„Arithmetik“* von Donald E. Knuth verweisen. Dort wird in Kapitel 4.5.2 *„Der größte gemeinsame Teiler“* dieses Problem behandelt.

3.2 Fakultäten

Das Programm FAKEXAKT

Nachdem wir nun eine Klasse besitzen, die mit langen Zahlen rechnen kann, ist es keine Schwierigkeit mehr Fakultäten exakt zu berechnen. Dafür verwenden wir eine Funktion *fak()*, die sich selbst aufruft. Dieser Vorgang wird Rekursion genannt und bietet anstatt einer Schleife eine elegante Lösung zur Berechnung von Fakultäten. Bei jedem Aufruf von *fak()* wird das Argument des Parameters *l* um eins erniedrigt (Zeile 16).
Sobald *l* = 1 ist, wird die Folge der rekursiven Aufrufe beendet (Zeile 15).

```
09    TLongInt& fak(TLongInt& l)
10    {
11       TLongInt* e = new TLongInt;
12
13       if(l.empty()) return *e;
14       else if(l < 0) return *e;
15       else if((l == 0) || (l == 1)) *e = 1;
16       else *e = l * fak(l-1);
17
18       return *e;
19    }
```

Nun müssen wir zur Berechnung der Fakultät von *n* nur noch die Funktion *fak()* mit *n* als Argument im ausführenden Teil aufrufen. Zuvor wandeln wir den eingelesenen String noch in eine lange Zahl um. Das Ergebnis erhalten wir auf umgekehrtem Weg durch Aufruf der Elementfunktion *getChar()* der Klasse TLongInt.

```
36   n.setChar(szN);
37   f = fak(n);
38   result = f.getChar(iZiffern);
```

3.3 Binomialkoeffizienten

Das Programm BINOEXKT

Die exakte Berechnung von Binomialkoeffizienten erfolgt auf ähnliche Art wie bei Fakultäten. Nur dass diesmal die rekursive Funktion *fak()* um eine Abbruchbedingung erweitert wird. Wir möchten nämlich die Berechnung abschließen, sobald das Argument des Parameters *l* einen bestimmten Wert erreicht hat. So ersparen wir uns unnötige Rechenzeit, d. h. wir kürzen $n!$ im Zähler mit $(n\text{-}k)!$ im Nenner.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{(n-k+1)\cdot(n-k+2)\cdot\ldots\cdot(n-1)\cdot n}{k!}$$

Der Parameter *lastFak* gibt den Faktor an, der zuletzt mit in die Berechnung eingehen soll. Die Abbruchbedingung ist in Zeile 15 des Quellcodes integriert.

```
09    TLongInt& fak(TLongInt& l, TLongInt& lastFak)
10    {
11       TLongInt* e = new TLongInt;
12
13       if(l.empty()) return *e;
14       else if(l < 0) return *e;
15       else if((l == 0) || (l == 1) || (l == lastFak-1)) *e = 1;
16       else *e = l * fak(l-1, lastFak);
17
18       return *e;
19    }
```

Im ausführenden Teil des Programms BINOEXKT wird der Quotient aus Zähler und Nenner, die zuvor über die Funktion *fak()* berechnet wurden, in der Struktur *f* des Typs TDivData gespeichert. Diese Stuktur enthält zwei lange Zahlen, zum Einen den Quotienten und zum Anderen den Rest der Division, der bei der Berechnung von Binomialkoeffizienten Null ist.

```
41   if(k > n) printf("\nk muss kleiner als n sein!\n\n");
42   else if(k == 0) printf("\nk aus n = 1\n\n");
43   else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
44   else
45   {
46      f = DivLong(fak(n, n-k+1), fak(k, (TLongInt)(long)0));
47      result = f.quot.getChar(iZiffern);
48      printf("\nk aus n =\n");
49      if(result == NULL) printf("Fehler\n\n");
50      else
51      {
52         printf("%s\n\n", result);
53         delete result;
54      }
55   }
```

4 Win32-Programme

4.1 Das Programm CalcLongInt

Das Programm CalcLongInt diente in erster Linie dazu, die Funktion der Klasse TLongInt zu überprüfen und hat weiter für das Thema dieser Arbeit keine Bedeutung. Ich erwähne es trotzdem, um den interessierten Programmierern unter Ihnen die Möglichkeit zu geben den Umgang mit der Klasse TLongInt studieren zu können und deren Fähigkeiten auszuprobieren. Der Quellcode steht auf der beiliegenden CD zur Verfügung.

4.2 Das Programm FakBinomial

Eigentliches Herzstück dieser Arbeit bildet das Programm FakBinomial für Windows. Es vereinigt alle DOS-Programme in sich und stellt eine benutzerfreundliche und einfach zu handhabende Oberfläche zur exakten und näherungsweisen Berechnung von Fakultäten und Binomialkoeffizienten dar.
In der Gruppe Methode kann man auswählen, auf welche, in den beiden vorherigen Kapiteln beschriebene Art die Berechnung ausgeführt werden soll. Gleich darunter findet sich die Möglichkeit die geltenden Ziffern bzw. bei der exakten Berechnung die Anzahl der Ziffern pro Block anzugeben. Diese Formatierung lässt sich bei der exakten Berechnung auch noch hinterher mittels des Schalters Aktualisieren ändern, da eine erneute Berechnung meist zu viel Zeit beanspruchen würde. Über den Schalter Drucken kann man schließlich das Ergebnis der exakten Berechnung ausdrucken. Während jeder Berechnung wird in der Statusanzeige der Fortschritt der Berechnung angezeigt. Außerdem steht direkt darüber jeweils eine Information was gerade ausgeführt wird.
Ein wesentlicher Nachteil des Programms FakBinomial liegt darin, dass das ganze System für die Dauer der Berechnung blockiert wird. Dies macht sich bei sehr großen Fakultäten bzw. Binomialkoeffizienten bemerkbar, deren genaue Berechnung allerdings selten von Bedeutung sein dürfte. Um solch zeitaufwendige Berechnungen aber dennoch (v. a. auf langsameren Rechnern) durchführen zu können ohne das System unnötig zu blockieren, bietet sich die Möglichkeit den Timer von Windows zu verwenden. Auf die genauere Beschreibung dieser Programmiertechnik kann ich aber im Rahmen dieser Arbeit leider nicht eingehen und möchte statt dessen auf das bereits erwähnte Buch von Charles Petzold zur Windows-Programmierung verweisen. Dort findet sich in Kapitel 8 *„Der Timer“* alles nötige zu diesem Thema.

5 Anhang

5.1 Beweis der Sterlingschen Näherungsformel

(1) $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ | ln

$$\ln(n!) = \frac{1}{2}\ln(2\pi n) + n\ln n - n = (n+\frac{1}{2})\ln n - n + \ln(\sqrt{2\pi})$$

(2) $\ln(\sqrt{2\pi}) = d_n = \ln n! - (n+\frac{1}{2})\ln n + n$

$\Rightarrow$ wenn gilt: $\lim_{n\to\infty} d_n = \ln(\sqrt{2\pi})$

dann gilt: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Weiter ist:

$$d_n - d_{n+1} = \ln n! - \left(n+\frac{1}{2}\right)\ln n + n - \ln(n+1)! + \left(n+\frac{3}{2}\right)\ln(n+1) - (n+1) =$$

$$= \ln\left(\frac{n!}{(n+1)!}\right) - \left(n+\frac{1}{2}\right)\ln n + \left(n+\frac{3}{2}\right)\ln(n+1) - 1 =$$

$$= \ln\left(\frac{1}{n+1}\right) - n\ln n - \frac{1}{2}\ln n + n\ln(n+1) + \frac{3}{2}\ln(n+1) - 1 =$$

$$= \ln\left(\frac{1}{n+1}\right) - \left(n+\frac{1}{2}\right)\ln n + \left(n+\frac{3}{2}\right)\ln(n+1) - 1 =$$

$$= \left(n+\frac{1}{2}\right)\ln(n+1) - \left(n+\frac{1}{2}\right)\ln n - 1 =$$

$$= \left(n+\frac{1}{2}\right)\ln\left(\frac{n+1}{n}\right) - 1 \Rightarrow$$

$$d_n - d_{n+1} = (2n+1)\frac{1}{2}\ln\left(\frac{1+\frac{1}{2n+1}}{1-\frac{1}{2n+1}}\right) - 1$$

NR: $$\frac{1+\frac{1}{2n+1}}{1-\frac{1}{2n+1}}=\frac{\frac{2(n+1)}{2n+1}}{\frac{2n}{2n+1}}=\frac{n+1}{n}$$

Aus: (I) $$\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+... \qquad -1<x\le 1$$

(II) $$\ln(1-x)=-x-\frac{x^2}{2}-\frac{x^3}{3}-\frac{x^4}{4}-... \qquad -1\le x<1$$

Folgt: $$\ln\left(\frac{1+x}{1-x}\right)=2x+2\frac{x^3}{3}+2\frac{x^5}{5}+... \qquad -1<x<1$$

$$\frac{1}{2}\ln\left(\frac{1+x}{1-x}\right)=x+\frac{x^3}{3}+\frac{x^5}{5}+... \qquad -1<x<1$$

Also gilt, da $-1<\frac{1}{2n+1}<1$ $\qquad n\in N$

$$d_n-d_{n+1}=(2n+1)\left(\frac{1}{2n+1}+\frac{1}{3(2n+1)^3}+\frac{1}{5(2n+1)^5}+...\right)-1=$$

$$=\frac{1}{3(2n+1)^2}+\frac{1}{5(2n+1)^4}+...<\frac{1}{3(2n+1)^2}+\frac{1}{3(2n+1)^4}+...$$

NR: $$\frac{1}{3(2n+1)^2}+\frac{1}{3(2n+1)^4}+...=\frac{1}{3(2n+1)^2}\frac{1}{1-(2n+1)^{-2}}=$$

$$=\frac{1}{3(2n+1)^2}\frac{(2n+1)^2}{(2n+1)^2-1}=\frac{1}{3\left((2n+1)^2-1\right)}$$

Oder: $$\frac{1}{3(2n+1)^2}<d_n-d_{n+1}<\frac{1}{3\left((2n+1)^2-1\right)}$$

Nun ist: $$\frac{1}{3(2n+1)^2}>\frac{1}{12n+1}-\frac{1}{12(n+1)+1}$$

NR: $\dfrac{1}{3(2n+1)^2} > \dfrac{12}{(12n+1)(12n+13)}$

$144n^2 + 168n + 13 > 144n^2 + 144n + 36$

$24n > 23 \qquad n \in N$

$$\frac{1}{12n} - \frac{1}{12(n+1)} = \frac{n+1-n}{12(n^2+n)} = \frac{1}{3(4n^2+4n+1-1)} = \frac{1}{3((2n+1)^2-1)}$$

Also: $\dfrac{1}{12n+1} - \dfrac{1}{12(n+1)+1} < d_n - d_{n+1} < \dfrac{1}{12n} - \dfrac{1}{12(n+1)}$

Oder: $\dfrac{1}{12n+1} - d_n - \dfrac{1}{12(n+1)+1} + d_{n+1} < 0$

$$0 < \frac{1}{12n} - d_n - \frac{1}{12(n+1)} + d_{n+1}$$

$$\left(d_n - \frac{1}{12n}\right) - \left(d_{n+1} - \frac{1}{12(n+1)}\right) < 0$$

$$0 < \left(d_n - \frac{1}{12n+1}\right) - \left(d_{n+1} - \frac{1}{12(n+1)+1}\right)$$

Daraus folgt: (III) $d_n - \dfrac{1}{12n+1}$ ist monoton fallend

(IV) $d_n - \dfrac{1}{12n}$ ist monoton steigend

Beide Folgen haben den gleichen Grenzwert $c = \lim\limits_{n\to\infty} d_n$

$\Rightarrow$ (3) $\quad d_n - \dfrac{1}{12n} < c < d_n - \dfrac{1}{12n+1}$

Oder:

(4) $\quad c + \dfrac{1}{12n+1} < d_n < c + \dfrac{1}{12n}$

n	$d_n - \frac{1}{12n}$	$d_n - \frac{1}{12n+1}$
1	0,917	0,923076923
5	0,918916558	0,919189782
10	0,918935763	0,919004634
20	0,918938187	0,918955476
50	0,918938511	0,918941284
80	0,918938533	0,918939616

Tabelle 5.1

Tabelle 5.1 gibt nach (3) Intervalle an, die *c* enthalten. Vergleicht man sie mit $\ln\sqrt{2\pi} = 0{,}918938533$, so haben wir bereits näherungsweise gezeigt, dass $e^c = \sqrt{2\pi}$ bzw. $\lim_{n\to\infty} d_n = \ln\sqrt{2\pi}$ gilt.

Exakter Beweis:

Aus (2) folgt: $e^{d_n} = n!\left(\frac{n}{e}\right)^{-n} n^{-\frac{1}{2}}$

Aus (4) folgt: $e^{c+\frac{1}{12n+1}} < n!\left(\frac{n}{e}\right)^{-n} n^{-\frac{1}{2}} < e^{c+\frac{1}{12n}}$

(5) $e^{\frac{1}{12n+1}} e^c \left(\frac{n}{e}\right)^n \sqrt{n} < n! < e^{\frac{1}{12n}} e^c \left(\frac{n}{e}\right)^n \sqrt{n}$

Aus $0 < x < \frac{\pi}{2}$ folgt $0 < \sin x < 1$

$\Rightarrow$ (6) $\sin^{2n-1} x > \sin^{2n} x > \sin^{2n+1} x$

Es sei $J_n = \int_0^{\frac{\pi}{2}} \sin^n x dx$

$$u(x) = \sin^{n-1} x; \qquad u'(x) = (n-1)\sin^{n-2} x \cos x$$
$$v'(x) = \sin x; \qquad v(x) = -\cos x$$

$$\Rightarrow J_n = \left[-\cos x \sin^{n-1} x\right]_0^{\frac{\pi}{2}} + (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2} x \cos^2 x dx =$$

$$= (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2} x(1-\sin^2 x)dx =$$

$$= (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2} x dx - (n-1)\int_0^{\frac{\pi}{2}} \sin^n x dx$$

Also gilt: $$J_n = (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2} x dx - (n-1)J_n$$

$$(1+n-1)J_n = (n-1)J_{n-2}$$

$$\Rightarrow (7)\ J_n = \frac{(n-1)}{n} J_{n-2}$$

Aus (7) und $J_0 = \frac{\pi}{2}, J_1 = 1$ folgt:

$$J_{2n} = \frac{\pi}{2}\frac{1}{2}\frac{3}{4}\frac{5}{6}\ldots\frac{2n-1}{2n}$$

$$J_{2n+1} = 1\frac{2}{3}\frac{4}{5}\frac{6}{7}\ldots\frac{2n}{2n+1}$$

Aus (6) und (7) folgt:

$$J_{2n-1} > J_{2n} > J_{2n+1} \Rightarrow \frac{J_{2n-1}}{J_{2n+1}} > \frac{J_{2n}}{J_{2n+1}} > 1$$

Eingesetzt: $$\frac{2n+1}{2n} > \frac{\pi}{2}\frac{1\cdot 3\cdot 3\cdot 5\cdot 5\cdot \ldots\cdot (2n-1)(2n+1)}{2\cdot 2\cdot 4\cdot 4\cdot 6\cdot 6\cdot \ldots\cdot 2n\cdot 2n} > 1$$

$$\Rightarrow \lim_{n\to\infty}\frac{1\cdot 3\cdot 3\cdot 5\cdot 5\cdot \ldots\cdot (2n-1)(2n+1)}{2\cdot 2\cdot 4\cdot 4\cdot 6\cdot 6\cdot \ldots\cdot 2n\cdot 2n}\frac{\pi}{2} = 1$$

Oder: $$\frac{\pi}{2} = \lim_{n\to\infty}\left(\frac{2\cdot 4\cdot 6\cdot \ldots\cdot 2n}{3\cdot 5\cdot \ldots\cdot (2n-1)}\right)^2 \frac{1}{2n+1}$$

(8) $$\frac{\pi}{2} = \lim_{n\to\infty}\frac{2^{4n}(n!)^4}{((2n)!)^2(2n+1)}$$

Nach (5) ist: $$n! = e^c n^n e^{-n} e^{\frac{1}{12n}}\sqrt{n}$$

$$(2n)! = e^c (2n)^{2n} e^{-2n} e^{\frac{1}{24n}}\sqrt{2n}$$

In (8): $$\frac{\pi}{2} = \lim_{n\to\infty}\frac{2^{4n}\left(e^c n^n e^{-n} e^{\frac{1}{12n}}\sqrt{n}\right)^4}{\left(e^c (2n)^{2n} e^{-2n} e^{\frac{1}{24n}}\sqrt{2n}\right)^2 (2n+1)} =$$

$$= \lim_{n\to\infty}\frac{2^{4n} e^{4c} n^2 n^{4n} e^{-4n} e^{\frac{1}{3n}}}{e^{2c} 2^{4n} n^{4n} e^{-4n} e^{\frac{1}{12n}} 2n\cdot(2n+1)} = \lim_{n\to\infty}\frac{e^{2c} e^{\frac{1}{4n}} n}{4n+2} =$$

$$\underset{l.H.}{=} \lim_{n\to\infty}\left[\frac{e^{2c}}{4}\left(1-\frac{1}{4n}\right)e^{\frac{1}{4n}}\right] = \frac{e^{2c}}{4}$$

$$\Rightarrow e^c = \sqrt{2\pi} \text{ oder } \lim_{n\to\infty} d_n = \ln\sqrt{2\pi}$$ qed.

Damit wäre also gezeigt, dass $\lim_{n\to\infty} d_n = \ln\sqrt{2\pi}$ gilt. Daraus folgt, dass

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

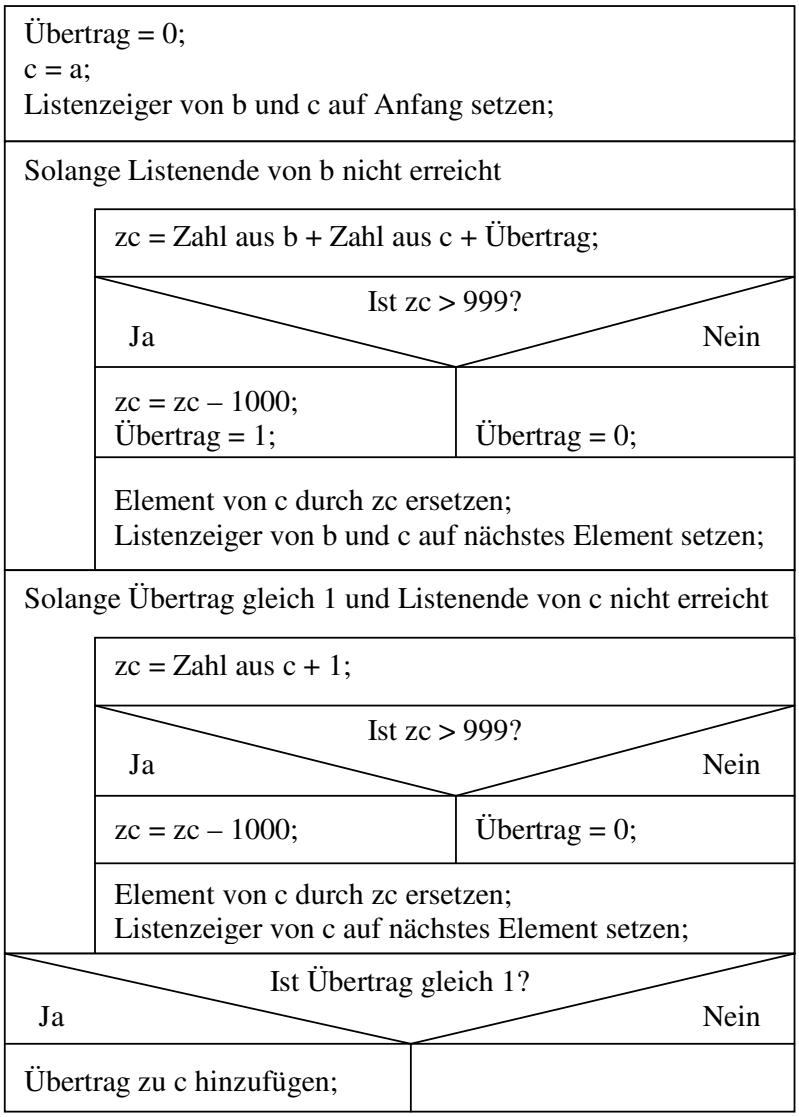
gilt. Aus Tabelle 5.1 geht hervor, dass die rechte Seite von (5) eine besonders gute Näherung für $n!$ liefert.

Also: $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

5.2 Rechnen mit langen Zahlen

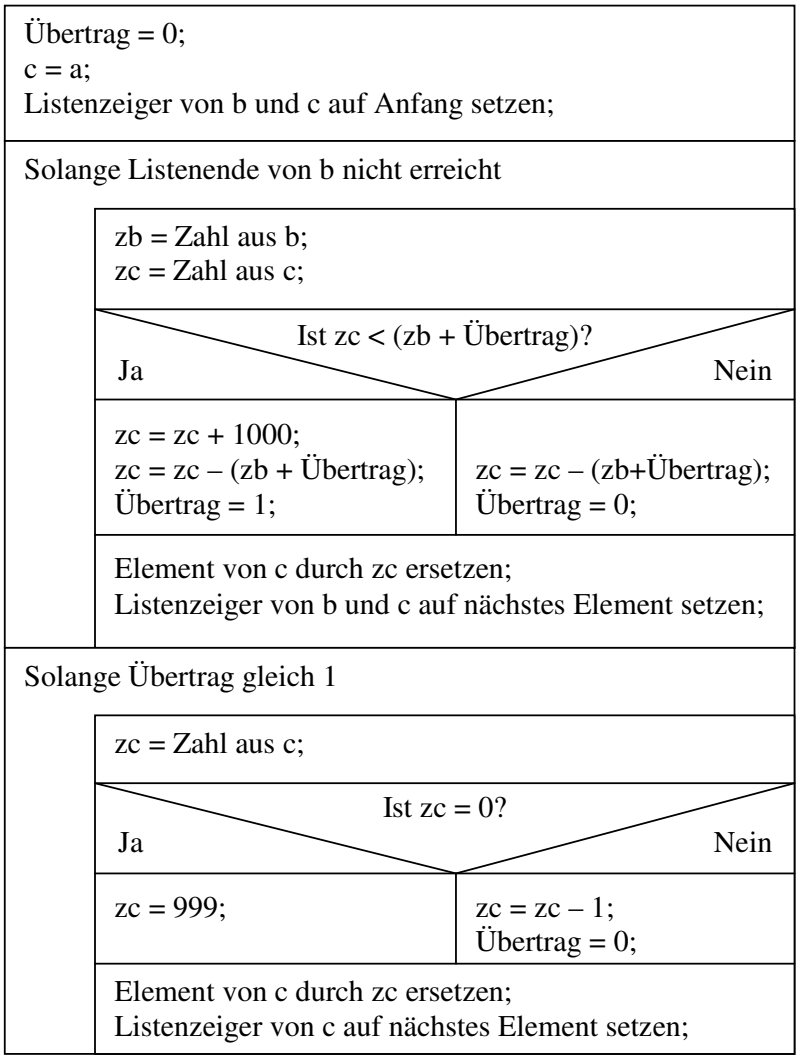
5.2.1 Struktogramme

Addition



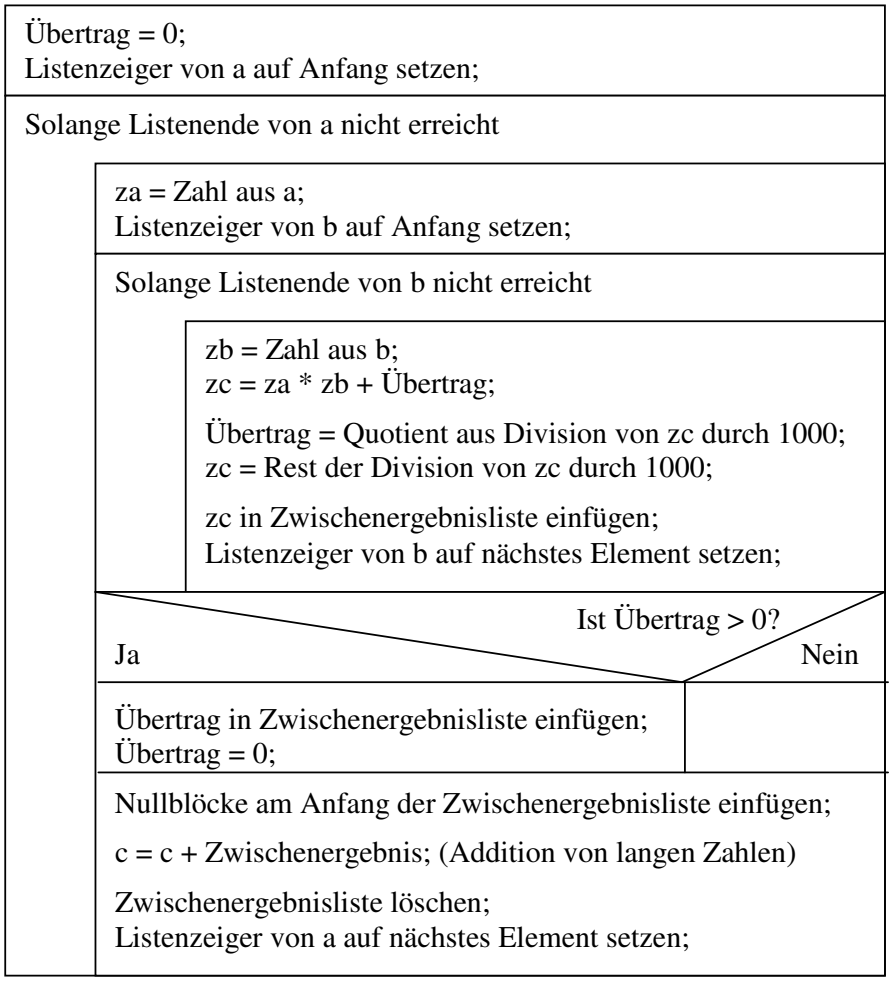
Struktogramm der Addition von langen Zahlen

Subtraktion



Struktogramm der Subtraktion von langen Zahlen

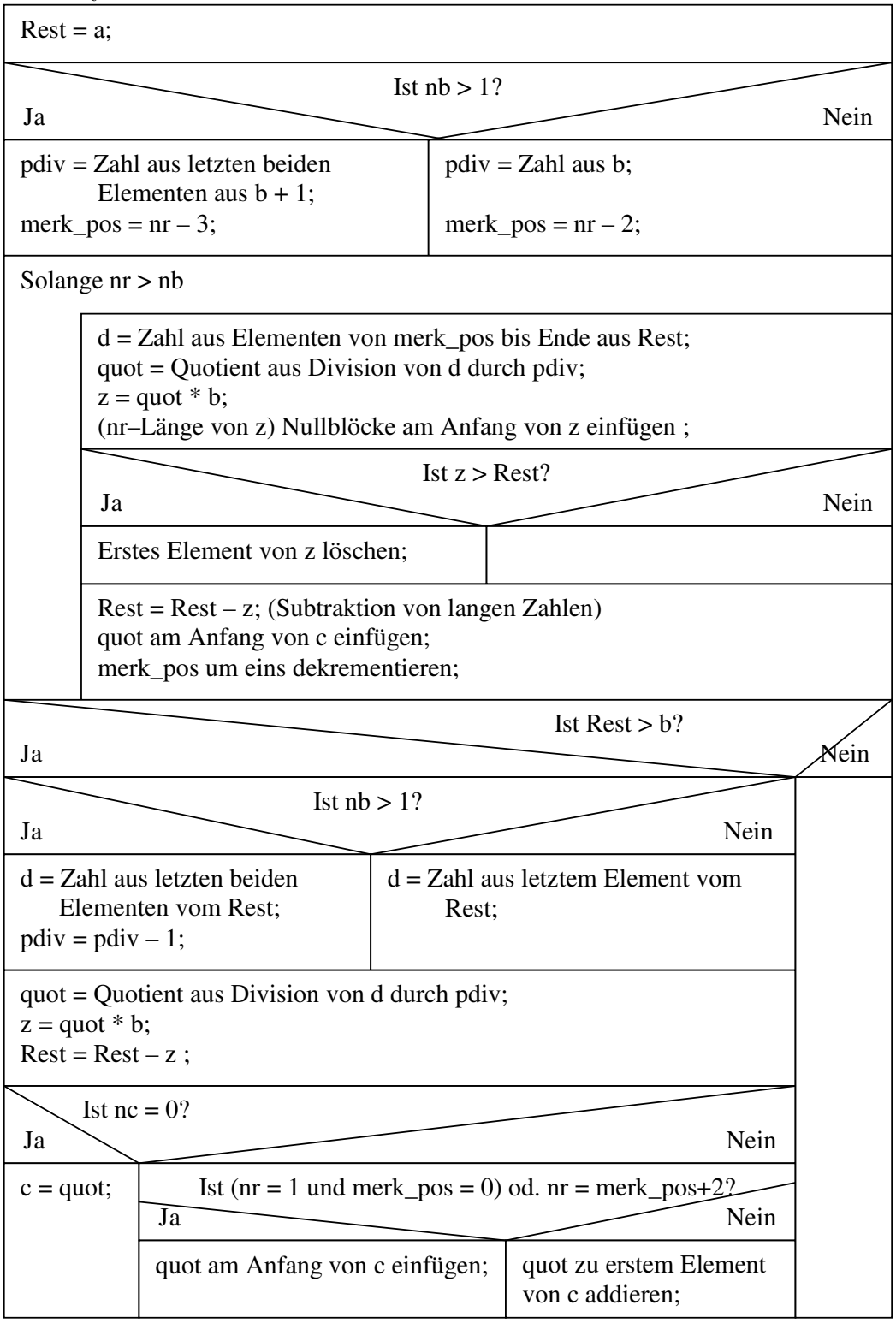
Multiplikation



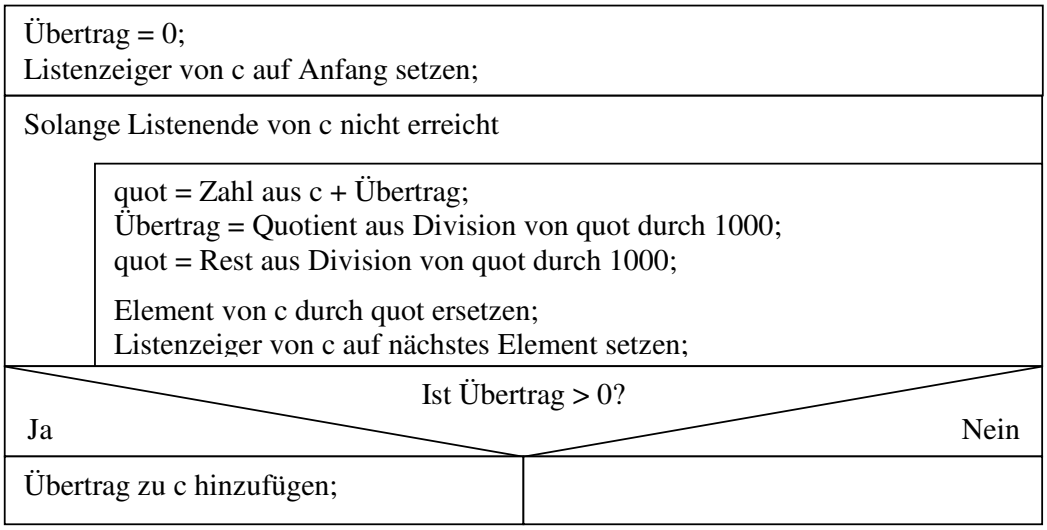
Struktogramm der Multiplikation von langen Zahlen

Division

nr: Anzahl der Listenelement des Rests
nb: Anzahl der Listenelemente von b



Struktogramm der Division von langen Zahlen – Teil I



Struktogramm der Division von langen Zahlen – Teil II

5.2.2 Beispiele

Addition

3 999 333 520 899 020 + 700 078 150 745 = 4 000 033 599 049 765

Pos	**5**	**4**	**3**	**2**	**1**	**0**
a	3	999	333	520	899	20
b **+**			700	78	150	745
Übertrag	1	1	0	1	0	0
c	4	0	33	599	49	765

Subtraktion

302 000 567 032 002 576 – 600 011 810 325 = 301 999 967 020 192 251

Pos	**5**	**4**	**3**	**2**	**1**	**0**
a	302	0	567	32	2	576
b **-**			600	11	810	325
Übertrag	1	1	0	1	0	0
c	301	999	967	20	192	251

Multiplikation

30 000 401 057 810 020 * 972 100 = 29 163 389 868 297 120 442 000

Pos	5	4	3	2	1	0
a					972	100
b	30	0	401	57	810	20

Pos	7	6	5	4	3	2	1	0
Übertrag	0	3	0	40	5	81	2	0
100 * *b*		3	0	40	105	781	2	0
Übertrag	29	0	389	56	787	19	0	
972 000 * *b*	29	160	389	828	191	339	440	**0**
+								
Übertrag	0	0	0	0	1	0	0	0
c	29	163	389	868	297	120	442	0

0 eingefügte Nullblöcke

Division

35 171 023 510 039 000 : 81 752 361 = 430 214 162 Rest: 30 902 518

pdiv = 81 752 + 1 = 81 753

Pos	5	4	3	2	1	0
Rest = *a*	35	171	23*	510	39	0
z	35	153	515	230	**0**	**0**
Rest		17	508	280*	39	0
z		17	495	5	254	**0**
Rest			13	274	785*	0
z			13	243	882	482
Rest				30	902	518*

* *merk_pos*
0 eingefügte Nullblöcke
□ *d*

35 171 023 : 81 753 = 430 ⇒ Anfang von *c*
17 508 280 : 81 753 = 214 ⇒ Anfang von *c*
13 274 785 : 81 753 = 162 ⇒ Anfang von *c*

Pos	2	1	0
c	430	214	162
Rest	30	902	518

5.3 Quellcodes

5.3.1 Das Programm FAKSTIR

```
/* fakstir.c      Näherungsweise Berechnung von Fakultäten
                          mit der Sterlingschen Formel
   Autor: Schuh Andreas
   Datum: 10.01.2003                                          */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
   char szN[11], szZiffern[3], szMa[21], szEx[11], szKey[2];
   int iZiffern;
   unsigned long n;
   double dF, ma, ex;
   int bWeiter = 1;

   printf("\n\n\tNAEHERUNGSWEISE BERECHNUNG VON FAKULTAETEN\n");
   printf("\t\tMIT DER STIRLINGSCHEN FORMEL\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Geltende Ziffern (0 bis 19): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      if(iZiffern > 19) iZiffern = 19;
      n = strtoul(szN, NULL, 10);
      if((n == 0) || (n == 1)) printf("\nn! = 1");
      else
      {
         dF = 0.5*log(2*M_PI)+(n+0.5)*log(n)-n+1/(12*n);
         dF /= log(10);
         ma = modf(dF, &ex);
         ma = exp(ma*log(10));

         gcvt(ma, iZiffern, szMa);
         itoa(ex, szEx, 10);

         printf("\nn! = %s E %s", szMa, szEx);
      }
      printf("\n\nNoch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.2 Das Programm FAKLOG

```
/* faklog.c Logarithmische Berechnung von Fakultäten
   Autor: Schuh Andreas
   Datum: 10.01.2003                                                */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
   char szN[11], szZiffern[3], szMa[21], szEx[11], szKey[2];
   int iZiffern;
   unsigned long n, j;
   double dF, ma, ex;
   int bWeiter = 1;

   printf("\n\n\tLOGARITHMISCHE BERECHNUNG VON FAKULTAETEN\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Geltende Ziffern (0 bis 19): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      if(iZiffern > 19) iZiffern = 19;
      n = strtoul(szN, NULL, 10);
      if((n == 0) || (n == 1)) printf("\nn! = 1");
      else
      {
         dF = log10(2);
         for(j = 3; j < n+1; j++) dF += log10(j);
         ma = modf(dF, &ex);
         ma = exp(ma*log(10));

         gcvt(ma, iZiffern, szMa);
         itoa(ex, szEx, 10);

         printf("\nn! = %s E %s", szMa, szEx);
      }
      printf("\n\nNoch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.3 Das Programm BINOMSTI

```
/* binomsti.c     Näherungsweise Berechnung von Binomialkoeffizienten
                          mit der Stirlingschen Formel
   Autor: Schuh Andreas
   Datum: 10.01.2003                                                  */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
   char szN[11], szK[11], szZiffern[3], szMa[21], szEx[11], szKey[2];
   int iZiffern;
   unsigned long n, k, j;
   double dF, ma, ex;
   int bWeiter = 1;

   printf("\n\n\tNAEHERUNGSWEISE BERECHNUNG VON
                                                 BINOMIALKOEFFIZIENTEN\n");
   printf("\t\tMIT DER STIRLINGSCHEN FORMEL\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Bitte geben Sie k ein (0 bis 4294967295): ");
      gets(szK);
      printf("Geltende Ziffern (0 bis 19): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      if(iZiffern > 19) iZiffern = 19;
      n = strtoul(szN, NULL, 10);
      k = strtoul(szK, NULL, 10);
      if(k > n) printf("\nk muss kleiner als n sein!\n\n");
      else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
      else if(k == 0) printf("\nk aus n = 1\n\n");
      else
      {
         dF = -0.5*log(2*M_PI)+(0.5+n)*log(n)-(0.5+k)*log(k)
                        -(n-k+0.5)*log(n-k)+(1/12)*(1/n-1/k-1/(n-k));
         dF /= log(10);
         ma = modf(dF, &ex);
         ma = exp(ma*log(10));

         gcvt(ma, iZiffern, szMa);
         itoa(ex, szEx, 10);

         printf("\nk aus n = %s E %s\n\n", szMa, szEx);
      }
      printf("Noch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.4 Das Programm BINOMLOG

```
/* binomlog.c     Logarithmische Berechnung von Binomialkoeffizienten
   Autor: Schuh Andreas
   Datum: 10.01.2003                                                  */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
   char szN[11], szK[11], szZiffern[3], szMa[21], szEx[11], szKey[2];
   int iZiffern;
   unsigned long n, k, j;
   double dF, ma, ex;
   int bWeiter = 1;

   printf("\n\n\tLOGARITHMISCHE BERECHNUNG VON
                                              BINOMIALKOEFFIZIENTEN\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Bitte geben Sie k ein (0 bis 4294967295): ");
      gets(szK);
      printf("Geltende Ziffern (0 bis 19): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      if(iZiffern > 19) iZiffern = 19;
      n = strtoul(szN, NULL, 10);
      k = strtoul(szK, NULL, 10);
      if(k > n) printf("\nk muss kleiner als n sein!\n\n");
      else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
      else if(k == 0) printf("\nk aus n = 1\n\n");
      else
      {
         printf("\nBitte warten...\n\n");
         if(k < n-k) k = n-k;
         dF = log10(k+1);
         for(j = k+2; j < n+1; j++) dF += log10(j);
         for(j = 2; j < n-k+1; j++) dF -= log10(j);
         ma = modf(dF, &ex);
         ma = exp(ma*log(10));

         gcvt(ma, iZiffern, szMa);
         itoa(ex, szEx, 10);

         printf("k aus n = %s E %s\n\n", szMa, szEx);
      }
      printf("Noch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.5 Das Programm FAKEXAKT

```
/* fakexakt.c     Exakte Berechnung von Fakultäten
   Autor: Schuh Andreas
   Datum: 10.01.2003                                                */

#include <stdlib.h>
#include <mathe\longint.h>

TLongInt& fak(TLongInt& l)
{
   TLongInt* e = new TLongInt;

   if(l.empty()) return *e;
   else if(l < 0) return *e;
   else if((l == 0) || (l == 1)) *e = 1;
   else *e = l * fak(l-1);

   return *e;
}

int main()
{
   char szN[11], szZiffern[7], szKey[2];
   char* result;
   int iZiffern, bWeiter = 1;
   TLongInt n, f;

   printf("\n\n\tEXAKTE BERECHNUNG VON FAKULTAETEN\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Anzahl der Ziffern pro Block (0 bis 999999): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      n.setChar(szN);
      f = fak(n);
      result = f.getChar(iZiffern);
      printf("\nn! =\n");
      if(result == NULL) printf("Fehler\n\n");
      else
      {
         printf("%s\n\n", result);
         delete result;
      }
      printf("Noch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.6 Das Programm BINOEXKT

```
/* binoexkt.c     Exakte Berechnung von Binomialkoeffizienten
   Autor: Schuh Andreas
   Datum: 10.01.2003                                          */

#include <stdlib.h>
#include <mathe\longint.h>

TLongInt& fak(TLongInt& l, TLongInt& lastFak)
{
   TLongInt* e = new TLongInt;
   if(l.empty()) return *e;
   else if(l < 0) return *e;
   else if((l == 0) || (l == 1) || (l == lastFak-1)) *e = 1;
   else *e = l * fak(l-1, lastFak);
   return *e;
}

int main()
{
   char szN[11], szK[11], szZiffern[7], szKey[2];
   char* result;
   int iZiffern, bWeiter = 1;
   TLongInt n, k;
   TDivData f;

   printf("\n\n\tEXAKTE BERECHNUNG VON BINOMIALKOEFFIZIENTEN\n");
   while(bWeiter != 0)
   {
      printf("\nBitte geben Sie n ein (0 bis 4294967295): ");
      gets(szN);
      printf("Bitte geben Sie k ein (0 bis 4294967295): ");
      gets(szK);
      printf("Anzahl der Ziffern pro Block (0 bis 999999): ");
      gets(szZiffern);
      iZiffern = atoi(szZiffern);
      n.setChar(szN);
      k.setChar(szK);
      if(k > n) printf("\nk muss kleiner als n sein!\n\n");
      else if(k == 0) printf("\nk aus n = 1\n\n");
      else if((k == 1) || (k == n)) printf("\nk aus n = %s\n\n", szN);
      else
      {
         f = DivLong(fak(n, n-k+1), fak(k, (TLongInt)(long)0));
         result = f.quot.getChar(iZiffern);
         printf("\nk aus n =\n");
         if(result == NULL) printf("Fehler\n\n");
         else
         {
            printf("%s\n\n", result);
            delete result;
         }
      }
      printf("Noch eine Berechnung (j, n)? ");
      gets(szKey);
      if(szKey[0] == 'n') bWeiter = 0;
      else szKey[0] = 'j';
   };
   return 0;
}
```

5.3.7 Grundrechenfunktionen der Klasse TLongInt

Addition

```
void TLongInt::AddiereBetrag(TLongInt& l)
{
   if(l.empty()) return;
   else if(list.empty())
   {
      *this = l;
      return;
   }

   if(KleinerB(l))
   {
      TLongInt e;

      e = l;
      e.AddiereBetrag(*this);
      *this = e;
   }
   else
   {
      int ue = 0; // Übertrag
      int z;

      while((l.getItemPos(-1) == 0) && (l.getNumberOfItems() > 1))
                                                          l.erasePos(-1);
      list.setActualPos(0);
      l.setActualPos(0);
      while(!l.isNULL())
      {
         z = *list.getItemActual() + l.getItemActual() + ue;
         if(z > 999)
         {
            z -= 1000;
            ue = 1;
         }
         else ue = 0;
         list.replaceActual(&z);
         list.next();
         l.next();
      }
      while((ue == 1) && !list.isNULL())
      {
         z = *list.getItemActual();
         if(z == 999)
         {
            z = 0;
         }
         else
         {
            z += 1;
            ue = 0;
         }
         list.replaceActual(&z);
         list.next();
      }
      if(ue == 1) list.insert(0, &ue);
   }
}
```

Subtraktion

```
void TLongInt::BildeDifferenzBetrag(TLongInt& l)
{
   if(l.empty()) return;
   else if(list.empty())
   {
      *this = l;
      return;
   }
   TLongInt e;
   if(KleinerB(l))
   {
      e = l;
      e.BildeDifferenzBetrag(*this);
      *this = e;
   }
   else
   {
      int ue = 0; // Übertrag
      int z, zl;

      e = *this;
      while((l.getItemPos(-1) == 0) && (l.getNumberOfItems() > 1))
                                                          l.erasePos(-1);
      e.setActualPos(0);
      l.setActualPos(0);
      while(!l.isNULL())
      {
         z = e.getItemActual();
         zl = l.getItemActual();
         if(z < (zl + ue))
         {
            z += 1000 - zl - ue;
            ue = 1;
         }
         else
         {
            z -= zl + ue;
            ue = 0;
         }
         e.replaceActual(z);
         e.next();
         l.next();
      }
      while(ue == 1)
      {
         z = e.getItemActual();
         if(z == 0) z = 999;
         else
         {
            z--;
            ue = 0;
         }
         e.replaceActual(z);
         e.next();
      }
   }
   while((e.getItemPos(-1) == 0) && (e.getNumberOfItems() > 1))
                                                          e.erasePos(-1);
   *this = e;
}
```

Multiplikation

```
void TLongInt::operator *= (TLongInt& l)
{
   if(l.empty() || list.empty()) return;

   TLongInt e;
   TLongInt zw; // Zur Zwischenspeicherung einer langen Zahl
   int za, zb; // Zahlen aus a und b
   int ze; // Zwischenergebnis der einfachen Multiplikation
   int ue = 0; // Übertrag
   int null = 0; // Zum Anhängen der Nullen

   if((*this == 0) || (l == 0))
   {
      *this = 0;
      return;
   }
   list.setActualPos(0);
   while(!list.isNULL())
   {
      za = *list.getItemActual();
      l.setActualPos(0);
      while(!l.isNULL())
      {
         zb = l.getItemActual();
         ze = za * zb + ue;

         ue = ze - ze%1000;
         ze -= ue;
         ue /= 1000;

         zw.insert(ze);
         l.next();
      }

      if(ue > 0)
      {
         zw.insert(ue);
         ue = 0;
      }

      for(long k = 0; k < list.getPosActual(); k++) // Anhängen der
                                                          Nullblöcke
         zw.insertPos(0, null);
      e += zw;
      zw.destroy();
      list.next();
   }

   // Vorzeichen
   if(((vz == 1) && (l.getVZ() == -1)) || ((vz == -1) &&
                                                (l.getVZ() == 1)))
      e.setVZ(-1);

   *this = e;
}
```

Division

```
TDivData& DivLong(TLongInt& a, TLongInt& b)
{
   TDivData* e = new TDivData;
   ldiv_t q;
   long d, pdiv, merk_pos;
   unsigned long ndiv, nr, nz;
   TLongInt div, div2, z;
   int ue;

   if(a.empty() || b.empty()) return *e;
   else if(b == 0) return *e;
   else if(a.getLong(d) && b.getLong(pdiv))
   {  // Einfache Division zweier Zahlen des Typs long
      q = ldiv(d, pdiv);
      e->quot.setLong(q.quot);
      e->rem.setLong(q.rem);
      return *e;
   }
   e->rem = a;       // Betrag von a in rem
   e->rem.setVZ(1);
   div = b;          // Betrag von b in div
   div.setVZ(1);
   ndiv = div.getNumberOfItems();

   if(e->rem < div) // Dividend größer als Divisor
   {
      e->quot = (long)0;
      return *e;
   }

   nr = e->rem.getNumberOfItems();
   div.setActualPos(-1);
   pdiv = div.getItemActual(); // pdiv erhält führenden zwei Blöcke
                                                          von Divisor
   if(ndiv > 1)                // plus 1, wenn Länge größer
                                                    (für Probedivision)
   {
      pdiv *= 1000;
      div.prior();
      pdiv += div.getItemActual() + 1;
      merk_pos = nr - 3;
   }
   else merk_pos = nr - 2;

   while(nr > ndiv) // so lange Rest länger (größer) als Divisor
   {
      e->rem.setActualPos(merk_pos);
      d = e->rem.getItemActual();
      e->rem.next();
      if(!e->rem.isNULL())
      {
         d += e->rem.getItemActual()*1000;
         e->rem.next();
         if(!e->rem.isNULL()) d += e->rem.getItemActual()*1000000;
      }

      q = ldiv(d, pdiv); // Probedivision, Quotient sicher kleiner als
                                                          tatsächlich,
      z = div;           // da pdiv um 1 erhöht wurde
      z *= q.quot;
```

```
      nz = z.getNumberOfItems();
      for(long i = 0; i < (nr - nz); i++) z.insertPos(0, 0);//Anhängen
                                                                der Nullen
      if(z > e->rem) z.erasePos(0);
      e->rem -= z;
      e->quot.insertPos(0, q.quot);
      nr = e->rem.getNumberOfItems();
      merk_pos--; // Gleichbedeutend mit nächsten Dreierblock
                                                          "herunterholen"
   }              // => vgl. schriftliche Division

   if(e->rem > div) // Nochmalige Division möglich, wenn Divisor
                                                            kleiner Rest
   {
      e->rem.setActualPos(-1);
      d = e->rem.getItemActual();
      if(ndiv > 1)
      {
         d *= 1000;
         e->rem.prior();
         d += e->rem.getItemActual();
         pdiv--;
      }
      q = ldiv(d, pdiv);
      z = div;
      z *= q.quot;
      e->rem -= z;
      if(e->quot.getNumberOfItems() == 0) e->quot.insert(q.quot);
      else
      {
         if((nr == 1) && (merk_pos == 0))e->quot.insertPos(0, q.quot);
         else if(nr == merk_pos+2) e->quot.insertPos(0, q.quot);
         else
         {
            q.quot += e->quot.getItemPos(0);
            e->quot.replacePos(0, q.quot);
         }
      }
   }
   e->quot.setActualPos(0);
   ue = 0;
   while(!e->quot.isNULL()) // Stellt sicher, dass jedes Element
                                                                wieder
   {                        // maximal 3 Ziffern enthält
      q.quot = e->quot.getItemActual() + ue;

      ue = q.quot - q.quot%1000;
      q.quot -= ue;
      ue /= 1000;

      e->quot.replaceActual(q.quot);
      e->quot.next();
   }
   if(ue > 0) e->quot.insert(ue);

   // Vorzeichen des Quotienten und des Rests setzen
   if(((a.getVZ() ==  1) && (b.getVZ() == -1)) ||
      ((a.getVZ() == -1) && (b.getVZ() ==  1))) e->quot.setVZ(-1);
   if(((a.getVZ() == -1) && (b.getVZ() ==  1)) ||
      ((a.getVZ() == -1) && (b.getVZ() == -1))) e->rem.setVZ(-1);

   return *e;
}
```

5.4 Literaturverzeichnis

- Louis, D., Borland C++ 5 – Das Kompendium, München, Markt&Technik, 1997
- Petzold, C., Windows-Programmierung, Unterschleißheim, Microsoft Press, 2001[5]
- Engel, A., Wahrscheinlichkeitsrechnung und Stochastik – Band 1, Stuttgart, Ernst Klett Verlag, 1973
- Knuth, D.E., Arithmetik, Berlin Heidelberg, Springer-Verlag, 2001

Ich erkläre hiermit, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

Bieswang, den 03. Februar 2003 _____

Unterschrift