Standards für objektorientiertes Programmieren in der Computeralgebra

Diplomarbeit

von

Patrick Merscher

nach einem Thema von Prof. Dr. Buchmann,
am Fachbereich 14 - Informatik -
der Universität des Saarlandes

Saarbrücken, im Januar 1992

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, daß ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Saarbrücken, den 19. Januar 1992

Patrick Merscher

Mein besonderer Dank gilt Herrn Prof. Buchmann für dieses sehr interessante Thema sowie die Betreuung.

Weiterhin danke ich Stephan Düllmann und Ralf Roth für ihre zahlreichen Anregungen und die Hilfe beim Korrekturlesen dieser Arbeit, meiner Freundin Judith Bogatzky für ihre Geduld in der Endphase meines Studiums sowie meinen Eltern für ihre Unterstützung während meiner Ausbildung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Ziel beim Design eines Programmsystems für Anwendungen in der Computeralgebra ist es, eine *große Austauschbarkeit der Algorithmen und Datenstrukturen zu gewährleisten.* Um diesem Anspruch gerecht werden zu können, ist es unabdingbar, im Voraus die Vorgehensweise bei der Implementation und Dokumentation zu *standardisieren*, was durch die vorliegende Arbeit realisiert wird.

Ein solches System in einer objektorientierten Programmiersprache wie C++ zu entwickeln, bringt einige konzeptionelle Probleme mit sich. Zum einen gibt es nicht immer genügend Literatur, die die technischen Probleme, die bei der Implementation auftreten können, ausreichend behandeln würde. Zum anderen würde es einen zu großen Zeitaufwand für den einzelnen Programmierer bedeuten, sich ausgiebig mit Sprachelementen zu beschäftigen, die mit der Aufgabe, die er aktuell zu lösen hat, nichts zu tun haben.

Ziel der vorliegenden Arbeit ist es, sowohl die Vorgehensweise beim Design und der Realisierung der Bibliotheksmodule zu *standardisieren* als auch den Programmierern durch ausführliche Erläuterungen dieser Standards einen schnelleren Einstieg in die Sprache C++ zu vermitteln. Da die vorliegende Arbeit keine Einführung in C++ sein soll, werden an zahlreichen Stellen im Text Hinweise auf in der Literatur gut dargestellte Beispiele zu finden sein. Somit haben Literaturhinweise im Text nicht nur den Sinn, Referenzen für die dargestellten Standards zu sein, sondern sollen den Zeitaufwand für den Leser verkleinern, sich mit Hilfe der Literatur in die Materie einzuarbeiten. Besonderer Wert wird auf die Darstellung eines effizienten Programmierstils in C++ gelegt, um verborgene *Laufzeitverluste* sowie die *Redundanz* von Code in Bezug auf den „objektorientierten Überbau“ von Anfang an zu minimieren. Dabei werden gefährliche Seiteneffekte der Sprache C++, die bei der Programmierung der Anwendungen an unserem Lehrstuhl aufgetreten sind, untersucht und Strategien zu ihrer Vermeidung entwickelt. Außerdem werden Dokumentationsstandards vorgeschlagen, die auf lange Sicht die Lesbarkeit der Programme und damit auch deren Austauschbarkeit gewährleisten sollen. Da es schon viele gute in C implementierte Algorithmen und Datenstrukturen an deutschen Universitäten gibt, soll versucht werden, die C++-Strukturen und die Organisation der Module in Dateien so zu designen, daß die in C programmierten Module leicht in eine objektorientierte Struktur eingefügt werden können. Um den Austausch von Software sowohl mit anderen Instituten als auch innerhalb des eigenen Lehrstuhls, nicht einseitig zu gestalten und eine große Modularität der Software zu gewährleisten, hat man sich an unserem Lehrstuhl entschieden, einige Anwendungen zunächst in C zu implementieren, um sie dann nach einer definierten Vorgehensweise in die Bibliothek einzufügen. Um den Einstieg in das Design eines Systems zueinander in Beziehung stehender C++-Klassen zu erleichtern, werden verschiedene Ansatzmöglichkeiten vorgestellt und ihre Vor- und Nachteile aufgezeigt. Schließlich sind im Anhang einige nützliche Tabellen und Hinweise bzgl. der Standards zu finden, die sowohl am unserem Lehrstuhl, als auch beim Workshop „Softwareentwicklung in der algorithmischen Zahlentheorie und Algebra" in Schloß Dagstuhl vom 29.-30. Oktober 1991 entwickelt wurden. Die in der vorliegenden Arbeit benutzten Programm-Listings enthalten *Skizzen* der tatsächlichen Implementierung, sind jedoch so vollständig, daß alle zur wirklichen Implementation wichtigen Details dargestellt werden.

Die vorgestellten Standards erheben nicht den Anspruch, die jeweils effizienteste Problemlösung darzustellen; vielmehr kam es bei deren Entwicklung auch darauf an, für die bei der Programmierung an unserem Lehrstuhl aufgetretenen sehr komplizierten Sachverhalte in Bezug auf die Sprache C++ *praktikable* Lösungen zu finden. Es hat sich beim Umgang mit der Sprache C++ gezeigt, daß „stabil" laufende Programme sehr leicht *nachträglich* in Bezug auf ihre Effizienz zu verbessern sind. Allerdings von Anfang an zu versuchen, die Effizienz zu maximieren, setzt eine sehr große Erfahrung mit der Sprache voraus, die i.a. nicht von allen Programmierern, die sich in das System einarbeiten, erwartet werden kann. Die Arbeit wurde wie folgt strukturiert:

- Einführung in die Begriffswelt des „objektorientierten Programmierens",
- Grobstrukturierung von Datentypen in C++ für die speziellen Anwendungen an unserem Lehrstuhl,
- Vorstellung von C++-Konzepten und Programmierstandards,
- Designvorschläge für ein System von „abstrakten Datentypen" für Problemstellungen in der Computeralgebra,
- Dokumentationsstandards und Organisation in einem Programmsystem wie dem vorliegenden.

Die wichtigsten dabei vorgestellten Programmier- und Dokumentationsstandards werden durch *Kursivschrift* hervorgehoben.

Natürlich konnten im Rahmen dieser Arbeit nicht alle C++-Features standardisiert werden, allerdings sind die vorgestellten Konzepte, für die eine Standardisierung vorgenommen wurden, bereits programmiert worden und somit verläßlich in Bezug auf ihre Sicherheit. Die im letzten Kapitel formulierten Dokumentationsstandards berücksichtigen sowohl die Dokumentation der Programme als auch ihre Organisation in Dateien. Die Steuerung der Generierung eines Programmsystems erfolgt in der Regel durch sogenannte *Makefiles*, deren Struktur ebenfalls standardisiert wird. Mit Hilfe des vorliegenden Textes sollte ein Programmierer daher in die Lage versetzt werden, eine C++-Klasse weitgehend standardisiert zu programmieren und richtig zu compilieren.

Kapitel 2

Objektorientiertes Programmieren (OOP)

2.1 Charakterisierung von OOP

Dieses Kapitel gibt eine kurze Einführung in die wichtigsten Eigenschaften der *objektorientierten* Sprache C++ und klärt für das weitere Verständnis wichtige Begriffe und Bezeichnungen. Unter *Objektorientiertem Programmierstil* versteht man die Abstrahierung von Daten und Operationen auf diesen Daten. Dadurch werden *Wiederverwendbarkeit* von bereits existierendem Programm-Code sowie eine außerordentlich *modulare Programmierung* unterstützt.

Im wesentlichen läßt sich OOP durch fünf Stichworte beschreiben:

- Definition abstrakter Datentypen (Klassen),
- Verkapselung,
- Vererbung,
- Dynamische Bindung,
- Überladen von Operatoren.

Mit diesen Sprachelementen wird es möglich, beliebige *abstrakte Datentypen*, wie z.B. die algebraischen Strukturen Ring, Körper etc. , zu definieren, deren Eigenschaften sich formal beschreiben lassen.

2.2 Klassen und Objekte

Bezeichnung 2.1 *Eine* **Klasse** *ist ein vom Programmierer definierter Datentyp. Die Deklaration einer Klasse spezifiziert eine Repräsentation von Objekten mit bestimmten Eigenschaften und eine Menge von Operationen (Funktionen) auf diesen Objekten. Objekte werden auch* **Instanzen einer Klasse** *genannt und sind Variablen oder Strukturen vom Typ einer Klasse.*

2.3 Verkapselung

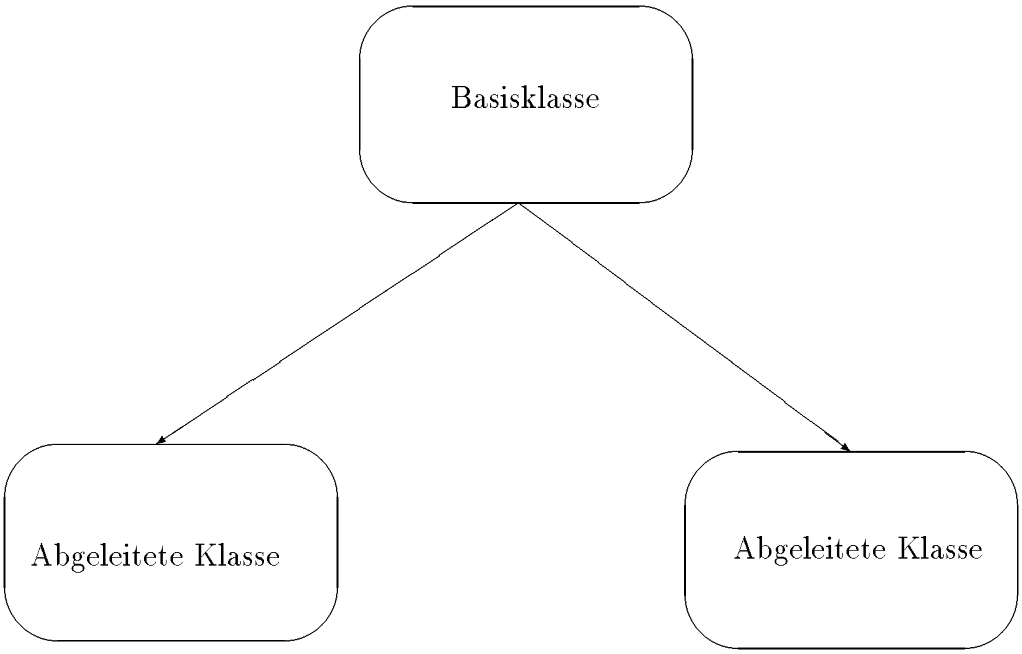
Der Mechanismus der *Verkapselung* regelt den Zugriff auf Objekte und deren Daten und definiert gleichzeitig die Menge der Operationen, die für ein Objekt zulässig und/oder gewünscht ist. C++ stellt hierfür die Schlüsselworte *public, private* und *protected* zur Verfügung, auf die im Kapitel „Verkapselung und Information-Hiding“ ausführlich eingegangen wird.

2.4 Vererbung

Vererbung ist ein Sprachelement von C++ , welches es bei der Definition einer neuen Klasse erlaubt, sich bereits in einer anderen Klasse definierte Abläufe oder Datenrepräsentationen (Funktionen, Variablen) zunutze zu machen.

Bezeichnung 2.2 *Eine Klasse, die Eigenschaften an eine andere Klasse vererbt, heißt* **Basisklasse***; die Klasse, die Eigenschaften erbt, heißt* **abgeleitete Klasse**.

Abb. 1: Basisklasse und abgeleitete Klasse



Das Vererbungskonzept hat zur Konsequenz, daß in einer abgeleiteten Klasse nur noch die *Unterschiede* zu ihrer schon existierenden Basisklasse, nicht aber die gesamte Klasse selbst implementiert werden müssen. Zwischen der abgeleiteten Klasse und ihrer Basisklasse besteht also eine „is kind of"-Beziehung. Existiert zu einer Klasse nur genau eine Basisklasse, so spricht man von *einfacher Vererbung*, bei mehr als einer Basisklasse von *mehrfacher Vererbung.*

2.5 Dynamische Bindung

Bezeichnung 2.3 *Ein System von untereinander durch Vererbung in Beziehung stehender Klassen heißt* **Klassenhierarchie**.

Dynamische Bindung macht es möglich, dieselbe Funktion in zueinander in Beziehung stehenden Klassen unterschiedlich zu implementieren. Wird diese Funktion für ein Objekt aufgerufen, so „sucht" sich der Compiler die ,,passende" Funktion in der Klassenhierachie. In diesem Falle wird ein *Objekt dynamisch* an die entsprechende Funktion *gebunden.* Um die internen Verweise der Funktionen untereinander verwalten zu können, stellt C++ das Schlüsselwort *virtual* zur Verfügung (siehe Kapitel 6).

2.6 Überladen von Operatoren

In C++ sind alle Standardoperatoren *überladbar*. Dies bedeutet, daß an einen von C++ zur Verfügung gestellten *Operator*, wie z.B. `+` oder `-`, ein vom Programmierer definierter Funktionsaufruf geknüpft werden kann, der automatisch erfolgt, wenn der Operator in Verbindung mit einem oder zwei Operanden eines vom Programmierer festgelegten Typs auftritt. Man sagt in einem solchen Fall, der Operator sei mit der Funktion *überladen*.

Daraus resultiert die Möglichkeit, unterschiedliche - *vom Benutzer definierte* - Datentypen durch *Standardoperatoren* miteinander verknüpfen zu können.

Eine Besonderheit von C++ ist, daß die Äquivalente zu `malloc` und `free` in C - `new` und `delete` - ebenfalls Operatoren sind. Damit wird es möglich, diese Operatoren mit einer effizienten Speicherverwaltung zu überladen, die von einer Basisklasse an alle in der Hierarchie vorkommenden Klassen vererbt wird (siehe Kapitel 5).

C++ stellt zum Überladen von Operatoren das Schlüsselwort *operator* zur Verfügung (siehe Kapitel 5).

Kapitel 3

Allgemeine Struktur einer Klasse

3.1 Membervariablen in Klassen

Für die Anwendungen in der Computeralgebra und die sehr speziellen Objekttypen, die in diesem Zusammenhang auftreten, wird im folgenden zusammengestellt, welche Bestandteile sinnvoll und welche insbesondere im Blick auf das spätere Zusammenbinden der Klassen zwingend notwendig sein werden. Diese Komponenten werden nun einzeln für eine einfache Klasse „Lange Zahlen“ (`Bignum`) vorgestellt und später im Überblick nochmals dargestellt. Dieser Objekttyp eignet sich gut als Beispiel, da hier schon kompliziertere Situationen in C++ auftreten, deren Ausprägungen aus der existierenden Standardliteratur nur sehr mühsam herauszulesen sind.

Bezeichnung 3.1 Membervariablen *oder* **Members** *von Klassen sind Variablen verschiedenen Typs, die - wie bei Strukturen in C - die Strukturkomponenten einer Klasse darstellen. Diese können vom Typ aller Standarddatentypen sowie beliebiger vom Benutzer definierter Typen sein.*

3.1.1 Deskriptoren

Die erste Komponente jeder Klasse, im folgenden *Deskriptor* genannt, ist die Variable, die den Namen der Klasse angibt:

```
static char* classname="Bignum";
```

Damit erreicht man, daß sich ein beliebiges Objekt jederzeit durch seinen Namen identifizieren kann, wenn der Typ des Objektes in der Hierarchie zu einem bestimmten Zeitpunkt nicht klar ist.

3.1.2 Membervariablen

Dem Klassennamen folgen die eigentlichen Strukturvariablen der Klasse. Diese enthalten alle für die vollständige Charakterisierung des Objektes notwendigen Variablen und Werte. Ein Objekt vom Typ „Lange Zahl“ wird durch seine Länge und seine Ziffern eindeutig charakterisiert. Da die Länge eines solchen Objektes variabel ist, geht man wie in C vor und deklariert in der Klasse einen *Zeiger* auf den Darstellungstyp der langen Zahl. Klassenintern ist dieses Objekt also im wesentlichen durch die Variablen

```
ULONG min;          //Laenge der Variable vom Typ unsigned long
LONGTYPE b;         //Zeiger auf die lange Zahl
```

repräsentiert.

Die Datentypen `ULONG` und `LONGTYPE` sind aus den C-Standarddatentypen `unsigned long` und `unsigned long *` entstanden:

```
typedef unsigned long ULONG;
typedef ULONG* LONGTYPE;
```

3.2 Konstruktoren und Destruktoren

Neben den Members (auch *Klassenelemente* genannt) gibt es auch sogenannte *Elementfunktionen (member-functions)*. Durch diese Funktionen wird eine Menge von Operationen auf den Members definiert. Zu einer besonderen Gattung von Elementfunktionen gehören die *Konstruktoren* und die *Destruktoren*:

Bezeichnung 3.2 Konstruktoren *sind Elementfunktionen, die einerseits Speicher für Objekte variabler Länge reservieren können und andererseits Typumwandlungen realisieren. Sie werden automatisch bei der Deklaration eines Objektes aufgerufen.*

Bezeichnung 3.3 *Der* **Destruktor** *ist die Elementfunktion der Klasse, die den von den Konstruktoren reservierten Speicher automatisch wieder freigibt, sofern er allokiert wurde.*

Wird ein Konstruktor aufgerufen, so wird das deklarierte Objekt *instanziiert*, d.h. es wird entweder eine bestimmte Repräsentation des Objektes gewählt und/oder ausreichend Speicher für Wertzuweisungen bereitgestellt. Verbindliche Konstruktor- und Destruktor- Definitionen sind:

- der *Default-Konstruktor* mit leerer Parameterliste,
- der *Value-Konstruktor* mit Länge- oder Wertangabe,
- der *Copy-Konstruktor*,
- und der *Destruktor* mit leerer Parameterliste.

In den nächsten Abschnitten werden diese Elementfunktionen genauer beschrieben.

3.2.1 Der Default-Konstruktor

In unserem Beispiel hat der Default-Konstruktor etwa folgendes Aussehen:

```
Bignum::Bignum()
{
  b = new LONGTYPE[SIZE];
  // b ist die interne Darstellung einer Bignum
}
```

Die Semantik von `Bignum::Bignum()` ist die Herstellung des *lokalen Bezugsrahmens* einer Elementfunktion zu einer Klasse. Die Funktion `Bignum()` könnte auch unter *demselben Namen* in einer anderen Klasse definiert sein. „::" ist der sogenannte *scope-operator*, der dem Compiler mitteilt, daß in diesem Falle gerade die Funktion `Bignum()` in der Klasse `Bignum` gemeint ist.

Der Destruktor der Klasse `Bignum` hat damit das folgende Aussehen :

```
Bignum::~Bignum()
// Die Tilde identifiziert die Funktion als Destruktor
{
  delete b;
}
```

Der Destruktor der Klasse Bignum löscht eine Instanz eines Objektes vom Typ `Bignum`. Die Besonderheit ist, daß in diesem Fall nicht bekannt sein muß, wie groß die zu löschende Variable aktuell ist. Die Destruktoren werden in C++ immer in *umgekehrter Reihenfolge* wie die Konstruktoren aufgerufen. Dadurch erhält der Compiler immer die Information, wieviel Speicher er freizugeben hat, sofern dieser Speicher zuvor von einem *Konstruktor* allokiert wurde. Bei allen anderen dynamischen Zugriffen muß die Länge immer explizit angegeben werden:

```
Bignum *a;
a = new Bignum[5];// Allokiert 5 Bignums,
                  // mit unbekannter Laenge

delete a;         // Inkorrekt, Laenge fehlt!!!
delete [5] a;     // Korrekt
```

Es gibt mehrere Situationen, in denen eine *Nichtdeklaration* dieses Konstruktors zu Fehlern führen kann:

1. die Deklaration von Feldern von Klassen,
2. die einfache Deklaration eines Objektes ohne Parameter,
3. implizite Aufrufe des Copy-Konstruktors.

Beispiel: Die Deklaration des Feldes

```
Bignum array[50];
```

ruft 50 mal den Konstruktor `Bignum()` auf. Wäre dieser nicht definiert, so würde es zur Compilezeit schon Fehler geben. Auch würde die Nichtdeklaration bedeuten, daß ein Benutzer bei einer Deklaration für ein Feld von Objekten des Typs `Bignum` die Konstruktoren und/oder Werte für die einzelnen Feldkomponenten explizit angeben müßte:

```
Bignum a[4] = {Bignum(2),Bignum(3),Bignum(),Bignum(8)}
```

3.2.2 Der Value-Konstruktor

Dieser Konstruktortyp soll folgende Aufgaben übernehmen:

- Typkonvertierungen (z.B. `int->Bignum`),
- Wertzuweisungen bei einer Deklaration,
- Bereitstellung von Speicher (z.B. eine feste Anzahl von Ziffern).

In der Klasse `Bignum` hat er folgendes Aussehen:

```
Bignum::Bignum(int len)
{
  b = new LONGTYPE[len + 1];
}
```

Dadurch werden dem Benutzer der langen Arithmetik Deklarationen der Form

```
Bignum zahl(50);
```

ermöglicht. Für die Variable `zahl` werden mit dieser Implementierung 50 Stellen reserviert. Der Nachteil dieser Deklaration ist die Einschränkung der Funktionalität des Konstruktors, da z.B. eine direkte Wertübergabe durch obige Deklaration nicht möglich ist. Die Deklaration

```
Bignum zahl(50);
```

sollte daher in den jeweiligen Situationen folgende Semantiken haben;

- Bereitstellung von Speicher,
- interne Umwandlung der `int`-Zahl `50` in den Typ `Bignum`,
- Wertzuweisung der Zahl `50` an die Variable `zahl`.

Um diese größere Funktionalität zu erreichen, muß die obige Defintion nur geringfügig abgeändert werden:

```
Bignum::Bignum(int arg)
{
  int len;
  len = sizeof(int);
      ...
  b = new LONGTYPE[len + 1];
  // Kopiere Ziffern von arg in die Ziffern von b
      ...
}
```

Damit werden Verknüpfungen der folgenden Art möglich:

```
Bignum a;
Bignum b;
int    c;

 a = b + c;
```

Die Variable `c` wird dann während dem Aufruf der `operator+()`-Funktion der Klasse `Bignum` automatisch in den passenden Typ (`Bignum`) konvertiert, d.h. das rechte Argument bei der Addition ist dann in Wirklichkeit `Bignum(c)` (siehe auch Kapitel 5). Statt dem Argumenttyp `int` für diesen Konstruktor hätte man auch z.B den Typ `long` nehmen können, da beide Datentypen, sofern die Länge der Zahl im jeweiligen Bereich dieser arithmetischen Datentypen liegt, kompatibel sind. Diese Konvertierung ist nicht notwendig, wenn die `operator+()`-Funktionen alle auftretenden Typen explizit überladen! Die Sicherheitsaspekte von „impliziten Typkonversionen" sind in [3] S.40ff grafisch dargestellt. Sollte der Fall auftreten, daß ein Benutzer vor einer Wertzuweisung nur Speicher für eine von ihm gewünschte Stellenanzahl reservieren möchte, kann er dies einfach durch die Übergabe einer Zahl der entsprechenden Länge tun. *Auch diese Konstruktorart muß in jeder arithmetischen Klasse vorhanden sein, um dem Benutzer der Klasse die Möglichkeit einfacher, effizienter Wertzuweisungen und Speicherplatzallokierung zu geben.*

3.2.3 Der Copy-Konstruktor

Dieser Konstruktor ist der wichtigste in den in unserem Zusammenhang auftretenden Klassen. Auf die meisten seiner Eigenschaften wird erst in späteren Kapiteln eingegangen. Die wichtigste Funktion dieses Konstruktortyps ist der *Aufbau eines Stacks zur Evaluierung geschachtelter Ausdrücke.* In C stellt sich bei arithmetischen Funktionen das Problem, wie die Ergebnisse geschachtelter binärer Operationen „gerettet" werden können. Will man beispielweise die Multiplikation zweier langer Zahlen funktionalisieren, gibt es zwei Möglichkeiten, die Multiplikationsroutine `mul()` zu realisieren:

1. die Funktion `mul()` gibt einen Zeiger auf das Ergebnis zurück,
2. Die Funktion schreibt das Ergebnis in einen formalen Parameter.

Da aber lange Zahlen durch eine *Struktur* dargestellt werden, ist die erste Möglichkeit in Hinblick auf geschachtelte Operationen sehr fehleranfällig. Die Funktion `mul` hätte dann nämlich etwa folgendes Aussehen:

```
LONGTYPE *Bignum::mul(LONGTYPE lval, LONGTYPE rval)
{
  LONGTYPE product[len]; // Ergebnisvariable

  product = lval * rval;//Der Multiplikationsalgorithmus
  return product;
}
```

Dies führt zu folgenden Problemen:

1. Die Variable `product` muß genügend Speicher für das Ergebnis bereitstellen.

2. Der statisch in der Funktion für die Ergebnisvariable allokierte Speicher wird nach dem Verlassen der Funktion wieder freigegeben, d.h. das Ergebnis liegt in einem wieder zum Überschreiben freigegebenen Speicherbereich.

3. Wird der Speicher lokal in der Funktion dynamisch allokiert, kann man ihn nicht mehr freigeben, da man nach dem Verlassen der Funktion keinen Zugriff mehr auf diese dynamisch allokierte Variable hat.

Ausdrücke der Form

```
a = mul(b, mul(c,d));
```

sind praktisch nicht zu kontrollieren, wenn die Variablen, wie in diesem Falle, Zeiger auf Felder o.ä. sind. Möchte man nun in C++ den Operator `*` mit einer solchen Funktion `mul` überladen, so müssen die Ergebnisse der einzelnen Schritte gerettet werden. Dies bedeutet, daß der Speicherbereich, in dem die Ergebnisse stehen, solange allokiert bleibt, bis der Ausdruck vollständig ausgewertet worden ist. Eben dies wird durch die Definition des *Copy-Konstruktors* gewährleistet. Der Copy-Konstruktor wird automatisch nach dem Abarbeiten des `return`-Statements der Operator-Funktion aufgerufen! In dem folgenden Beispiel wird dieser Ablauf nochmals dargestellt:

```
Bignum a,b,c,d;
a = b + c - d;
```

Diese Zuweisung hat intern die folgende Semantik:

```
=(a,+(b,-(c,d)))
```

Hier müssen also die Zwischenergebnisse

```
-(c,d)
+(b,-(c,d))
```

gerettet werden, bis das Gesamtergebnis in `a` steht. Die Syntax des Copy-Konstruktors ist nun wie folgt:

```
Bignum(const Bignum& a)
{
  int i = (int)a.b[0]; //Laenge der Zahl
  b = new LONGTYPE[i + 2];
  //Platz fuer Zwischenergebnisse reservieren
```

```
  while(i>=0)
  {
    b[i] = a.b[i];     //Ergebnisse kopieren
    i--;
  }
}
```

In der Funktion, die den Gleichheitsoperator überlädt (siehe Kapitel 5), wird der hier allokierte Speicher wieder freigegeben.

3.3 Weitere Elementfunktionen

3.3.1 Arithmetische Funktionen

Die für die Klassen in einem Computeralgebrasystem als Standard vereinbarten *arithmetischen* Funktionen werden im Anhang aufgelistet. Wie die Operatoren überladen werden und auf welche Programmierfallen dabei zu achten ist, wird im Kapitel 5 erklärt. Eine tabellarische Auflistung der Funktionen und Operatoren ist in Anhang A zu finden. *Für alle diese Funktionen müssen auch die entsprechenden Operatoren überladen werden.*

3.3.2 Identifikationsfunktionen und Debughilfen

Im Blick auf die zukünftige Einbindung der Klassen in eine Hierachie, wird eine Funktion `Bignum::desc()` definiert, die einfach den Klassennamen zurückgibt:

```
char *Bignum::desc()
{
  return &classname;//Adresse des Namens
}
```

Innerhalb einer Vererbungshierarchie, wo die Situation auftreten kann, daß ein Objekt*typ* erst *dynamisch* festgestellt werden kann, ist dies eine Möglichkeit, ein Objekt durch seinen „ursprünglichen Typ" identifizieren.

Da die Aufrufreihenfolge der Konstruktoren und Destruktoren nicht leicht nachzuvollziehen ist, werden nun einige Members und korrespondierende Elementfunktionen vorgestellt, die ein Objekt, welches in einem Konstruktor verändert wird, eindeutig identifizieren. Damit kann man den Weg eines bestimmten Objektes in der Aufrufkette von Konstruktoren und Destruktoren genau nachvollziehen. *Die Realisierung dieses Hilfsmittels sollte gleich bei der*

Implementation der Konstruktoren erfolgen, um aufwendiges Debugging in der Anfangsphase der Implementierung einer Klasse zu vermeiden.

Es hat sich dazu als sinnvoll erwiesen, die Membervariablen

```
static int new_number = 1;
int onr;
int first_object = 1;
//first_object = 1 ==> Neue Nummer kann vergeben werden
//first_object = 0 ==> Nummer bereits vergeben
```

und die Funktionen

```
void Bignum::inc_nn()
{ if(first_object)
  {
     onr = new_number++;//Vergabe der Identifikationsnummer
     first_object = 0;  //Nummer nur einmal vergeben
  }
}

int identify() const { return onr; };
//Identifiziere aktuelles Objekt mit seiner Nummer
//identify muss public definiert sein, um den Zugriff
//von aussen zu gewaehrleisten!!!
```

in jede Klasse aufzunehmen. Die Variable `new_number` wird für jede Instanz einer Klasse inkrementiert. Dies erfolgt durch den Aufruf von `inc_nn()` in jedem Konstruktor der Klasse. Das Flag `first_object` garantiert, daß die dem Objekt gegebene Objektnummer nur einmal vergeben wird und dann bis zur Zerstörung des Objektes dieselbe bleibt. Nach Deklarationen wie

```
Bignum a;
Bignum b;
Bignum c;
```

sind die Objekte a,b,c mit ihren Namen (Funktion `desc()`) und ihrer Objektnummer „markiert". Möchte man nun beispielsweise den Weg von `a` verfolgen, so läßt man sich nach der Deklaration seine Nummer ausgeben, was mit Hilfe der Funktion `identify()` geschieht:

```
a.identify(); // Gibt die an a vergebene Objektnummer aus
```

Möchte man an irgendeiner Stelle (z.B. innerhalb eines Konstruktors) genau wissen, welches der vorher deklarierten Objekte gerade aktuell manipuliert wird, ruft man nochmals die Funktion `identify()` für dieses Objekt auf und vergleicht die Objektnummer mit der Nummer, die man durch einen der Aufrufe von `identify()` bei den Variablendeklarationen erhalten hat. Stimmen beide Nummern überein, so handelt es sich um dasselbe Objekt.

3.3.3 Hilfsfunktionen

In einer Klasse werden in aller Regel eine Menge von Hilfsfunktionen definiert, die nur in dieser Klasse auftretende *Zwischenrechnungen* realisieren. Wie man diese Funktionen am besten gestaltet, wird im nächsten Kapitel behandelt. Bei diesen Funktionen ist insbesondere auf ihre korrekte Organisation in Dateien zu achten (siehe Kapitel 7).

3.3.4 Fehlerbehandlung

Jede Klasse sollte einen gewissen Stamm an Sicherheits- und Fehlerfunktionen beinhalten. Diese Funktionen sollen dem Programmierer und dem späteren Anwender im Fehlerfalle gute Debugmöglichkeiten bieten und gleichzeitig eine gewisse Menge *vorhersehbarer* Fehler abfangen. Ein Beispiel dazu ist das Überladen des Indexoperators `[]`. Dieser kann so überladen werden, daß eine Bereichsüberschreitung in Schleifen oder Deklarationen zur Laufzeit zu *kontrollierten* Fehlern führt. Den Ablauf, wie der Indexoperator zu überladen ist, kann man ohne große Änderungen aus der Literatur ([5] Kap. 5.2.2, oder [3])übernehmen. In Kapitel 5 werden diese Funktionen nochmals behandelt.

3.3.5 Ein/Ausgabe-Routinen

Bei der Programmierung in C++ soll die Benutzung der C-Funktionenfamilie `printf` und `scanf` aus folgenden Gründen vermieden werden:

- Die Funktion `printf` verlangt bei jedem Aufruf *explizite Formate* und ist damit sehr unkomfortabel.
- Der Zugriff auf tief verschachtelte Variablen ist sehr fehleranfällig und eventuell gar nicht möglich, wenn die auszugebende Membervariable verkapselt ist.

Der objektorientierte Ansatz zielt darauf, diese umständliche Ausgabe nur *einmal* als Elementfunktion in der Klasse zu definieren und damit dem Benutzer des Systems zur Verfügung zu stellen. Im Gegensatz zu C, wo die Ein/Ausgabe über *Funktionen* erfolgt, stellt C++ die *Objekte* `cout` und `cin` bereit. Diese sind Instanzen der C++-Standardklassen `ostream` und `istream` und haben den *Linkshiftoperator* `<<` bzw. den *Rechtsshiftoperator* `>>` mit Ein/Ausgabe-Funktionen überladen. Auf die Technik, diese speziellen Operatoren klassenspezifisch zu überladen, wird in Kapitel 5 eingegangen.

3.3.6 Die Grobstruktur

Im folgenden werden die einzelnen Bestandteile nochmal in der Reihenfolge ihres Auftretens in einer Klasse zusammengefaßt:

1. Deskriptoren,
2. Membervariablen,
3. Hilfsfunktionen,
4. Operatorfunktionen,
5. Sicherheits- und Fehlerfunktionen,
6. Ein/Ausgabe-Funktionen,
7. Konstruktoren,
8. Destruktoren.

Eine detaillierte Auflistung der Funktionen im einzelnen ist ebenfalls im Anhang zu finden.

Kapitel 4

Verkapselung und Information-Hiding

C++ stellt explizite Mechanismen bereit, um die *Zugriffskontrolle* auf Membervariablen, Elementfunktionen und Objekte zu regeln. Dies geschieht mit Hilfe der Schlüsselworte *private, public* und *protected*. Bei der Implementation einer Klasse sollte großer Wert auf die Benutzung dieser Mechanismen gelegt werden, um von Anfang an die Sicherheit des Systems in Bezug auf Mißbrauch von Funktionen und Variablen durch die Benutzer der Klasse zu maximieren.

4.1 Verkapselungsmechanismen

4.1.1 Private (*private*) Zugriffe

Mit dem Schlüsselwort *private* gibt der Programmierer dem Compiler die Anweisung, alle so innerhalb einer Klasse deklarierten Variablen und Funktionen vor dem eventuellen Zugriff durch Benutzer zu schützen.

Eine *private*-Deklaration könnte zum Beispiel so aussehen:

```
class Bignum
{
 private:
  ...
  LONGTYPE b;//Die lange Zahl
  init();
  ...
}
```

Damit sind die Variable `b` und die Funktion `init()` vor allen Zugriffen außerhalb der Klasse Bignum geschützt:

```
Bignum a;
a.b = <Wert>;     //Compilerfehler, b ist private!!!
a.init();    //Compilerfehler,init() ist private!!!
```

Von diesem Schlüsselwort sollte ausgiebig Gebrauch gemacht werden, wenn klar ist, daß auf bestimmte Variablen oder Funktionen von Benutzern der Klasse nicht zugegriffen werden soll. Dies gilt ebenfalls für „kritische Funktionen“ wie zum Beispiel Speicherverwaltungsfunktionen und elementare Hilfsfunktionen, die die Länge einer Membervariable verändern. In dem obigen Beispiel, in dem der Zugriff

```
a.b = <Wert>;
```

einen Compilerfehler verursacht, nennt man die Variable b *verkapselt*.

4.1.2 Freunde (*friends*) einer Klasse

Wenn man die Restriktionen, die `private`-Deklarationen einer Klasse mit sich bringen, *gezielt* kontrollieren will, gibt es die Möglichkeit, Funktionen und Objekte als sogenannte *friends* (Freunde) einer Klasse zu definieren. Für Freunde einer Klasse wird der Verkapselungsmechnismus außer Kraft gesetzt. Eine befreundete *Funktion* darf auf die Members und Funktionen der Klasse zugreifen, deren Freund sie ist.

```
class Bignum
{
 private:
  LONGTYPE b;
  friend Bignum operator+(Bignum&, Bignum&);
}
```

Durch diese Deklaration ist es innerhalb der Funktion `operator+()` nunmehr erlaubt, die Variable `b` anzusprechen, obwohl diese `private` deklariert ist. Damit wird es auch sinnvoll, in bestimmten Situationen *Private Klassen* zu definieren (siehe [5] S.65f).

Bezeichnung 4.1 *Eine* **Private Klasse** *ist eine Klasse, in der alle Members und Funktionen* `private` *deklariert sind.*

Private Klassen werden in aller Regel als Hilfsklassen definiert. Sie entziehen sich dem Sichtbarkeitsbereich aller Klassen und Funktionen, die in der privaten Klasse nicht als Freunde ausgewiesen sind:

```
class private_klasse
{
  friend class Bignum;// Bignum hat Zugriff
          ...
}
```

Natürlich muß auch in der Klasse `Bignum` die Deklaration

```
friend class private_klasse;
```

vorkommen, damit der Zugriff auf `private_klasse` erlaubt ist.

Auf andere wichtige Eigenschaften der Freund-Funktionen wird im Kapitel 5 eingegangen.

4.1.3 Öffentlicher (*public*) Zugriff

Möchte man Variablen oder Funktionen einer Klasse für das gesamte System zugänglich machen, so deklariert man sie als *public*-Elemente. Die wichtigsten *public-Funktionen* einer nicht privaten Klasse sind die Konstruktoren:

```
class Bignum
{
  public:
   Bignum();
}
```

Wäre in diesem Beispiel der Default-Konstruktor nicht als eine public-Funktion definiert, so würde die Deklaration

```
Bignum a;// Fehler, Bignum() ist privat!!!
```

einen Compilerfehler verursachen, da mit der Deklaration implizit von „außen“ versucht wurde, eine als privat deklarierte Funktion (den Konstruktor) aufzurufen. Natürlich gibt es auch die Möglichkeit, Konstruktoren als private Funktionen zu deklarieren. In privaten Klassen sind natürlich auch die Konstruktoren privat und dürfen nur von Freunden benutzt werden.

4.1.4 Geschützter (*protected*) Zugriff

Dieses Schlüsselwort hat dieselbe Wirkung wie Benutzung von *private*. *protected* deklarierte Members und Funktionen sind nach „außen“ vor Zugriffen der Benutzer geschützt. Der Unterschied zu *private* ist, daß innerhalb von Klassen auf als `protected` definierte Variablen und Funktionen einer anderen Klasse nur dann zugegriffen werden darf, wenn diese Klasse eine Basisklasse ist. Man betrachte dazu auszugsweise die Klasse Object:

```
class Object
{
    ...
 protected:
  print();
    ...
}
```

Auf die Funktion `print()` dürfte innerhalb der Klasse Bignum zugegriffen werden, wenn `Object` eine Basisklasse von Bignum wäre:

```
class Bignum : public Object  // Erbt von Basisklasse Object
{
        ...
  print() { Object::print();} // Korrekt
}
class Bignum
{
        ...
  print() { Object::print();} // Fehler,print() ist protected!
```

Im allgemeinen gilt damit die Regel: *Elementfunktionen und Members einer Klasse sollten so oft wie möglich* private *bzw.* protected *deklariert werden.*

4.2 Information-Hiding

Wie bei professioneller Softwareentwicklung sollte man auch bei der Programmierung mit Klassen großen Wert auf die scharfe *Trennung zwischen Definition und Deklaration* legen. Die Realisierung einer Klasse besteht also folglich aus mindestens zwei Files.

1. class.h (enthält die *Deklaration* der Variablen und Funktionen),

2. class.c (enthält die *Implementation* der Funktionen).

Die arithmetischen reinen C-Funktionen werden sinnvoll gruppiert in Dateien abgelegt (siehe Kapitel 7). Der Funktionsskopf

```
Bignum Bignum::operator+(Bignum& lval,Bignum& rval);
```

ist in `Bignum.h` *deklariert* und der Funktionskörper ist in `Bignum.c` *implementiert* Ein Benutzer, der die Klasse `Bignum` benutzen möchte, baut sein Programm wie folgt auf :

```
#include <Bignum.h>
main()
{
  Bignum a;
    ...
}
```

Mit der obigen Organisation werden mit Hilfe von `#include` nur *Vorwärtsdeklarationen* auf die Funktionen von Bignum, jedoch keine Implementationsdetails kopiert. Die Information über die Implementierungen ist in den .c-Dateien „versteckt".

Kapitel 5

Effiziente Programmierstile

In diesem Kapitel werden einige Konzepte von C++ vorgestellt, die die *Effizienz* und *Sicherheit* des Programmierens mit Klassen sehr gut unterstützen. Einige beim Studium von Literatur über C++ nicht offensichtliche „Programmierfallen", die insbesondere beim Überladen von Operatoren auftreten, und „verborgene" Laufzeitverluste werden dabei näher untersucht. Die so gewonnenen Erkenntnisse werden als Programmierstandards formuliert, um eine aufwendige Fehlersuche bei der Neuprogrammierung frühzeitig zu vermeiden, sofern dies möglich ist.

5.1 Makros (*inline*-Funktionen)

Die Verwaltung von Funktionen zur Laufzeit ist sowohl in C als auch in C++ sehr aufwendig. Jeder Funktionsaufruf beinhaltet zur Laufzeit des Programmes den Aufbau eines Funktionsstacks zum Sichern von Registern und Rücksprungadressen sowie eine Verzweigung des Programmes zur Stelle der Funktionsdefinition. Die meisten C-Programme sind im wesentlichen so programmiert, daß der Ablauf des Programmes in der Funktion `main()` durch den Aufruf einer Folge von Funktionen realisiert ist. Vom Standpunkt des Laufzeitverhaltens wäre es viel „billiger", den gesamten Ablauf in der Funktion `main()` zu realisieren, da damit die aufwendige Verwaltung der Funktionen entfallen würde. Die Modularität eines solchen Systems ginge natürlich ganz verloren, und der Code wäre bei großen Anwendungen nur noch schwer lesbar. Was man also benötigt, ist eine Möglichkeit, die Funktionen an der Stelle ihres Aufrufs *textuell* einzusetzen, ohne daß der Programmierer davon etwas merkt. In C gibt es dafür das Feature der Definition von *Makros* mit Hilfe von `#define`. Die Nachteile dieser Methode sind

- umständliches Debugging der Fehler in diesen Makros,

- manchmal unübersichtlicher Code.

C++ bietet die sogenannten *inline*-Funktionen als eine Alternative, um den Overhead, der beim Aufruf vieler *kleiner* Funktionen (Konstruktoren, Ein/Ausgabe-Funktionen etc.) entsteht, elegant zu minimieren. Durch die Deklaration

```
inline Bignum();// Konstruktor fuer Bignum
```

wird der Funktionskörper textuell im Objektcode des Programms an die Stelle gesetzt, von der `Bignum::Bignum()` aufgerufen wurde. Der Vorteil liegt auf der Hand. Man benutzt einfach die Syntax für die Definition von Funktionen und hat durch das Voranschreiben des Schlüsselwortes *inline* gleichzeitig alle Vorteile von Makro- und Funktionsdefinitionen zur Verfügung. *Um die Benutzung von Funktionen in Klassen in Bezug auf das Laufzeitverhalten zu optimieren, soll* inline *in den folgenden Situationen immer benutzt werden:*

- bei Funktionen mit einem kleinen Funktionskörper, d.h. unter 500 Zeilen (siehe auch [5]),
- bei *allen* Funktionen, die sehr oft aufgerufen werden (Konstruktoren und Destruktoren),
- bei Funktionen mit vielen Parametern.

Die meisten C++-Compiler sind heute jedoch noch reine Präprozessoren, die den C++-Code nach C übersetzen und ihn dann nochmals mit einem reinen C-Compiler übersetzen. Da die Konstruktoren abgeleiteter Klassen aber oft tief verschachtelte Aufrufe von Konstruktoren anderer Klassen nach sich ziehen, wird der vom Präprozessor generierte C-Code für die meisten C-Compiler zu kompliziert, und der Compiler erzeugt scheinbar zusammenhangslose Fehler. *Daher dürfen* `inline`*-Funktionen nur bei „trivialen" Konstruktoren benutzt werden, d.h. Konstruktoren, die keine impliziten Aufrufe anderer Konstruktoren nach sich ziehen.* `inline`*-Funktionen sollen allgemein nicht benutzt werden bei:*

- rekursiven Funktionen,
- komplizierten Konstruktoren und virtuellen Destruktoren abgeleiteter Klassen,
- sehr großen Funktionen.

Die Benutzung ist ebenfalls kritisch, wenn zu erwarten ist, daß die Funktion sehr oft geändert wird, da im Gegensatz zu nicht als `inline` definierten Funktionen bei einer Änderung alle Programme, die die Funktion aufrufen, neu übersetzt werden müssen. In einem solchen Fall muß man einen Mittelweg zwischen den langen Compilezeiten und der eventuell schlechteren Laufzeit dieser Funktionen finden. Dabei geht man am besten nach folgender Heuristik vor:

- Wird die Funktion sehr oft aufgerufen, dann deklariere man sie auf jeden Fall `inline`.
- Ist zu erwarten, daß die Funktion oft geändert wird, aber nicht sehr oft aufgerufen wird, dann vermeide man die inline-Deklaration, auch wenn sie nur aus einem *kleinen* Funktionskörper besteht.
- Andernfalls benutze man die oben erklärten Regeln.

Es gibt die Möglichkeit, die Benutzung von *inline* abzukürzen, indem man in der Headerdatei der Klasse (class.h), den Funktionskörper explizit hinschreibt. Statt

```
inline ~Bignum();
```

könnte man auch schreiben :

```
~Bignum(){ delete b;};
```

Dies widerspricht zwar der sauberen Trennung von Implementation und Deklaration, ist aber in Hinsicht auf die Masse der Dateien, die sich bei der Implementierung einer Klasse ergeben, für Funktionen von nicht mehr als *drei Zeilen* durchaus praktikabel. Allerdings muß bei dieser Verwendungsart der `inline`-Funktionen auf die korrekte Darstellung der Abhängigkeiten in dem entsprechenden Makefile (siehe Kapitel 7) geachtet werden!

Eine weitere Effizienzverbesserung des Codes und ein großer Gewinn an Sicherheit ist die Benutzung des Schlüsselwortes *const*, auf welches in den folgenden Abschnitten ausführlich eingegangen wird.

5.2 Deklaration von *const*-Funktionen und -Variablen

Mit dem Schlüsselwort *const* bietet C++ eine gute Methode, Funktionsrückgabewerte, formale Funktionsparameter sowie Variablen als *Konstanten* zu deklarieren.

5.2.1 Gegenüberstellung von *const* und #define

Die C-Deklaration

```
#define MAX 100 // In C
```

belegt die Konstante MAX mit dem Wert 100. MAX ist unter diesem Wert im *gesamten Programm* bekannt und kann nicht mehr mittels `#define` verändert werden. In C++ kommt man allerdings völlig ohne die Benutzung von `#define` aus, sofern man nur Konstanten deklarieren möchte. Das Analogon zur obigen `#define`-Definition in C++ ist

```
const int MAX = 100;// In C++
```

Ein großer Vorteil der Benutzung von const-Deklarationen ist die *Kontrolle über den Bezugsrahmen*, in dem MAX bekannt sein soll. Mit `#define` deklarierte Konstanten sind nach dem Aufruf der Funktion, in der sie definiert sind, überall im Programm bekannt, d.h. der *Bezugsrahmen* der Konstanten in C ist grundsätzlich *global* und kann u.a. zu Namenskonflikten führen. Eine const-Deklaration innerhalb einer Funktion ist nur *in der Funktion selbst* bekannt; eine Konstante gleichen Namens kann in jeder anderen Funktion noch einmal deklariert werden. *Daher sollen Konstanten, die nur von* einem *Programmierer benutzt werden, grundsätzlich mit* `const` *und solche, die global bekannt sein sollen, immer mit* `#define` *deklariert werden.*

5.2.2 Formale *const*-Parameter von Funktionen

Eine andere Möglichkeit der Benutzung des Schlüsselwortes *const* ist die Deklaration formaler Funktionsparameter als *const*-Parameter:

```
Bignum::Bignum operator+(const Bignum& lval,const Bignum& rval);
```

Die Variablen `lval` und `rval` dürfen mit dieser Deklaration innerhalb der Funktion `operator+()` *nicht mehr verändert werden.* Dies ist natürlich sinnvoll, da die Berechnung der Summe von lval und rval keine Veränderung dieser Variablen erfordert. Somit verursacht ein Programmierer, der an dieser Funktion etwas verändert und - evtl. „aus Versehen" - versucht, Operationen auf diesen Variablen durchzuführen, einen Compilerfehler. *Wann immer also klar ist, daß Variablen, die als formaler Funktionsparameter übergeben wurden, nicht mehr manipuliert werden dürfen, sollten sie als* const-*Variablen deklariert werden.*

5.2.3 *const*-Funktionen

Es gibt bei der in Kapitel 2 vorgestellten allgemeinen Klassenstruktur Funktionen (z.B. Ein/Ausgabe-Funktionen), die keine Membervariablen verändern. Ein Steuerungsmechanismus, der einen Compilerfehler provoziert, wenn in einer Funktion eine Membervariable verändert wird, wäre von Vorteil, da damit verhindert würde, daß sich in bestimmten Situationen eine Membervariable verändert, ohne daß dies vom Designer der Klasse gewünscht war. Dieses Ziel erreicht man durch das Nachstellen des Schlüsselwortes *const* hinter den Funktionskopf:

```
char *Bignum::desc() const
{
  return &classname;
}
```

Die Funktion `desc()` gibt nur den Namen der Klasse zurück, soll aber z.B. an der Variable `b` der Klasse `Bignum` nichts ändern. Eine Zuweisung innerhalb der Funktion `desc()`

```
*b = 2; // Fehler,b ist Membervariable !!!
```

wäre also nicht erlaubt. *In den Situationen, in denen sich innerhalb einer Elementfunktion einer Klasse nichts verändern darf, sollte* `const` *also immer dem Funktionskopf nachgestellt werden.* Im Beispiel der Funktion `desc()` bietet sich außerdem noch die Möglichkeit an, die Variable `classname` als *Konstante* zurückzugeben:

```
const char * Bignum::desc() const;
```

Falls die von einer Funktion zurückgegebene Variable nicht verändert werden soll, kann man durch die *const*-Typisierung der Funktion für diesen Fall einen Compilerfehler provozieren.

5.3 Benutzung von vordeklarierten Konstanten

Sowohl in C als auch in C++ sind eine große Anzahl von „arithmetischen Konstanten" bereits nach dem ANSI-Standard vordeklariert. Beispielsweise benötigt man beim Programmieren oft Konstanten wie die Maximalgröße für einen `long`-Wert oder die größte darstellbare Zehnerpotenz als `double`-Zahl. Diese Konstanten sind vordeklariert in der Headerdatei `limits.h`, die jeder C- und C++-Compiler zur Verfügung stellt (siehe auch [3], S.28-30). *Um Doppeldeklarationen und Namenskonflikte zu vermeiden, sollten diese Konstanten aus* `limits.h` *benutzt werden.*

5.4 Referenztypen

C++ bietet als neuen Sprachbestandteil den *Referenztyp*, der schon bei Sprachen wie PASCAL und ALGOL bekannt war. Damit wird es wieder möglich, Funktionsargumente per *Referenz* zu übergeben.

Beispiel: Die Berechnung

```
int x;
x = x * x;
```

kann man in C durch eine Funktion `square()` durchführen, die folgendes Aussehen hat:

```
void square(x)
int *x;
{
*x = *x * *x;//x wird veraendert
}
```

Ein eventueller Benutzer dieser Funktion muß immer gewährleisten, daß beim Aufruf der Funktion der formale Parameter (`x`) den *richtigen Typ* hat. Ein korrekter Aufruf dieser Funktion von einem Benutzer würde also so aussehen:

```
 int x;
 square(&x);// Adresse von x
//oder
 int *x;
 x = new int;
 square(x);
```

In C++ muß man nur die Deklaration der formalen Parameter von `square()` wie folgt abändern, um zu erreichen, daß durch eine einfache Übergabe einer `int`-Variable diese geändert wird:

```
void square(int& x)
{
  x = x * x;//x wird hier ebenfalls veraendert
}
```

Die Benutzung von Referenztypen impliziert also eine *automatische Dereferenzierung* von Zeigern. *Da durch den Gebrauch der Referenztypen die Notation des Aufrufs solcher Funktionen vereinheitlicht wird, ist es sinnvoll, in diesen Situationen Referenztypen zu benutzen.*

5.5 Speicherklassen

C++ stellt, wie C, die Speicherklassen *extern, register, static* und *automatic* zur Verfügung. Im folgenden werden nicht alle Eigenschaften dieser Speicherklassen dargestellt, sondern nur solche, die zu einem effizienteren Programmierstil beitragen.

5.5.1 *register*-Variablen

Bezeichnung 5.1 *Eine als* **register** *deklarierte Variable wird, wenn möglich, in das Maschinenregister geschrieben. Sie kann nur in einem lokalen Bezugsrahmen (Funktionen) deklariert werden.*

Die Deklaration erfolgt einfach durch Voranstellen des Schlüsselwortes *register* vor die Typbezeichnung der Variable:

```
f()
{
  register int i;
}
```

Falls genügend Kapazität im Maschinenregister vorhanden ist, steht `i` zur Laufzeit im Maschinenregister, was bei stark benutzten Variablen (Laufvariablen, Zeiger etc.) einen großen *Laufzeitgewinn* und *kleineren Code* zur Folge hat. Ist die Benutzung von *register* an einer Stelle sinnlos oder nicht erlaubt - wie z.B. bei voller ,,Auslastung" des Maschinenregisters - , so wird das Schlüsselwort vom Compiler einfach ignoriert. Die einzige Ausnahme ist der Zugriff auf die Adresse einer `register`-Variable. *Ausdrücke der Form*

```
  ...
register int i;
square(&i);
```

sind auf jeden Fall zu vermeiden! Diese Ausdrücke verursachen zwar keinen Compilerfehler, sind jedoch lt. [7] sehr kritisch. *Für „unkritische" Variablen in Funktionen sollte daher immer* register *benutzt werden.* (Siehe dazu auch [7], S.81,192).

5.5.2 *static*-Members

In unserem Zusammenhang soll nur auf die folgende Eigenschaft von *static*-Variablen (siehe auch [7], S.81) eingegangen werden: Bei der Definition der Klasse `Bignum` wurde die Deklaration

```
static char *classname="Bignum";
```

benutzt. Die Benutzung des Schlüsselwortes *static* an dieser Stelle hat zur Folge, daß sich alle Instanzen der Klasse der Klasse Bignum den Speicher für den Klassennamen „teilen“. Die Variable `classname` wird nur einmal deklariert, egal wieviele Instanzen von `Bignum` es gibt. *Alle Members einer Klasse, die für alle Instanzen einer Klasse nur einmal instanziiert werden sollen, sollten daher immer als* static-*Variablen deklariert werden.* Beim Gebrauch von `static` ist zu beachten, daß *Konstruktoren bzw. Destruktoren niemals* `static` *deklariert werden dürfen!* (siehe auch [3] S.262).

5.6 Die Operatoren *new* und *delete*

Der Gebrauch von `new` und `delete` ist im Gegensatz zu den Funktionen `malloc` und `free` in C wesentlich einfacher und allgemeiner benutzbar. Eine typische Allokierung von Speicher für einen Zeiger auf `Bignum` via `malloc` könnte in C z.B. so aussehen (siehe auch [5]):

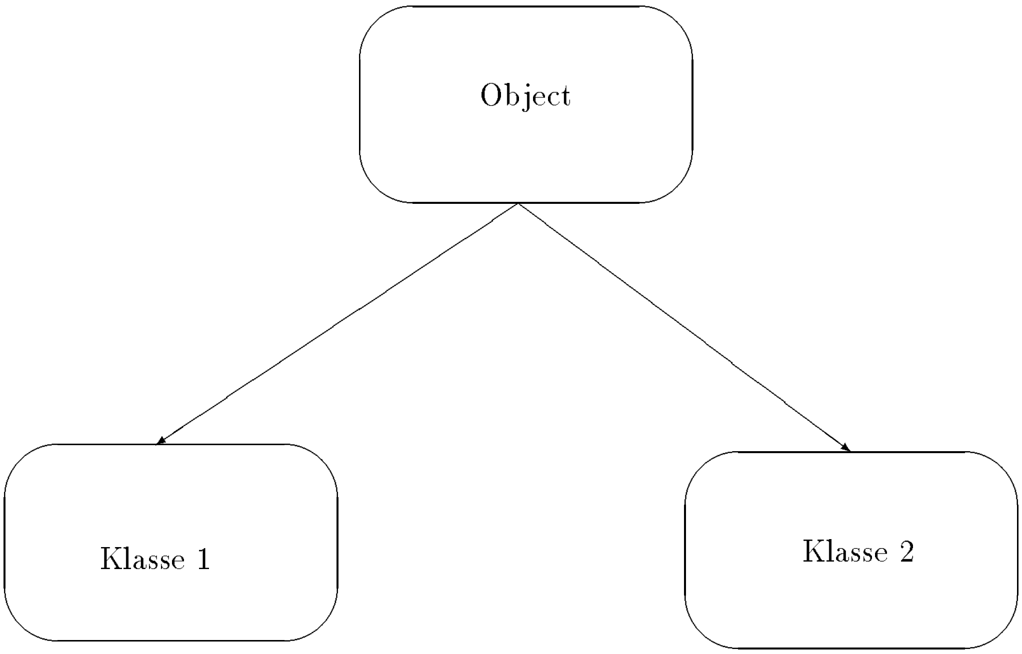
```
Bignum *a;
a = (struct Bignum *) malloc(sizeof(struct Bignum));
```

Das Casting und die Berechnung der Strukturgröße fällt bei der Benutzung von `new` weg :

```
Bignum *a;
a = new Bignum;
```

Sobald also Speicherallokierung und Deallokierung innerhalb von C++-Code (Konstruktoren, Destruktoren) erfolgen soll, ist die Benutzung von `new` *und* `delete` *verbindlich.* Dies hat vor allem den Grund, daß in einer Klassenbibliothek in C++ eine für alle Klassen zwingend zu erbende allgemeine Basisklasse `Object` entwickelt werden wird, die u.a. die Operatoren `new` und `delete` mit einer effizienten Speicherverwaltung überlädt. Bei der Tagung in Schloß Dagstuhl wurde eine Schnittstelle erarbeitet, mit deren Hilfe es möglich werden soll, beliebige Speicherverwaltungskonzepte *ohne Aufwand* in ein bestehendes Software-System einzufügen. Da jedoch keine Speicherverwaltung so allgemein sein kann, alle auftretenden Situationen effizient abzudecken, kann es im Einzelfall durchaus sinnvoll sein, mit Hilfe der ,,klassischen” C-Funktionen `malloc` und `free` lokal eine eigene Speicherverwaltung zu definieren. Ist dies aber nicht der Fall, *so sollten standardmäßig alle Speicherallokationen mit* `new` *und* `delete` *durchgeführt werden, um die Vererbung der Überladung von* `new` *und* `delete` *beim Einsatz der Klasse* `Object` *zu gewährleisten:*

Abb.2: Die Klasse `Object`



5.7 Überladen von binären Operatoren

Wie in Kapitel 2 schon angedeutet, ist ein wesentlicher Bestandteil von OOP das *Überladen von Operatoren.* Im folgenden wird gezeigt, wie bei der Definition der Operatoren vorgegangen werden muß und es werden einige ,,Ungereimtheiten", die in der Literatur vorkommen, aufgedeckt. Begonnen wird mit dem Überladen von *binären Operatoren*; das Überladen von unären Operatoren wird im Abschnitt „Überladen der Ein/Ausgabe-Operatoren“ behandelt.

In C++ gibt es zwei Techniken, um binäre Operatoren wie `+`, `-`, `*` oder `%` zu überladen.

1. die Operator-Funktion wird als *Elementfunktion* implementiert,
2. die Operator-Funktion wird als *friend-Funktion* implementiert.

Als Elementfunktion sähe dann die `operator+()`-Funktion am Beispiel von `Bignum` etwa so aus:

```
inline Bignum Bignum::operator+(const Bignum& rval)
{
```

```
  Bignum sum;// Ergebnisvariable

  add(b,rval,sum);// sum = b + rval
// b ist die implizite bekannte Membervariable der Klasse
// Bignum

  return(sum);
}
```

Mit dieser Definition und einem überladenen Gleichheitsoperator (siehe Überladen des Gleichheitsoperators) wird ein Ausdruck der Form

```
Bignum a;
a = a + 10;
```

möglich. Die Semantik dieses Ausdrucks in C++ ist aber in Wirklichkeit:

```
Bignum::operator=(Bignum::operator+(Bignum(10)))
```

Wie man sieht, muß die Zahl 10 von dem `Bignum`-Value-Konstruktor zunächst klassenintern in den Typ `Bignum` *konvertiert* werden, bevor die Addition durchgeführt werden kann. Die linke Seite (die implizit bekannte Variable `a`) ist bereits vom adäquaten Typ `Bignum`, so daß damit die Berechnung stattfinden kann. Der entscheidende Nachteil dieser Technik zeigt sich dann, wenn ein Benutzer der Klasse `Bignum` einen Ausdruck der Form

```
Bignum a;
a = 10 + a;//Compilerfehler!!!
```

schreiben möchte. Dieser Ausdruck verursacht einen Compilerfehler, da die Definition von Operator-Funktionen als Elementfunktionen die linke Seite immer als bereits „passend konvertierten" Operanden erwartet. Dies hat zwei Konsequenzen:

- Die Allgemeinheit geht verloren, d.h. ein Benutzer muß auf die *Reihenfolge* achten, in der er Variablen miteinander verknüpft.
- Die Verknüpfung von einem Objekt der Klasse `Bignum` mit einem Objekt eines anderen Datentyps wird sehr schwierig, da das einzig mögliche Argument einer als Elementfunktion implementierten Operator-Funktion den Typ der Klasse haben muß.

Diese Schwierigkeiten kann man sehr gut vermeiden, indem man die an den `+`-Operator gekoppelte Funktion als Freund der Klasse `Bignum` definiert:

```
friend operator+(Bignum& lval, Bignum& rval);
```

Mit dieser Technik erhält man - im Gegensatz zur Definition der Operator-Funktionen als Elementfunktionen - einige entscheidende Vorteile:

- Die Operator-Funktionen, die i.a. „public" sind, dürfen nur von Elementfunktionen der Klasse `Bignum` und der Klassen explizit aufgerufen werden, deren Instanzen mit Instanzen von `Bignum` verknüpft werden dürfen. Damit wird die Philosophie der Verkapselung aufrechterhalten.
- Die Parameterliste von Freund-Operator-Funktionen erlaubt im Gegensatz zu den Operator-Elementfunktionen *zwei* Parameter unterschiedlichen Datentyps.

Die Verknüpfung einer langen Zahl mit einer langen Floatingpoint-Zahl bereitet jetzt keine Schwierigkeiten mehr. Um dies zu realisieren, implementiert man einfach folgende Freund-Funktion:

```
friend operator+(const Bignum& lval,const Bigfloat& rval);
```

Natürlich darf hier nicht vergessen werden, *alle* „Verknüpfungsrichtungen" zu berücksichtigen. Sowohl der Ausdruck

```
Bigfloat a;
Bignum b;
a = a + b;
```

als auch der Ausdruck

```
a = b + a;
```

sollten möglich sein. Dies erreicht man jedoch einfach, indem man die formalen Parameter der Funktion `operator+()` in der Reihenfolge vertauscht und dieselbe Funktion nochmals implementiert:

```
friend operator+(const Bigfloat& lval,const Bignum& rval);
```

Hierbei ist nur zu beachten, daß die Vorwärtsdeklarationen der Freund-Funktionen sowohl in `Bigfloat` *als auch in* `Bignum` *vorkommen müssen.*

Obwohl es in C++ möglich ist, die Typkonvertierungsoperatoren wie z.B. `(char *)`, `(int)`, aber auch `(Bignum)` zu überladen, ist es einfacher und wesentlich sicherer, die Code-Redundanz, die bei einer doppelten Implementation ensteht, hinzunehmen. Das Überladen dieser „Konvertierer" ist oft mehrdeutig und damit sehr schwer zu kontrollieren. (Siehe dazu auch [2],S.300). Im Kapitel 2 wurde allerdings eine Implementationsmöglichkeit des Value-Konstruktor erläutert, die die doppelte Implementierung unnötig macht. *Wird dieser Konstruktor in der angegebenen Weise realisiert, so muß die Operatorfunktion, die einen formalen Parameter vom Typ* `int` *besitzt nur einmal implementiert werden.*

Operator-Funktionen müssen immer als `friend`*-Funktionen* definiert werden.

5.7.1 Überladen des Gleichheitsoperators

In der Zuweisung

```
a = b + a;
```

wurde der `=`-Operator verwendet. Da dieser natürlich zunächst nur Standarddatentypen miteinander verknüpfen kann, muß er ebenfalls überladen werden. Die `operator=()`-Funktion hat eine spezielle Syntax und kann als einzige Funktion, die binäre Operatoren überlädt, als *Elementfunktion* der Klasse implementiert werden. Bevor nun auf die Implementierung der `operator=()`-Funktion eingegangen wird, wird noch der sog. *this*-Pointer eingeführt: Dieser Zeiger wird in jeder Elementfunktion von C++ automatisch deklariert, ohne daß diese Deklaration für den Programmierer sichtbar ist. Im Falle `Bignum` ist er in den Elementfunktionen als

```
Bignum *this
```

implizit enthalten und zeigt immer auf die aktuelle Instanz des Objektes, welches ebenfalls implizit in der Funktion bekannt ist. Die Bedeutung dieses Zeigers wird nun klar in der Definition der `operator=()`-Funktion :

```
void Bignum::operator=(Bignum& rval)
{
  if(this == &rval) return;// Behandelt Zuweisungen x=x korrekt
  delete b;// b initialisieren
// Allokiere Platz fuer die linke Seite
// Kopiere Ziffern
}
```

In dieser Funktion ist die Variable `b` die *implizit* in allen Elementfunktionen (in diesem Falle in der Funktion `operator=()`) bekannte Membervariable der Klasse `Bignum`. *Auch der Gleichheitsoperator muß mehrmals für die jeweils erforderlichen Typen der rechten Seite überladen werden, da man evtl.*

z.B. einer Bignum-Zahl eine ganzzahlige Bigfloat-Zahl zuweisen möchte. Die `operator=`-Funktionen funktionieren also umgekehrt wie der Copy-Konstruktor. Durch `delete b` wird der Speicher für die Zwischenergebnisse, den der Copy-Konstruktor bei der Auswertung der binären Operationen allokiert hat, zunächst freigegeben, um dann das Ergebnis der Operationen korrekt in `b` zu speichern. Der Copy-Konstruktor und die `operator=()`-Funktionen arbeiten also Hand in Hand. Bei einer Nichtdeklaration des Copy-Konstruktors und der Überladungsfunktion für den Gleichheitsoperator werden diese Funktionen von C++ *automatisch* deklariert. Dies führt bei Klassen, deren Members eine variable Länge haben, zu sehr kritischen Programmsituationen. Die Klasse `Bignum` ist dafür ein gutes Beispiel. Dort hat die eigentliche Darstellungs-Komponente variable Länge:

```
class Bignum
{
  unsigned long* b;
}
```

Eine Programmsequenz

```
Bignum a;
Bignum b;
// Wertzuweisungen
a = b;
```

hat bei der Nichtdeklaration des Copy-Konstruktors die Semantik:

```
a.b = b.b;
```

Das heißt der Wert von `b` wird nicht in `a` kopiert, sondern sowohl `a` als auch `b` zeigen auf dieselbe Instanz der Klasse `Bignum`. *Dies bedeutet also, daß der automatisch von C++ generierte Copy-Konstruktor nur* elementweise Initialisierungen *vornimmt, was schon bei einer Schachtelungstiefe von 1 zu extremen Laufzeitfehlern führen kann!* Wäre die Variable `unsigned long* b` von einem Standarddatentyp fester Länge, wäre kein Laufzeitfehler zu befürchten. *Innerhalb einer Klassenhierarchie sollte aber dennoch immer der Copy-Konstruktor definiert werden, da der von C++ automatisch generierte Copy-Konstruktor immer* public *ist.* Ein weiterer häufig auftretender Programmierfehler beim Überladen von binären Operatoren ist die Rückgabe einer Referenz beim Verlassen der Operator-Funktion. Die Definition

```
Bignum& operator+(Bignum& lval, Bignum& rval)
{
      ...
}
```

verhindert den Aufruf des Copy-Konstruktors. *Der Copy-Konstruktor wird nur aufgerufen, wenn das Zwischenergebnis der Rechnung noch gerettet werden muß. In diesem Falle signalisiert der Referenzoperator, daß bereits Speicher für das Ergebnis vorhanden ist und der Copy-Konstruktor keinen mehr anlegen muß bzw. die Variable bereits initialisiert ist.*

5.7.2 Modularität in Operator-Funktionen

Ziel des objektorientierten Ansatzes ist u.a. die *Austauschbarkeit von Algorithmen.* Die Operator-Funktionen stellen nur den *objektorientierten Überbau* zu den eigentlichen arithmetischen Funktionen dar. Algorithmen, die z.B. effiziente Multiplikationsroutinen realisieren, sind oft in C oder Assembler geschrieben und u.U. an anderen Instituten entwickelt worden. Würde man, wie in unserem obigen Beispiel, die Addition in der `operator+()`-Funktion explizit durchführen, wäre es sehr schwierig, sie durch einen u.U. besseren Algorithmus zu ersetzen. *Daher dürfen in den Operator-Funktionen nur Aufrufe von reinen C-Funktionen erfolgen, um die Rechnung durchzuführen.* Dieser Ansatz läßt sich jedoch nicht in allen Fällen verwirklichen. Ausnahmen sind:

- Die arithmetischen Funktionen wurden bereits in C++ implementiert,
- die Algorithmen, die schon keine Grundarithmetik mehr sind.

Möchte ein Programmierer bereits für andere Klassen überladene Operatoren in seiner Klasse benutzen, darf und soll er dies natürlich tun. Der zusätzliche Arbeitsaufwand, jedesmal alle Grundarithmetiken, die bereits in C++ implementiert sind, neu zu programmieren, wäre nicht vertretbar. In der `operator+()`-Funktion wird also im wesentlichen der Aufruf einer Elementfunktion `add()` der Klasse Bignum erfolgen:

```
Bignum Bignum::operator+(const Bignum& rval,const Bignum& lval)
{
   ...

   add(lval,rval,sum);//sum = rval + lval
   ...
   return(sum);
}
```

Soll `add()` gegen eine „bessere" Funktion ausgetauscht werden, muß nur der Aufruf ausgewechselt , nicht aber die `operator+()`-Funktion neu geschrieben werden. Dies bedeutet, daß sich dadurch an den Benutzerprogrammen nichts verändert.

5.7.3 Allgemeine Vorgehensweise

Beim Überladen binärer Operatoren sind also folgende Regeln zu beachten:

- Alle im Anhang aufgelisteten binären Operatoren sind zu überladen.
- Alle binären Operator-Funktionen sind als *Freund-Funktionen* der Klasse zu implementieren. Dies gilt nicht für die Operator-Funktionen, die den Gleichheitsoperator überladen.
- Die formalen Parameter dieser Funktionen sind als `const`-Parameter zu deklarieren, da sie im Verlauf der Operation nicht verändert werden dürfen.
- In den Operator-Funktionen findet keine Implementation der eigentlichen Operationen statt (Modularität).
- Es muß eine Implementation aller Verknüpfungen, die mit Objekten einer Klasse möglich sein sollen, vorhanden sein.
- Der Gleichheitsoperator muß ebenfalls für die entsprechenden Objekttypen mehrmals überladen werden.
- Implementation der Operationen sollte weitestgehend in portablem C-Code erfolgen.

5.8 Überladen von unären Operatoren

Die Vorgehensweise beim Überladen unärer Operatoren wird am Beispiel der Operator-Funktionen deutlich gemacht, die die Ein/Ausgabe-Operatoren überladen. In Kapitel 2 wurde bereits darauf hingewiesen, daß die Ein/Ausgabe-„Funktionen“ in C++ (`cin` und `cout`) in Wirklichkeit Instanzen der C++-Standardklassen `istream` bzw. `ostream` sind. In diesen Streamklassen sind die Links-und Rechtsshiftoperatoren, `<<` und `>>`, bereits mit Ein/Ausgabe-Funktionen so überladen, daß sie in Verbindung mit Objekten vom Typ `istream` oder `ostream` eine formatierte Ausgabe einer Standardvariablen erzeugen können:

```
#include <stream.h> //Deklarationen von istream und ostream

main()
{
  int i;
```

```
  i = 1;
  printf("i ist gleich: %d\n", i);// In C
  cout<<"i ist gleich:"<<i ;       // In C++
}
```

Da `i` vom Standarddatentyp `int` war und `<<` für alle Standarddatentypen bereits überladen ist, benötigt der Compiler den Typ von `i` als Formatstring wie etwa `%d` nicht mehr. Wie wir schon gesehen haben, ist es in C++ möglich, Operatoren *beliebig oft* zu überladen. *Dabei ist nur darauf zu achten, daß der Compiler verschiedene Überladungen desselben Operators unterscheiden können muß.* Diese Unterscheidung findet durch das sehr strenge „Type-Checking" der formalen Parameter, Namen und Rückgabetypen der Operator-Funktionen statt. Die Überladungen

```
Bignum Bignum::operator+(Bignum& lval, Bignum& rval)
{
    // Implementation 1
}
Bignum Bignum::operator+(Bignum& lval, Bignum& rval)
{
    // Implementation 2
}
```

sind nicht möglich, da die Argumente beider Funktionen gleich sind und der Compiler damit kein Unterscheidungskriterium hat. Dieses Type-Checking wird im Falle der Objekte `cin` und `cout` zum Nachteil, wenn man daran interessiert ist, sich von einem Objekt *verschiedene* Formate ausgeben zu lassen:

```
int i;
printf("%d\n", i);// Zahl i dezimal
printf("%x\n", i);// Zahl i hexadezimal
printf("%o\n", i);// Zahl i oktal

// aber ...

cout<<i; //zahl i immer dezimal
```

In solchen Situationen ist es natürlich durchaus sinnvoll, weiterhin `printf` *oder* `scanf` *zu benutzen.*

In jedem Fall ist es auch möglich, die Operatoren `<<` und `>>` mit klassenspezifischen Ein/Ausgabe-Formaten zu überladen. Ein Benutzer der Klasse `Bignum` will und darf nicht Ausgaben der Form

```
Bignum a = 12345678;
int i;
printf("Die lange Zahl ist\n");
for(i = 0;a.b != NULL;i++)
{
  printf("%ld\n", a.b[i]);
}
```

implementieren. Dies wäre sehr fehleranfällig, unkomfortabel und darüberhinaus in unserem Beispiel gar nicht erlaubt, weil z.B. `Bignum::b` *private* deklariert ist. Die Alternative wäre eine Ausgabe der Form:

```
cout<<"Die lange Zahl ist:\n"<< a;
```

In der Klasse `Bignum` gibt es demnach eine Funktion `operator<<()`, die die formatierte Ausgabe für eine lange Zahl implementiert. Wie eine solche Funktion *genau* aussieht, ist der Implementation der Klasse zu entnehmen. An dieser Stelle sei nur erklärt, wie sie funktioniert. `operator<<()` hat zwei Argumente, die lange Zahl, die ausgegeben werden soll und einen Parameter vom Typ `ostream`:

```
friend ostream& operator<<(ostream&,const Bignum &zahl)
{
  zahl.print(stream);//Ruft die Ausgaberoutine auf
  return(stream);
}
```

Die Funktion `Bignum::print()` hat folgendes Aussehen :

```
void Bignum::print(ostream& stream)
{
  for(i = 0;i < len;i++)
  {
    stream<<(LONGTYPE)b[i];// Gibt den Standarddatentyp aus
  }
}
```

Für die interne Repräsentation (Feldelement des `long`-Feldes) der ganzen langen Zahlen ist der Operator `<<` natürlich bereits überladen und kann in Verbindung mit einer Instanz der Klasse `ostream` benutzt werden. Näheres zum Überladen der Operatoren `<<` und `>>` mit einem sehr guten Beispiel findet man in [5] S.81-89, sowie in [3]. Die Überladung des `>>`-Operators ist etwas komplizierter, ist aber in [5] Kap.5 ebenfalls sehr gut dargestellt. Die Klasse `istream` bietet dem Programmierer darüberhinaus eine Fülle von Funktionen für die Fehlerbehandlung, um fast alle beim Einlesen von Variablen möglichen Fehler abfangen. *In jeder neu zu definierenden Klasse sollten daher die Links- und Rechtsshiftoperatoren mit komfortablen Ein/Ausgabe-Funktionen überladen werden, wobei es in Situationen wie in obigem Beispiel nach wie vor sinnvoll bleibt,* `printf` *und* `scanf` *zu benutzen.* Weitere unäre Operatoren, die in Klassen wie `Bignum` überladen werden sollten, wie z.B. `++` und `--`, sind im Anhang aufgelistet.

5.9 Initialisierungslisten

Es tritt häufig die Situation auf, daß Members einer Klasse Instanzen einer anderen Klasse sind:

```
class X
{
  Y a;
  X() { a = 0;}
}
class Y
{
  int b;
  Y();
  Y(const Y&);
  Y(int);
  operator=(y&);
}
```

Die Zuweisung

```
X a;
X b;
b = 200;
a = b;
```

impliziert mit den obigen Klassendefinitionen die folgenden Konstruktoraufrufe:

1. Aufruf des Konstruktors `X()`, um `X` zu instanziieren,
2. Aufruf des Konstruktors `Y(int)`, um die Zahl 200 in den Typ Y umzuwandeln,

3. Aufruf des Copy-Konstruktors `Y(const Y&)` um das Ergebnis zu retten,

4. Aufruf der `operator=Y(Y&)`-Funktion, um das Ergebnis zuzuweisen.

Den ersten Schritt hätte man vermeiden können, wenn man die Klasse `X` wie folgt definiert hätte:

```
class X
{
  Y a;
  X() : Y(a);
      ...
}
```

Mit der Übergabe von `a` im Funktionskopf des Klassenkonstruktors als Argument des Klassennamens von `Y` umgeht man die Initialisierung in der Klasse `X`. Hinter dem Funktionskopf könnten auch mehrere Members sukzessive initialisiert werden. Diese „Initialisierungsliste" heißt auch *Memberinitialisierungsliste*. In diesem einfachen Beispiel fällt die verlorengegangene Laufzeit nicht sehr ins Gewicht, weil die Initialisierung des Members `Y a` nur eine Zuweisung beinhaltet. Bei komplizierteren Konstruktoren jedoch kann durch die Nichtbenutzung der Memberinitialisierungsliste sehr viel Zeit verloren gehen. *Daher sollte man immer, wenn ein Member in einer Klasse als Instanz einer anderen Klasse auftritt, die Technik der Memberinitialisierungsliste benutzen, um unnötige Aufrufe von Konstruktoren zu vermeiden.* Die genaue Darstellung eines komplizierteren Beispiels würde mehrere Seiten Platz benötigen, weswegen an dieser Stelle auf einige gut dargestellte Beispiele in [5] S.106,107,116,117 verwiesen sei.

Kapitel 6

OOP-Konzepte in C++

6.1 Zielsetzungen

In diesem Kapitel werden einige typische konzeptionelle Probleme untersucht, die beim Design einer Klassenhierarchie auftreten können.

Ein wesentlicher Bestandteil des objektorientierten Programmierens ist das Konzept der *Vererbung* und das Ausnutzen der *dynamischen Bindung* (siehe Kapitel 2). Diese Sprachelemente von C++ sind in ihrer Konzeption einfach zu verstehen, bergen aber eine Vielzahl von Problemen, die bei der tatsächlichen Programmierung auftreten können. Dies liegt vor allem an den an den konzeptionellen Schwierigkeiten, die auftreten können, wenn Gemeinsamkeiten zwischen Objekten untersucht und ihre Eigenschaften an andere Objekte vererbt werden sollen. In den folgenden Abschnitten werden einige wesentliche Situationen vorgestellt, an denen aufgezeigt wird, worauf bei der Benutzung des Vererbungsmechnismus besonders zu achten ist und wo die eigentlichen Schwierigkeiten beim Design einer hierarchischen Anordnung von Klassen liegen. Die dort erarbeiteteten Methoden sind sehr gut als Hilfsmittel für das Design einer hierarchischen Anordnung von Klassen, die Objekte aus der Algebra repräsentieren, zu gebrauchen.

6.2 Vererbung und virtuelle Funktionen

Mit Hilfe des Vererbungsmechnismus können u.a. folgende Problemstellungen gelöst werden:

- Haben zwei Klassen dieselbe Menge von Elementfunktionen, so müssen diese nur einmal definiert werden, da sie vererbt werden können.

- Stimmen die Elementfunktionen zweier Klassen bis auf Ausnahmen überein, so müssen nur noch die unterschiedlichen Funktionen in der jeweils abgeleiteten Klasse implementiert werden.

Den Bezug zwischen ähnlichen Funktionen in einer Klassenhierachie wird durch das Schlüsselwort *virtual* hergestellt, wie das folgende Beispiel zeigt:

```
class X
{
  virtual int f(int)
  {
    ...
  }
}

class Y : public X // X vererbt an Y
{
      ...

  virtual int f(int);
  void test_f()
  {
    int x;
    x = f(x);
  }
      ...
}
```

Der Aufruf

```
x = f(x);
```

benutzt die in der Klasse `X` definierte Funktion `f()`. Die Klasse `Y` erbt also die Funktionsdefinition von `f()` in der Klasse `X` (siehe auch [3] S.208ff). *Die Benutzung virtueller Funktionen in unserem Zusammenhang setzt ein exaktes Verständnis der Zusammenhänge zwischen algebraischen Strukturen voraus.* Beispielsweise ist ein auf den ersten Blick naheliegender und scheinbar intuitiver Ansatz zur Gestaltung einer Teilhierachie der, daß man die Verknüpfungen, die in einem Ring möglich sein sollen, „definiert", und an die entsprechenden Zahlentypen weitervererbt. Mit diesem Ansatz erhielte man also folgende vereinfacht dargestellte Vererbungsstruktur:

```
Ring
  Bignum
  Integer
```

Damit wäre die Klasse `Ring` die Basisklasse der Zahlentypen `Integer` und `Bignum` und würde ihre Operationen und Elemente entsprechend vererben. Ein Ring ist allerdings eine Menge von Objekten mit zwei Verknüpfungen, die auf diesen Objekten definiert sind, d.h. die Struktur Ring kann nicht nur durch die Verknüpfungen oder nur durch die Darstellung seiner Elemente beschrieben werden, sondern ausschließlich durch beides zusammen. Dies hat beim Design einer Hierarchie algebraischer Objekte und Strukturen die Konsequenz, daß in der Klasse `Ring` sowohl die Elemente als auch die Verknüpfungen, die auf diesen Elementen definiert sind, implementiert werden müßten, um eine „abstrakte" Klasse `Ring` zu erhalten. In allen bereits existierenden Klassendefinitionen für ganze lange Zahlen, lange Floating-Point-Zahlen etc. wird aber bereits implizit in einer algebraischen Struktur wie Ring oder Körper gerechnet, da sonst die Operationen auf den Elementen gar nicht darstellbar wären. Die Operationen und Darstellungen der Elemente in dem abstrakten Datentyp (ADT) Ring zu definieren, würde `Ring` zu einer sehr großen Klasse machen, die alle Arithmetiken auf Zahlen, die in einem Ring möglich sind, in sich vereinigt. Dies hätte die Konsequenz, daß die bereits bestehenden Zahlenklassen zumindest sehr stark verändert werden müßten, damit die Klasse `Ring` den gewünschten Abtsraktionsgrad erreicht. An dieser Stelle wird schon klar, daß die streng hierachische Anordnung der in der Computeralgebra auftretenden Datentypen und Klassen ein großes Maß an theoretischer Vorarbeit und praktischer Implementierungsarbeit erfordert. Tatsächlich existieren aber die Arithmetiken in Ringen und Körpern bereits in den einzelnen C++-Zahlenklassen. Eine praktikable Lösung für das Zusammenfassen von diesen Klassen zu abstrakteren Datentypen muß daher zumindest die Wiederverwendbarkeit dieser z.T. sogar schon standardisierten Klassen gewährleisten. Trotz dieser Probleme gibt es allerdings Möglichkeiten, sowohl das Ziel, eine Klassenhierarchie zu designen als auch abstrakte Datentypen nach Bedarf zu entwickeln, nicht verwerfen zu müssen. Es haben sich bei der Programmierung sowohl an unserem Lehrstuhl als auch an anderen Instituten typische Problemstellungen herauskristalliert, die, wenn sie gelöst werden können, die Komplexität der oben angesprochenen Aufgabenstellungen stark vereinfachen können. In den folgenden Abschnitten wird an einer typischen Problemstellung aufgezeigt, wie in dem jetzigen Stadium der Entwicklung verfahren werden kann, um das eigentliche Ziel des Designs einer objektorientierten Klassenbibliothek für Anwendungen aus der Computeralgebra erreichen zu können.

6.3 Eine repräsentative Problemstellung

Eine für das Programmieren von Anwendungen aus der Computeralgebra typische Situation, die an unserem Lehrstuhl aufgetreten ist, stellt sich wie folgt dar:

Es soll eine Klasse `Matrix` definiert werden, die sowohl lange Zahlen als auch lange Floatingpoint-Zahlen als Einträge in der Matrixstruktur zuläßt. Dies setzt die Definition eines abstrakten Datentyps voraus, der die Arithmetik sowohl für lange ganze Zahlen als auch für lange Floatingpoint-Zahlen implementiert. Mit Hilfe dieser abstrakten Klasse (`Zahl`) hätte die Klasse `Matrix` etwa folgendes Ausehen:

```
class Matrix
{
  Zahl **matrix;// Zeilen, Spalten
       ...
}
```

Für das Design einer Klasse `Zahl`, die so allgemein ist, daß sie die obigen Anforderungen erfüllen kann, gibt es verschiedene Ansatzmöglichkeiten, von denen an dieser Stelle die zwei wichtigsten vorgestellt werden:

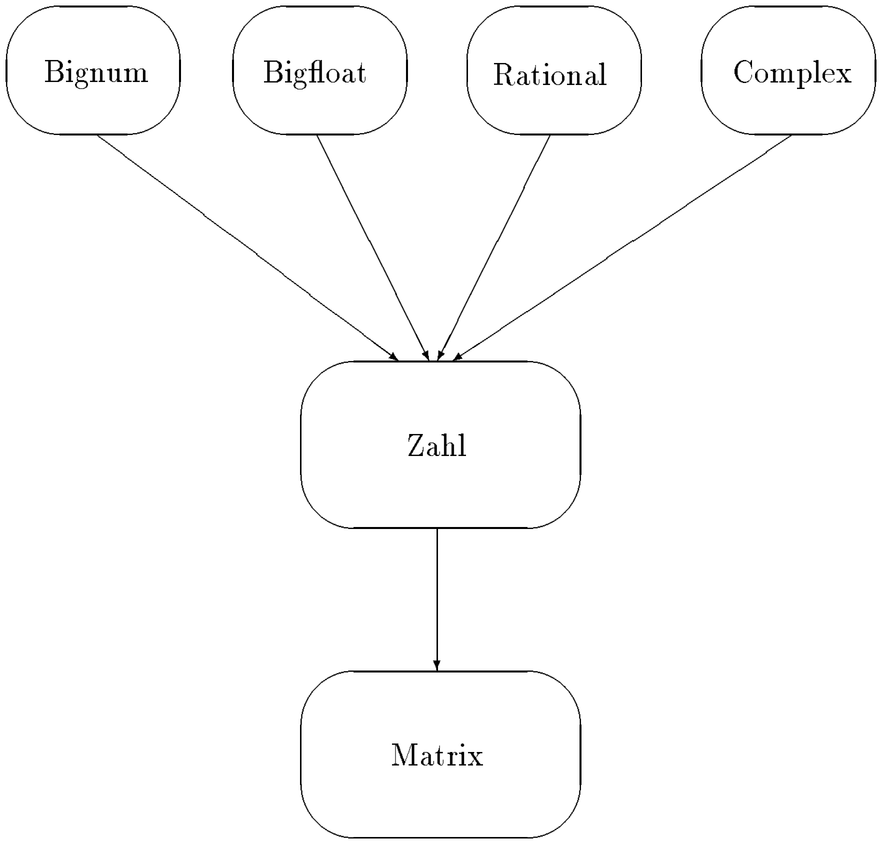
- das Mehrfachvererbungskonzept,
- das Zahlen-Container-Konzept.

6.4 Das Mehrfachvererbungskonzept

Der oben vorgestellte abstrakte Datentyp `Zahl` muß alle Operationen und Zahlentypen, die in der Klasse `Matrix` möglich sein sollen, in sich vereinigen. Dies würde bedeuten, daß in der Struktur `Zahl` verschiedene Zahlentypen und die Operationen auf diesen Zahlen implementiert werden müßten. Um dies zu umgehen, bietet es sich an, die Eigenschaften der Zahlenklassen, die die Klasse `Zahl` in sich vereinigen soll, zu vererben. Die Klasse `Zahl` würde also von mehreren Klassen die Arithmetiken und Zahlendarstellungen erben und hätte damit etwa folgendes Aussehen:

```
class Zahl : public Bignum : public Bigfloat ...
{
  ...
}
```

Abb.3 Mehrfachvererbung



Klassen wie `Bignum` sind aber bereits so kompliziert, daß die Kontrolle über Aufrufreihenfolge der virtuellen Funktionen sowie der Konstruktoren und Destruktoren nur sehr schwer zu erlangen ist (siehe [3] S.219ff). Auch gibt es zum Thema „Mehrfachvererbung" in Verbindung mit komplizierteren Klassen nur sehr wenige Veröffentlichungen und Standardliteratur, so daß ein Programmierer in einem solchen System zunächst viel Zeit damit verbringen müßte, Erfahrungen mit dem Mehrfachvererbungskonzept zu sammeln. Außerdem müßten während der Entwicklung die Basisklassen wie `Bignum` ständig geändert werden, um sie für dieses Konzept kompatibel zu machen, was einen nicht vertretbaren Programmieraufwand bedeuten würde. Darüberhinaus ist das Konzept der Mehrfachvererbung noch nicht für alle existierenden C++-Compiler verwirklicht, so daß seine Benutzung u.U. einen hohen Grad an Inkompatibilität gegenüber fremder Software schaffen würde. Dieses Konzept ist damit für unsere Zwecke, ein praktikables, sicheres und kompatibles System zu schaffen, nicht geeignet. Ein weiterer entscheidender Nachteil, der bei der Benutzung virtueller Funktionen ensteht, wird im folgenden Abschnitt behandelt.

6.5 Probleme bei der Benutzung virtueller Funktionen

Virtuelle Funktionen zeichnen sich dadurch aus, daß sie in verschiedenen durch Vererbung zueinander in Beziehung stehenden Klassen deklariert oder definiert sind. Eine in der Klasse `Bignum` definierte Additionsfunktion für lange Zahlen soll in der Klasse `Zahl` natürlich nur *deklariert* werden müssen, da ja `Zahl` diese Funktionen und ihre Implementierung u.a. von `Bignum` erben soll. Bezogen auf die Additionsfunktion würden in den beiden Klassen u.a. folgende Deklarationen vorkommen:

```
class Bignum
{
    ...
  virtual add(LONGTYPE, LONGTYPE, LONGTYPE)
  {
    //Implementierung
    ...
  }
}

class Zahl
{
    ...
  virtual add(LONGTYPE, LONGTYPE, LONGTYPE);
  //Deklaration
    ...
}
```

Im Falle der einfachen Vererbung würde ein korrekter Aufruf der Funktion `add()` in der Funktion

```
Zahl::operator+()
```

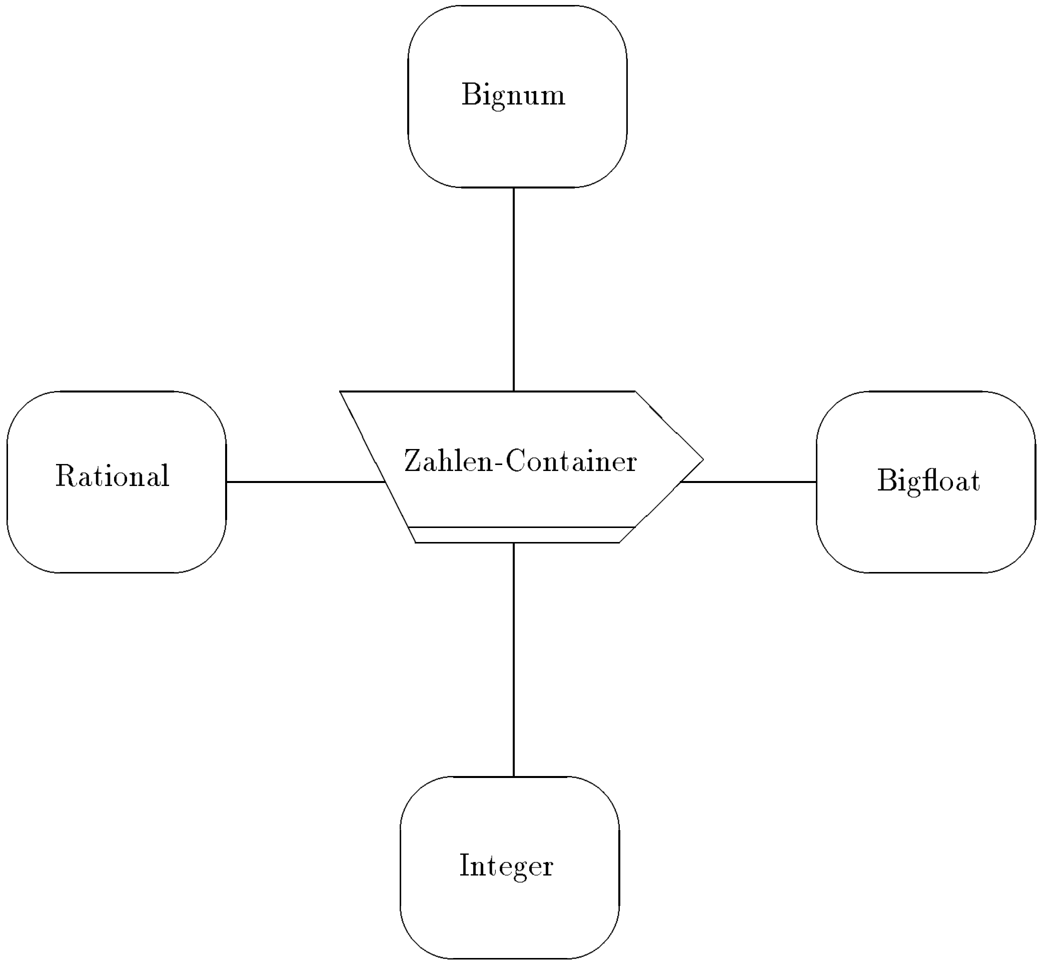
erfolgen, allerdings hätte man in diesem Falle nichts an Allgemeinheit gewonnen, da dann `Zahl` nur ein Synonym für `Bignum` wäre. Wollte man zusätzlich die Eigenschaften des Datentyps `Bigfloat` an die Klasse `Zahl` vererben, so würde eine weitere `add()`-Funktion der Klasse `Bigfloat` hinzukommen, wobei die Parameter dieser Funktion von anderem Typ wären als die Parameter der `Bignum::add()`-Funktion.

Damit ist schon die kritische Stelle beim Design der Klasse `Zahl` mit Hilfe des Mehrfachvererbungskonzeptes erreicht. *Damit der* virtual-*Mechanismus überhaupt funktioniert, müssen die entsprechenden Funktionen in der Basisklasse und in der abgeleiteten Klasse* Typübereinstimmung *sowohl bei ihren Parametern, als auch ihren Rückgabetypen gewährleisten.* Dies wäre jedoch nur zu erreichen, wenn alle Zahlentypen klassenintern *dieselbe Repräsentation* hätten, was aber nicht der Fall ist. Dieser Zustand wäre zwar theoretisch zu erreichen, ist aber für unsere Zwecke unangemessen aufwendig. Dazu kämen noch die oben angesprochenen Probleme bei der Benutzung des Mehrfachvererbungskonzeptes. Eine wesentlich weniger aufwendige Lösung bietet das „Zahlen-Container-Konzept", welches im nächsten Abschnitt besprochen wird.

6.6 Das Zahlen-Container-Konzept

Die Grundidee dieses Konzeptes ist die Vereinigung bereits existierender Zahlentypen in einer Klasse `Zahl` *ohne Benutzung virtueller Funktionen.* `Zahl` wird so zu einem *Container* aller bereits existierender Zahlentypen wie `Bignum`, `Bigfloat` etc. . Wie beim Mehrfachvererbungskonzept soll der Zahlen-Container alle Arithmetiken und Repräsentationen bereits existierender Zahlentypen in sich vereinigen. Im Bild läßt sich dieses Konzept wie folgt darstellen:

Abb.4 Containerklasse Zahl



Damit würde es beispielsweise möglich werden, die Klasse `Matrix` zu entwickeln, die als Einträge einen abstrakten Datentyp `Zahl` hat und somit z.B. sowohl lange Zahlen, als auch lange Floatingpointzahlen als Einträge zuläßt. Das Problem besteht darin, einen Weg zu finden, die Eigenschaften der bereits existierenden Klassen auszunutzen, ohne den Vererbungsmechanismus zu benutzen. Um dieses Ziel zu erreichen, deklariert man in der Klasse `Zahl` die Members wie folgt:

```
class Zahl
char *wert;//Wert der Zahl als char
// Hier sind eventuell noch andere Typen moeglich!!!
```

```
        ...

Bignum *lzahl;
Bigfloat *lfloat;
      ...
```

Die Repräsentation eines Objektes vom Typ `Zahl` wäre damit innerhalb der Klasse `Zahl` in Wirklichkeit vom Typ eines der so deklarierten Members. Den Konstruktor der Klasse `Zahl` würde man dann wie folgt realisieren:

```
Zahl()
{
  //Kopiere wert nach lzahl
  lzahl = new Bignum(<Wert>);
  // Wenn Zahl eine lange Zahl ist

  //kopiere wert nach lfloat
  lfloat = new Bigfloat(<Wert>);
  // Wenn Zahl eine lange Floatingpointzahl ist

  // Andere Zahlentypen
}
```

Mit der Zuweisung `lzahl = new Bignum(<Wert>)` wird sofort die Kontrolle an den Konstruktor der Klasse `Bignum` übergeben, d.h. der Programmierer der Klasse `Zahl` kümmert sich nicht weiter um Details, sondern verläßt sich auf die korrekte Realisierung des `Bignum`-Konstruktors. Man sieht hier sofort, daß eine Funktion `check_type()` benötigt wird, die zunächst feststellt, welchen Typs ein von einem Benutzer als `Zahl` deklariertes Objekt wirklich ist. Die einzelnen Zahlentypen sind jedoch aufgrund ihrer Darstellungen recht gut zu unterscheiden. Zum Beispiel ist eine Zahl mit 100 Vorkommastellen und einigen Nachkommastellen leicht als Objekt vom Typ `Bigfloat` zu identifizieren. Ist das Objekt noch nicht mit Werten belegt, kann die Typprüfung allerdings nicht auf diese Weise stattfinden. Bei einer Zuweisung

```
a = b + c;
```

wobei `a` vom Typ `Zahl` `b` und `c` vom Typ `Bignum` sind, muß die Typprüfung für `a` den Typ `Bignum` ergeben. Das heißt, die Funktion `check_type()` muß in einem solchen Falle die Typprüfung in Abhängigkeit der Operandentypen durchführen. Ist diese Typprüfung erfolgt, wird je nach Ergebnis der entsprechende Konstruktor der bereits existierenden Zahlentypen aufgerufen. Der entscheidende Vorteil der klasseninternen Repräsentation eines Objektes vom

Typ Zahl ist die nun die *implizit bekannte* Arithmetik der einzelnen Zahlenklassen. Konstruiert man nun die Operator-Funktionen der Klasse `Zahl`, so kann auf bereits überladene Operatoren , z.B. der Klasse `Bignum`, zurückgreifen. Die Operator-Funktionen von `Zahl` sehen damit etwa wie folgt aus:

```
Zahl operator+(Zahl& lval, Zahl& rval)
{
  Zahl summe;
  (Bignum)summe = (Bignum)lval + (Bignum)rval);
  // Wenn Zahl eine lange Zahl ist
  // Ruft Bignum::operator+(Bignum&,Bignum&) auf
  return (Zahl)(summe);
}
```

In diesem Beispiel wird der in `Bignum` bereits überladene `+`-Operator benutzt.

Jenachdem welche interne Repräsentation in der Funktion `check_type()` für das aktuelle Objekt gefunden wurde, werden die Argumente der Operator-Funktion „gecastet". Die Klasse `Zahl` hat also in gewisser Weise die Eigenschaften der Klasse `Bignum` „geerbt". Nun ist eine Klasse `Matrix`, die alle möglichen Datentypen als Einträge zuläßt, einfach zu realisieren:

```
class Matrix
{
  Zahl **matrix;// siehe oben
      ...
}
```

Die Matrixoperationen werden so mit einer Definition für alle Kombinationen von Zahlentypen, die in `Zahl` vorkommen, definiert. Ähnliches gilt auch für Datentypen wie Polynome, Vektoren etc. . Der Nachteil des Zahlen-Container-Konzeptes liegt auf der Hand. Kommt ein neuer Zahlentyp hinzu, muß die Klasse `Zahl` in der Art überarbeitet werden, daß die Casts, die Konstruktoraufrufe und die Funktion, die den Typ der Zahl feststellt, entsprechend abgeändert werden. Diese Änderungen sind jedoch immer gleichartig und verlangen keine konzeptuellen Umstellungen der Klasse. Die Vorteile dieses Konzeptes sind:

- hoher Laufzeitgewinn aufgrund fehlender virtueller Funktionen (siehe [5] S.211),
- hoher Grad an Allgemeinheit, der nicht zu Lasten der Laufzeit geht,
- sehr gute Ausnutzung bereits bestehender Arithmetiken (durch die bereits überladenen Operatoren),

- großer Bestand an existierender Literatur, die die Situationen, die bei diesem Konzept auftreten können, ausreichend behandelt,
- größere Nähe zur Programmiersprache C, was die Einarbeitung neuer Programmierer wesentlich vereinfacht.

Mit diesem Konzept geht man also einen Kompromiß zwischen dem rein objektorientierten Ansatz und einer nicht objektorientierten Programmierung ein, die C++ nur als ein verbessertes C benutzt. Der Nachteil dieses Ansatzes ist allerdings die u.U. *aufwendige Wartung* der Klasse, da für jeden neu hinzukommenden Zahlentyp alle Typabfragen und Casts in der Klasse realisiert werden müssen. Wann immer ein Objekt vom Typ `Zahl` deklariert wird, muß erst die interne Repräsentation des Objektes festgestellt werden und in allen Operator-Funktionen das aktuelle Objekt passend gecastet werden. Was man also bräuchte, wäre eine Möglichkeit, der Klasse Zahl einen Parameter zu übergeben, der den Typ des aktuellen Objektes spezifiziert. Damit würde die Klasse Zahl nicht an Allgemeinheit verlieren und wäre wesentlich einfacher zu implementieren und zu warten.

6.7 Parametrisierbare Datentypen

In der Entwicklung von C++ wurde die Unvollständigkeit des reinen Vererbungskonzeptes nach einigen Compilergenerationen erkannt und ein neues Feature (*sog. Templates*) geschaffen, mit dem die oben beschriebenen Probleme, die bei der Konstruktion von Containerklassen auftreten, wesentlich besser behandelt werden können. Die Templates haben sich seit kurzem als Sprachbestandteil etabliert und sind in der AT&T-Version 3.0 des C++-Compilers bereits realisiert. Dieses Konzept weicht zwar vom rein objektorientierten Ansatz ab, läßt sich aber dennoch sehr gut mit dem Vererbungskonzept kombinieren. Die Funktionsweise von Templates wird im folgenden Abschnitt kurz beschrieben. Da zu diesem Konzept bisher nur wenig Literatur erhältlich ist und es nur wenige Erfahrungswerte gibt, wird es hier nur kurz vorgestellt und es wird auf seine wesentlichen Bestandteile hingewiesen. Die vorgestellten Programm-Listings sind nur Vorschläge für eine wirkliche Implementierung, da es bislang (mangels Compiler) noch keine Möglichkeit gab, dieses Feature in der Realität zu testen.

Bezeichnung 6.1 *Ein* **Template** *definiert eine Familie von Datentypen oder Funktionen.*

Wie wir im vorherigen Abschnitt gesehen haben, benötigen wir eine Möglichkeit, *die klasseninterne Repräsentation eines Objektes zu parametrisieren.*

Bezeichnung 6.2 *Ein* **Klassen-Template** *bestimmt generisch (durch Übergabe eines Parameters) den eigentlichen Typ einer Klasse.*

Unsere Container-Klasse `Zahl` hat mit der Benutzung des `template`-Features etwa folgendes Aussehen.

```
template<class T> class Zahl
{
  T* v;

   ...

  T operator+(T&, T&);
   ...
}
```

Das C++-Konstrukt

```
<class T>
```

ist in diesem Falle ein *Platzhalter* für einen Zahlentyp (z.B. `Bigfloat`). Die Deklaration eines Objektes vom Typ `Zahl` bestimmt nun *über einen Parameter* den klasseninternen Typ des Objektes:

```
Zahl<Bigfloat> a;
Zahl<Bignum>   b;
```

Damit fällt das aufwendige „Casting" in den Operator-Funktionen, Konstruktoren und Destruktoren einfach weg, da man in ihnen mit dem abstrakten Datentyp `T` rechnet, der von C++ klassenintern automatisch in den als Parameter übergebenen Datentyp „gecastet" wird (siehe [5] S.355ff und [9]). Ein Aufruf in einem Konstruktor der Klasse `Zahl` hat beispielsweise folgendes Aussehen:

```
class Zahl
{
  Zahl::Zahl()
  {
    ...
    interne_zahl = new Zahl;// Platz f. eine Zahl
    ...
  }
}
```

Wäre der klasseninterne Typ des Objektes

`Zahl::interne_zahl`

vom Typ `Bignum`, so würde der Aufruf von

`Zahl::Zahl()`

sofort den Konstruktor `Bignum::Bignum()` aufrufen. Somit wird die Kontrolle der Speicherallokierung an die Klasse `Bignum` übergeben. Auch die Code-intensiven Typbestimmungen und Verzweigungen der Operator-Funktionen, die beim Zahlen-Container-Konzept, das keine Templates benutzte, auftraten, würden bei dieser Methode wegfallen, so daß praktisch alle Nachteile des reinen Zahlen-Container-Konzeptes beseitigt wären. Es ist somit zu erwarten, daß sich dieses Konzept für unsere Anwendungen als sehr brauchbar erweisen wird, da es als eine wirklich praktikable Lösung erscheint, abstrakte Datentypen für Zahlen oder algebraische Strukturen zu definieren.

Wie in [5] dargelegt wird, ist die Benutzung virtueller Funktionen in der Performanz etwa um den Faktor 2 schlechter als die Benutzung nicht virtueller Funktionen. Interessanterweise ergaben diese Messungen, daß auch die `inline`-Deklaration der virtuellen Funktionen keinerlei Verbesserung brachte. Das Laufzeitverhalten bei der Benutzung von Templates ist zwar noch nicht erforscht worden, jedoch ist zu erwarten, daß sich die einfachere Grundkonzeption dieses Features - im Gegensatz zu virtuellen Funktionen - günstig auswirkt.

Auch wird die Benutzung virtueller Funktionen durch Hinzunahme dieses Konzeptes wesentlich vereinfacht, da durch die Parametrisierbarkeit der Klassen die Typübereinstimmungen der Parameter und Rückgabewerte dieser Funktionen „erzwungen" werden können. Die angesprochene hierarchische Anordnung von Objekten wie beispielweise `Ring` und `Koerper` sind mit Hilfe diese Features einfacher zu realisieren.

Kapitel 7

Dokumentationsstandards

Unter dem Oberbegriff *Dokumentation* eines Programmes wird im folgenden das *Layout* des Programmlistings, die im Listing vorkommenden *Kommentare* sowie die *Organisation* des Systems in Dateien zusammengefaßt.

Das Ziel der Definition dieser Standards ist einerseits, die Lesbarkeit der Programme zu verbessern, und andererseits, es zukünftigen Benutzern leichter zu machen, Implementierungsdetails von Algorithmen, Funktionen und Klassen nachzuvollziehen. Die Dateienorganisation ist für die Größe des Programmcodes sowie die textuelle Erstellung dieser Dateien von großer Bedeutung. Auch soll die Erstellung einer rein textuellen Dokumentation der Programme weitgehend automatisiert werden, was durch die Schaffung einer Schnittstelle zum Schriftsatzprogramm LaTeX geschieht. Ein wichtiger Punkt bei der Gestaltung von Dokumentationsstandards ist die Namensgebung für Funktionen, Variablen und Makros. Die Namen sollen dabei soweit wie möglich selbsterklärend sein. Auf diese Problematik wird im folgenden Abschnitt eingegangen.

7.1 Namen und Schreibweisen

Die Namensgebung ist ein wichtiger Bestandteil der Lesbarkeit von Programm-Code und der Aussagekräftigkeit von Dateinamen. Bei Funktionen und Variablennamen sollten daher sehr aussagekräftige Namen gewählt werden. Beispielweise wird die Funktion, die den größten gemeinsamen Teiler zweier Zahlen berechnet, nicht etwa als

```
f(zahl1, zahl2);
```

im Programm auftreten, sondern, sofern der Name der Funktion nicht zu lang wird, exakt Auskunft über den Zweck und Inhalt der Funktion geben:

```
ggt(zahl1, zahl2);
```

7.1.1 Namensgebung bei Variablen und Funktionen

Bei der Namensgebung der C++-Konstrukte ist auf folgende Punkte zu achten:

- Die Namen sollten nicht länger als 16 Zeichen sein.
- Namen sollten nicht mit einem Underline (z.B. `_test`) beginnen, da dies sowohl in C als auch in C++ zu Namenskonflikten mit Bibliotheksnamen führen kann.
- Typdefinitionen und Klassennamen beginnen immer mit einem Großbuchstaben. z.B. `class Bignum` oder `typedef struct Complex ....`
- `#define`-Konstanten werden immer mit Großbuchstaben versehen (z.B. `#define MAX 100`) .
- Variablennamen werden klein geschrieben und werden bei zusammengesetzten Namen mit Underlines verbunden:

  ```
  int first_item;
  ```

- Für `const`-Variablen gilt dasselbe wie für die Standardvariablen, jedoch gelten bzgl. der Groß- und Kleinschreibung keine Beschränkungen.

Ein Weiterentwickler eines bereits bestehenden Programmes kann somit schon anhand des Namens erkennen, ob eine Klasse, Variable oder Funktion gemeint ist.

7.1.2 Dateinamen

Die vorher schon als Standard festgelegte Philosophie der *Trennung zwischen Implementation und Deklaration* bei der Realisierung von Klassen in C++ bringt es mit sich, daß schon die Definition von nur einer Klasse in der Hierarchie eine sehr große Anzahl von Dateien verursacht. Um Änderungen an einer Klasse schnell vornehmen zu können, ist es sehr wichtig, die Funktionen in den einzelnen Dateien schon am Dateinamen zu erkennen. Daher haben die enstehenden Dateien ein bestimmtes Namensformat:

- Die Datei, die die Deklaration der Klasse enthält, heißt genauso wie die Klasse und hat als Endung ein h (z.B. `Bignum.h`).
- Die einzelnen .c-Dateien, die die Implementierung der Klasse beinhalten, müssen die realisierten Algorithmen, Funktionen etc. durch ihren Namen charakterisieren (z.B. `big_op_all.c` für die Operator-Funktionen der Klasse `Bignum`).

Generell sollten die Dateinamen etwa folgendes Format haben:

`<Klassenname>_<Funktionstyp>_<Funktion>.<Dateiart>`

Dabei sind

- `<Klassenname>` eine Abkürzung des Klassennamens,
- `<Funktionstyp>` Typenbezeichnungen wie z.B. *Konstruktor, Destruktor, Identifikationsfunktion* oder *Arithmetische Funktion*,
- `<Funktion>` die realisierte Funktion oder Algorithmus, wie z.B. `ggt()` oder `mul`, als Funktionen, die den ggT oder das Produkt zweier Zahlen berechnen.
- `<Dateiart>` die Art der Implementierung, d.h. Vorwärtsdeklaration (.h-Dateien), oder die tatsächliche Realisierung der Funktion (.c-Dateien)

Wie die Abkürzungen im einzelnen heißen, ist dem Anhang zu entnehmen. *Allgemein sollte aus dem Dateinamen gut zu ersehen sein, welcher Art der Code ist, den die Datei beinhaltet.*

7.2 Formate von Strukturen und Anweisungen

Anweisungen (wie z.B. `if, while, for, switch` etc.) sind oft tief verschachtelte Ausdrücke, deren einzelne „Ebenen" nur durch strukturierte Darstellung gut erkennbar sind. Die dabei auftretenden Klammern repräsentieren aufgrund ihrer Einrückungstiefe im Text der Quelldatei die jeweilige Ebene:

```
while(i < 100)
{
  for(j = 0;j < 20;j++)
  {
    if(!valid)
    {
      a = b;
    }
  }
}
```

Hierbei sollten die geschweiften Klammern immer in einer eigenen Zeile stehen. *Ein Anweisungsblock muß, auch wenn er nur aus einer Anweisung besteht, von öffnenden und schließenden Klammern umschlossen sein*, um evtl. den Ablauf erweitern zu können, ohne sich um die Klammerung kümmern zu müssen. Dabei wird der Anweisungsblock immer um ein „Tabstop" - *empfohlen werden zwei Blanks* - unter der Klammer eingerückt. Besteht der Anweisungsblock nur aus einer Anweisung und ist nicht zu erwarten, daß er aufgrund einer Änderung zu mehreren Anweisungen vergrößert wird, so kann die Umschließung dieses Blocks entfallen. Allerdings muß dann der nur aus einer Anweisung bestehende Block in einer Zeile hinter dem entsprechenden Statement stehen:

```
for(;;) i++; // Korrekt
for(;;)
  i++;       // Verboten
```

Bezüglich der Plazierung der öffnenden Klammern ist allerdings auch der Standard aus [7] ist ebenfalls möglich:

```
while(i < 100) {
  for(j = 0;j < 20;j++) {
    if(!valid) {
      a = b;
    }
  }
}
```

Strukturen wie Klassen, `structs` und `unions` haben folgendes Aussehen:

```
class Bignum
{
  LONGTYPE b;
      ...
};
struct Complex
{
  LONGTYPE re;
  LONGTYPE im;
};
```

Hierbei ist zu beachten, daß die Strukturkomponenten ähnlich wie bei den Anweisungen eine Tabulatorweite gegenüber den geschweiften Klammern eingerückt werden.

7.2.1 Verwendung von Leerzeilen

Wenn es der Übersichtlichkeit des Programmcodes dienlich ist, sollten zur Strukturierung Leerzeilen verwendet werden. Die Benutzung von Leerzeilen ist allerdings dem Programmierer freigestellt.

7.2.2 Verwendung von Leerzeichen

Normalerweise werden Leerzeichen von den meisten C, C++-Compilern als Quellzeichen ignoriert. Dennoch können bei falscher Plazierung der Leerzeichen im Quellcode des Programms Mehrdeutigkeiten auftreten. Die Blankverwendung in C, C++-Code sollte die nachfolgenden Regeln berücksichtigen.
Blanks sollten nicht auftreten

- um die Operatoren „`->`" und „`.`":

  ```
  a -> b = c; // Unzulaessig (Compilerfehler)
  a . b = c;  // Unzulaessig
  a->b = c;   // Korrekt
  a.b = c;    // Korrekt
  ```

- um unäre Operatoren wie z.B. „`--`" oder „`++`":

  ```
  a ++;// Unzulaessig
  a++; // Korrekt
  ```

- zwischen `#` und `define` (Compilerfehler!)

Hingegen sind Blanks erwünscht

- Vor und hinter binären Operatoren:

  ```
  a = a + b;
  //statt
  a=a+b;
  ```

- nach Kommata, Semikolons und Schlüsselwörtern:

  ```
  a = a + b, b = b + c;//Blank hinter dem Komma
  friend operator ...
  //statt
  friendoperator...
  ```

7.3 Formate von Funktionen

7.3.1 Der Funktionskopf

Der Funktionskopf setzt sich aus einem Kommentarteil (siehe nächsten Abschnitt) und der Parameterliste zusammen. Bei allen C++-Compilern ist es möglich - im Gegensatz zu älteren C-Compilern - schon in der Parameterliste die Typisierung der Parameter vorzunehmen. Diese Möglichkeit hat sich nunmehr im Rahmen des ANSI-Standards durchgesetzt. Wird jedoch die Liste der Parameter zu lang, so sollte der Übersichtlichkeit wegen auf die „alte" Technik zurückgegriffen werden: Statt

```
<Typ> f(<Typ> var1, <Typ> var2 , <Typ> var3,
        <Typ> var4, <Typ> var5 ...)
```

sollte man besser schreiben:

```
<Typ> f(var1, var2, var3, var4, var5, ...)
<Typ> var1;
<Typ> var2;
   ...
```

Diese Technik ist sogar auf den meisten Compilern unter UNIX noch verbindlich, da diese meistens den Standard aus [7] benutzen.

7.3.2 Der Deklarationsteil

Der Deklarationsteil deklariert alle *lokal* in der Funktion benötigten Variablen.

```
f(int x)
{
 extern int global;
     ...
}
```

In C und C++ ist es möglich, Variablen zum Zeitpunkt ihres Aufrufs zu deklarieren:

```
for (int i;i < MAX;i++);
```

Um jedoch die Funktionen änderungsfreundlicher zu machen, sollten alle Variablen im Deklarationsteil der Funktion deklariert werden, da damit immer klar ist, welche Variablen in der Funktion bereits deklariert sind:

```
int i;
for(i = 0;i < MAX;i++);
```

statt

```
for(int i;i < MAX;i++);
```

Es gibt allerdings öfter Situationen, in denen es Sinn macht, die Variablen dann erst zu deklarieren, wenn man sie braucht. Dies ist in Bezug auf den Speicherplatzbedarf sinnvoll, da z.B der für eine Variable in einem Anweisungsblock statisch allokierte Speicher nach dem Austreten aus dem Anweisungsblock wieder freigegeben wird. Jede Deklaration steht dabei in einer eigenen Zeile. Sowohl die Variablen als auch die Funktionen werden eine Tabulatorbreite, mindestens jedoch ein Blank gegenüber der öffnenden Klammer der Funktion eingerückt.

7.4 Kommentare

Die gute Kommentierung der Programme ist die Voraussetzung für ihre Lesbarkeit und Änderungsfreundlichkeit. Kommentare dokumentieren sowohl den Ablauf einer Funktion oder eines Algorithmus, als auch den Gebrauch von Variablen. Kommentarzeichen in C++ sind die von C bekannten Zeichen „Slash" und „Stern":

```
/*  <Kommentar> */
```

als auch die Doppelverwendung des Slashes:

```
// <Kommentar>
```

Wir unterscheiden zwei Arten der Kommentierung:

1. Die *Inline-Kommentare*, das sind die nur an der aktuellen Stelle im Quellcode wichtigen Kommentare.

2. Die *Dokumentationskommentare*, das sind die Kommentare, die Bestandteil einer „externen" Dokumentation des Programmes sein sollen.

Die Dokumentationskommentare

```
/* <Kommentar> */
```

stellen die Schnittstelle zum LaTeX-Dokumentationsprogramm `cnoweb` her, welches den mit diesen Kommentarzeichen beginnenden Text aus dem Quellcode als „in LaTeX zu formatierenden Text" kennzeichnet.

Die mit

```
// <Kommentar>
```

beginnenden Inline-Kommentare sind eine Hilfestellung für Programmierer und Anwender, die an einer bestimmten Stelle im Programm eine Erklärung des aktuellen Ablaufs benötigen. Was die einzelnen Kommentartypen an Erklärungen beinhalten sollen, wird in den nächsten Abschnitten dargestellt. Auf die Verwendung der LaTeX-Kommentare wird im Abschnitt „Die Schnittstelle zu LaTeX" eingegangen. *Falls nicht ausdrücklich darauf hingewiesen wird, sind für eine Kommentierung nur Dokumentationskommentare zu verwenden!*

7.4.1 Kommentare in Funktionen

Der Kommentarteil eines Funktionskopfes sollte folgende Grobstruktur haben:

```
/*-----------------------------------------------------------
Dateiname:     Name der Datei
Autor:         Name und Institut des Programmierers
Datum:         Datum der Ersterstellung der Funktion
Funktionsname: Name der Funktion
Aenderungen:   Datum und Art von Aenderungen
Parameter:     Erlaeuterung der Parameter
Return:        Erlaeuterung der Rueckgabewerte
Extern:        Externe Variablen und Funktionen,
               die benutzt werden
Fehler:        Erlaeuterung der Fehlerbehandlung
Seiteneffekte: z.B. von der Funktion veraenderte
               globale Variablen
Algorithmus:   Kurzbeschreibung des verwendeten Algorithmus
-------------------------------------------------------------
*/
```

Die durch diese Kommentarart bedingten möglichen LaTeX-Befehle sind an dieser Stelle weggelassen worden. Dem Kopf folgt eine genaue *Beschreibung des Algorithmus*, der in der Funktion benutzt wird. Bevor die Implementierung der Funktion beginnt, sollte außerdem eine genaue Beschreibung des Ablaufs der Funktion vorhanden sein. Bei Algorithmen eignet sich dafür z.B. eine *Pseudo-Code-Darstellung.* Welche Angaben und Daten die einzelnen Punkte beinhalten sollen, wird im folgenden behandelt.

Angaben zum Autor der Funktion

Außer dem Namen und des Institutes des Autors sollten u.U. *das Projekt*, innerhalb dessen die Software entwickelt wurde, sowie *der Entwicklungsrechner* angegeben werden.

Änderungshistorie

Hier sollten exakte Angaben über Art der Änderung, den Autor, das Datum, sowie die Konsequenzen der Änderung für das gesamte System vorhanden sein.

Parameter

Die ausführliche Erklärung der einzelnen Funktionsparameter ist von entscheidender Bedeutung für den korrekten Aufruf der Funktion. Bei deren Kommentierung ist besonders auf folgende Punkte zu achten:

- Welche Funktion hat der formale Parameter in der Funktion?
- Wie wird ein formaler Parameter übergeben (call by value, call by reference)?
- Wird neuer Speicher für den Parameter allokiert?
- Erfolgt eine Typumwandlung des Parameters innerhalb der Funktion (Casting)?

Rückgabewert der Funktion

Sofern keine Zeiger auf Strukturen von der Funktion zurückgegeben werden, charakterisieren die Rückgabewerte zumeist die in der Funktion aufgetretenen Fehler. Die einem Rückgabewert zugeordnete Fehlersituation sollte an dieser Stelle genau beschrieben werden, damit ein Benutzer leicht den in der Funktion aufgetretenen Fehler lokalisieren kann. Hat die Funktion einen Rückgabetyp, der nicht unmittelbar klar ist, sollte dieser kurz beschrieben werden und die Datei angegeben werden, in der die Typisierung des Rückgabewertes erfolgt.

Fehlerbehandlung

Erfolgt die Fehlerbehandlung nicht über Rückgabewerte, so steht hier eine Beschreibung eventueller Fehlermeldungen.

Seiteneffekte

Werden innerhalb der Funktion globale Variablen verändert, so muß dies an dieser Stelle gut kommentiert sein. Zu den Seiteneffekten zählt auch das Auftreten gleichwertiger Situationen, in denen sich die Funktion unterschiedlich verhält.

Die Beschreibung des Algorithmus

Realisiert die Funktion einen Algorithmus, so wird dieser ausführlich mit folgenden Stichworten kommentiert:

- Literatur, Autor,
- Ablauf,
- verwendete Variablen,
- Laufzeitverhalten (Mittelwert, ,,worst case"),
- Fehleranfälligkeit.

7.4.2 Kommentierung von Variablen in Funktionen

Alle in einer Funktion verwendeten Variablen müssen per Kommentar an der Stelle ihres Auftretens erklärt werden. Der Kommentar sollte folgende mögliche Eigenschaften der Variablen charakterisieren:

- den Sichtbarkeitsbereich der Variablen (global, lokal, statisch),
- ihren Verwendungszweck (Laufvariable, Hilfsvariable, Membervariable),
- ihren globalen Bezug (automatic, static, extern).

7.4.3 Kommentierung von Funktionen in Funktionen

Alle in einer Funktion verwendeten Funktionen sollten in einer Aufrufliste als Dokumentationskommentar vertreten sein. Die Aufrufliste einer Funktion sollte sowohl die *aufrufenden*, als auch die *aufgerufenen* Funktionen beinhalten:

```
/* Aufrufende Funktionen:
                              Funktion 1
                              Funktion 2
                              Funktion 3
```

```
                     Funktion 4
                         ...
 Aufgerufene Funktionen:
                     Funktion 5
                     Funktion 6
                         ...
*/
```

7.4.4 Kommentare in Klassen

Der Dateikopf der Deklarationsdatei einer Klasse beinhaltet all die Informationen, die auch im Kopf von Funktionen kommentiert werden, sofern sie für eine Klassenbeschreibung sinnvoll sind. Die Grobstruktur des Dateikopfes ist daher analog zu der Grobstruktur bei Funktionen:

```
/*---------------------------------------------------------
Dateiname:      Name der Datei
Autor:          Name und Institut des Programmierers
Datum:          Datum der Ersterstellung der Klasse
Name:           Name der Klasse
Datentyp:       Der von der Klasse repraesentierte Datentyp
Andere Klassen:Evtl. klassenintern benutzte andere
                Klassen
Aenderungen:    Datum und Art von Aenderungen
Fehler:         Erlaeuterung der Fehlerbehandlung
Seiteneffekte: z.B. von der Klasse veraenderte
                globale Variablen
Vererbung:      Platz der Klasse in der Hierarchie
Templates:      Parametrisierungsmoeglichkeiten des
                Datentyps
Operatoren:     Liste der ueberladenen Operatoren
Verknuepfungen:Liste der Verknuepfungsmoeglichkeiten mit
                anderen Klasseninstanzen
                (z.B. Bignum + Bigfloat)
Deklarationen: Anleitung zum Deklarieren einer
                Instanz der Klasse
/*---------------------------------------------------------
```

Der Datentyp

Klassen repräsentieren meistens abstrakte Datentypen (ADT's), die in ihrer Struktur u.U. sehr kompliziert sind. Um einen solchen ADT sinnvoll verwenden zu können, benötigt der Benutzer eine genaue Beschreibung des von der Klasse repräsentierten Objektes oder Datentyps. In der Klasse `Bignum` würde unter dem Stichwort „Datentyp" die Beschreibung der Repräsentation der Ziffern als ein Feld von `long`-Variablen erfolgen. Die Beschreibung des Datentyps impliziert damit auch eine Beschreibung der Membervariablen der Klasse.

Benutzte Fremdklassen

Unter dem Stichwort „Andere Klassen" in obigen Kommentarteil sollten alle Membervariablen aufgelistet werden, die als Instanzen einer Fremdklasse auftreten, sofern sie nicht vererbt wurden. *In diesem Zusammenhang ist es besonders wichtig, die Members genau zu beschreiben, die Instanzen privater Klassen sind.* Alle Freundklassen der Klasse werden unter diesem Punkt ebenfalls aufgelistet.

Die Fehlerbehandlung

In einer Klasse erfolgt die Fehlerbehandlung zumeist über Funktionen oder überladene Operatoren (siehe „Effiziente Programmierstile in C++). Wie bei Funktionen werden hier die möglichen Fehlersituationen und die Reaktion des Programmes darauf beschrieben.

Seiteneffekte

Analog zur Kommentierung der Funktionen sollen an dieser Stelle alle bekannten Seiteneffekte, die bei der Benutzung der Klasse auftreten können, dokumentiert werden. Auch hier sind insbesondere eventuelle Veränderungen an globalen Variablen oder an Members von Freundklassen zu dokumentieren.

Vererbung

Sofern die Klasse in eine Klassenhierachie eingebunden ist, sollte der Platz der Klasse in der Hierachie in Form eines „Vererbungsteilbaumes" dargestellt werden:

```
Object
  Ring
    Koerper
```

Diese Darstellung bedeutet, daß die Basisklasse von `Ring` die Klasse `Object` und die Basisklasse von `Koerper` die Klasse `Ring` ist. Die Situation der Mehrfachvererbung wird wie folgt dargestellt:

```
Basisklasse1 Basisklasse2 Basisklasse3
  Abgeleitete Klasse
```

Für Benutzer von abgeleiteten Klassen ist zusätzlich die Infomation wichtig, ob deren Basisklasse „public" oder „private" geerbt hat.

Templates

Wie bereits erwähnt, ist es in C++ ist es möglich, Datentypen zu *parametrisieren* (siehe „OOP-Konzepte in C++). Für jede Parametrisierungsmöglichkeit sollte an dieser Stelle eine Auflistung der möglichen Parameter erfolgen, z.B. :

```
Templates:
 vector<int>
 vector<float>
 vector<Bignum>
    ...
```

Liste der überladenen Operatoren

Da zumeist alle sinnvoll überladbaren Operatoren überladen werden, genügt hier eine Auflistung der *nicht* gemäß der Tabelle im Anhang standardmäßig überladenen Operatoren.

Verknüpfungsmöglichkeiten mit anderen Klassen

Hier soll dargestellt werden, mit welchen anderen Datentypen eine Instanz der Klasse über binäre Operatoren verknüpft werden kann. Da unser Standard vorsieht, daß immer alle Verknüpfungsrichtungen überladen werden, genügt die Darstellung *einer* Richtung:

```
/* Bignum <--> Bigfloat
   Bignum <--> int
   Bignum <--> long
          ...
*/
```

Deklarationen

Um die Benutzung der Klasse so einfach wie möglich zu machen, sollte eine genaue Anleitung zum Instanziieren der Klasse vorgesehen werden. Diese Anleitung enthält damit implizit auch eine Beschreibung der Konstruktoren der Klasse. Im Falle von `Bignum` müssen dafür als Kommentar mindestens vorhanden sein:

```
// Deklarationen:
// 1) Bignum a;     Dekl. fuer eine Bignum mit unbekannter
//                  Stellenzahl
// 2) Bignum a(10);Zuweisung der Zahl 10 an a
// 3) Bignum a[10];Dekl. fuer ein Feld von
//                  Bignums der Laenge 10
```

7.5 Die Organisation in Dateien

In einem Algebrasystem gibt es eine Vielzahl von Modulen, deren Organisation in Dateien im wesentlichen unter zwei Gesichtspunkten erfolgen muß:

1. der Größe des beim Übersetzen enstehenden Codes und das dadurch bedingte Laufzeitverhalten,
2. der strukturierten Dateiorganisation.

Es hat sich bei der C-Programmierung an unserem Lehrstuhl gezeigt, daß eine Dateiorganisation, bei der jede Funktion in einer eigenen Datei steht, sehr übersichtlich ist. In C++ stellt sich das Problem, daß in jede .c-Datei die Deklarationsdatei (.h-Datei) eingebunden werden muß, was einen sehr großen Overhead in Bezug auf die Codegröße bedeutet. Der Objektcode der `Bignum`-Arithmetik beispielweise wurde mit dieser Technik etwa acht mal so groß wie bei der Implementierung *aller* Funktionen in *einer* Datei. *Daher hat man sich an unserem Lehrstuhl vorerst entschieden, alle C++-spezifischen Funktionen einer Klasse (z.B. die Operator-Funktionen) in* einer *Datei zu implementieren.* Eine Klasse besteht somit aus zwei Dateien, (z.B `Bignum.h, Bignum.c`). Die die Operator-Funktionen unterstützenden reinen C-Hilfsfunktionen werden *nach Gruppen geordnet* in einzelnen Dateien abgelegt. Sinnvolle Gruppierungen wären z.B. die im zweiten Kapitel vorgestellten Oberbegriffe für die Elementfunktionen einer Klasse. Oft ist es so, daß ein Benutzer einer Klasse (z.B. `Polynom`) u.U. nur mit kleinen ganzzahligen Koeffizienten rechnen will und nicht für die Koeffizienten die aufwendigere Langzahl-Arithmetik involvieren möchte. In einem solchen Falle böte es sich an, zwei Versionen der

Klasse `Polynom` zu implementieren, nämlich eine Version, in der die Darstellung der Koeffizienten des Polynoms mit `long`-Werten erfolgt und eine andere, die die Koeffizienten als „lange Zahlen" darstellt. Damit wird beim Rechnen mit Polynomen, die „kleine" Koeffizienten haben, das Laufzeitverhalten und der Speicherplatzbedarf wesentlich günstiger.

7.6 Deklaration von Hilfsfunktionen

In Kapitel 3 wurde auf die Hilfsfunktionen hingewiesen, die Zwischenrechnungen innerhalb der Elementfunktionen einer Klasse durchführen. Da diese Funktionen oft reine C-Funktionen sind, müssen sie innerhalb einer C++-Umgebung als solche kenntlich gemacht werden. Die Vorwärtsdeklaration einer reinen C-Funktion, die außerhalb des Moduls definiert ist, in dem die Deklaration stehen soll, hat innerhalb eines C++-Programmes das folgende Aussehen:

```
extern "C" <Funktion>
```

7.7 Manual-Pages

Für alle Implementierungen von Klassen und Funktionen in C, C++ sowie UNIX-Systemaufrufen etc. gibt es auf UNIX-Systemen eine Online-Hilfe für deren Benutzer, die sogenannten *Manual-Pages*. Mit dem Aufruf

```
man <Funktionsname>
```

erhält der Benutzer Informationen über die gewünschte Funktion oder Klasse. Die Manual-Pages sind Dateien, in denen strukturiert Informationen abgelegt sind. Die Einrichtung einer Manual-Page ist sehr einfach; sie erfolgt einfach durch die Anlage einer ASCII-Datei mit dem Namen der Funktion oder Klasse und wird vom Systemverwalter in einem dafür vorgesehenen Unterverzeichnis auf einem der Rechner des Netzes bereitgestellt. Es hat sich gezeigt, daß die Manual-Pages eine der am meisten benutzten Informationsquelle für unter UNIX installierte Software sind. *Daher sollte nach Fertigstellung der Implementation einer Klasse oder Funktion von jedem Programmierer eine Manual-Page erstellt werden, die die von ihm erstellte Software dokumentiert.* Die Strukturierung entnimmt man leicht den bereits bestehenden Manual-Pages.

7.8 Die Schnittstelle zu LaTeX (`cnoweb`)

Ohne auf eine *vollständige Dokumentation* zu verzichten, soll Benutzern von Modulen und Klassen unserer Software unabhängig von den Manual-Pages die Möglichkeit geboten werden, schnell auf eine ausreichende Dokumentation dieser Software zuzugreifen. Die Weitergabe von Informationen zu einzelnen Funktionen und Klassen ist nicht immer einfach, da die Dokumentation in aller Regel nur in wenigen Exemplaren existiert oder mit einem Textverarbeitungsprogramm erstellt wurde, das nicht an jedem Institut existiert. Das Schriftsatzprogramm LaTeX hat - zumindest im universitären Bereich - große Verbreitung gefunden. *Ziel des oben erwähnten Dokumentationsstils ist es, mit der Bereitstellung von Software gleichzeitig die Bereitstellung ihrer Dokumentation zu gewährleisten.* Dies soll mit Hilfe der Dokumentationskommentare geschehen, die mit Hilfe eines Tools (`cnoweb`) in eine gut formatierte LaTeX-Datei umgewandelt werden können.

Wir haben bisher zwei Kommentarmöglichkeiten unterschieden:

1. die Inline-Kommentare:

   ```
   // <Kommentar>
   ```

2. und die Dokumentationskommentare:

   ```
   /* <Kommentar> */
   ```

Alle Kommentare, die von den gewohnten C-Kommentarzeichen im Text umschlossen werden, können mit Hilfe von `cnoweb` in einen LaTeX-formatierten Text umgewandelt werden. Dies hat den Vorteil, daß mit einer Änderung des Quellcodes gleichzeitig eine Änderung der Dokumentation verbunden ist, da beide in derselben Datei stehen. Dazu geht man wie folgt vor: Zunächst legt man sich eine LaTeX-Datei an, in der der unformatierte Text mit den LaTeX-Kommandos stehen soll. Diese Datei sollte etwa folgendes Grundformat haben:

```
\documentstyle[german,cnoweb]{report}
\begin{document}
\input cnowebcode{\Bignum.c}
\end{document}
```

Das Format ist also dasselbe wie das einer herkömmlichen LaTeX-Datei. Zu beachten ist die Übergabe der Dokumentstiloption „`cnoweb`". Mit dem `input`-Befehl wird die entsprechende Programmdatei - in diesem Falle `Bignum.c` - in die LaTeX-Datei geladen. Aus ihr werden beim Aufruf von

```
glatex Bignum.tex
```

die Dokumentationskommentare aus der Quelldatei als Dokument betrachtet. Das eigentliche Listing des Programmes wird im sogenannten `verbatim`-Modus von LaTeX in seinem ASCII-Format übernommen. Prinzipiell können in den Dokumentationskommentaren von `Bignum.c` alle üblichen LaTeX-Befehle benutzt werden. Die *Dokumentationsdatei* wird nach ihrer Erstellung vom Systemverwalter auf einem der Allgemeinheit zugänglichen Unterverzeichnis bereitgestellt. Natürlich will ein Benuzter einer Klasse nicht unbedingt das komplette Programm einschließlich seiner Kommentierung ausgedruckt bekommen. Es wird daher eine lehrstuhleigene Modifikation des Programmes `cnoweb` geben, welche es erlauben soll, verschiedene „Stufen" der Dokumenterstellung durch Parameter zu steuern. Ein Benutzer kann dann wählen, wieviel Information über die Quelldatei ausgedruckt werden soll.

7.9 Die Erstellung von UNIX-Makefiles

Ein weiterer Faktor der Portabilität von Software unter UNIX, ist die Erzeugung von sogenannten *Makefiles*. Diese sind Dateien, mit deren Hilfe man u.a. das automatische Erkennen von Abhängigkeiten in Programmsystemen erreichen kann. Jenachdem wie allgemein diese Skripte gehalten werden, kann ein Benutzer die ihm zugegangene Software auf seinem System übersetzen, ohne daß er Pfade o.ä. setzen muß. In diesem Abschnitt werden daher einige Punkte behandelt, die man bei der Generierung dieser Dateien beachten muß.

7.9.1 Die Benutzung von *make*

`make` ist ein Programm unter UNIX, welches eine automatische Generierung eines Programmsystems erlaubt (siehe auch [10]). `make` benötigt zur Generierung eines Programmpaketes im wesentlichen zwei Informationen:

1. Die Beschreibung der Abhängigkeiten der Module,
2. das Datum der letzten Änderung der aus 1) ermittelten Dateien.

Die unter 1) angegebenen Abhängigkeiten muß der Benutzer in einer Datei (dem Makefile) spezifizieren. In dieser Spezifikation werden die folgenden Arten von Anweisungen benutzt:

- Makrodefinitionen
- Definition der Abhängigkeiten

- Shellkommandos
- Steueranweisungen

Im folgenden wird sukzessive ein typisches Makefile entwickelt. *Die Struktur dieses Makefiles soll als Standard für die Programmierung an unserem Lehrstuhl übernommen werden.* Die Namensgebung der Datei, die das Makefile spezifiziert, ist beliebig. Standardmäßig erwartet UNIX den Namen `makefile`, wobei alle Kombinationen bzgl. der Groß- und -Kleinschreibung der einzelnen Buchstaben möglich sind. Der Dateinamen `MaKeFilE` wird ebenfalls erkannt. Möchte ein Programmierer einen eigenen Namen für das Makefile vergeben (z.B. `Bigmake` für die `Bignum`-Arithmetik), so wird `make` wie folgt aufgerufen:

```
make -f Bigmake <Ziel>
```

Die Bedeutung des Zusatzes `Ziel` wird in den nächsten Abschnitten klarer.

Der Kopf des Makefiles

Im Kopf jedes Makefiles stehen, wie bei Funktionen und Klassen, die auch dort üblichen Kommentare, wie Autor, Datum, Änderungen, Besonderheiten etc. . Das Kommentarzeichen in einem UNIX-Makefile ist das Nummernzeichen (`#`). Der Anfang einer solchen Datei hat also etwa folgendes Aussehen:

```
#################################################
# Name:           Name des Makefiles
# Autor:
# Datum:
# Aenderungen:
# Programmsystem: z.B. Arithmetik f. lange Zahlen
# Aufruf:
#################################################
```

7.9.2 Die Grundeinstellungen

Zum Übersetzen einer Vielzahl von Modulen ist zunächst wichtig, welcher Compiler dafür verwendet werden soll und in welchen Verzeichnissen die einzelnen Dateien, die zur Übersetzung wichtig sind, zu finden sind. Für die Pfade zu den einzelnen Dateien verwendet man am besten Makros, da ein Benutzer diese Makros schlimmstenfalls nur durch seine eigenen Makros ersetzen muß:

```
# Es folgt der Name des benuzten Compilers:
CC = cc
CPP = g++

# Zum Uebersetzen benoetigte Verzeichnisse:

LIBDIR=/home/username/lib
INCLUDEDIR=/home/username/include
BINDIR=/home/username/bin
```

Mit dem Makro `CC` ist in dem Makefile zukünftig der UNIX-C-Compiler gemeint, mit `CPP` der C++-Compiler.

Eine im Zusammenhang mit Makefile häufig auftretende Umgebungsvariable ist `CFLAGS`. Diese kann im Makefile mit Optionen für den C, C++-Compiler belegt werden:

```
# Option fuer den Compiler, optimierten
# Objektcode zu generieren:
CFLAGS = -O
```

7.9.3 Die Bestandteile des Zielobjektes

Ein lauffähiges Programmsystem setzt sich u.U. aus einer großen Anzahl von Objekten zusammen, die später durch den Binder (`ld`) des UNIX-Systems zu einem lauffähigen Programm zusammengebunden werden. Das Objekt, zu dem die einzelnen Bestandteile zusammengebunden werden, heißt *Zielobjekt* des Systems. Zur Vereinfachung werden die Komponenten, aus denen das spätere Zielobjekt bestehen sollen, in einer Variablen zusammengefaßt:

```
# Zielobjekt:
BIG_OBJ= Bignum.o Bigfloat.o mul.o
```

Ein Makefile soll für den Benutzer komfortabel zu handhaben sein. Mit Hilfe der vom Benutzer frei definierbaren *Abhängigkeiten* können mit dem Aufruf von `make` gezielte Aktionen durchgeführt werden. Die Beschreibung von Abhängigkeiten ist sowohl in [13] als auch in [10] sehr gut dargestellt. Im folgenden werden die für unsere Zwecke relevanten Facetten davon untersucht. Mit Hilfe eines Makefiles sollen für den Benutzer folgende Aktionen möglich sein:

- Die Übersetzung des gesamten Systems,
- die Generierung von Teilen des Systems,

- die Installation des Systems,
- die automatische Anpassung des System an die neue Umgebung,
- die Archivierung des Systems in einer Bibliothek,
- die teilweise Archivierung von Neuerungen in der Bibliothek.

Um beispielsweise das komplette System zu übersetzen, müssen alle zur Systemgenerierung wichtigen Dateien bekannt gemacht werden. Dies erfolgt im UNIX-Makefile durch „Labels":

```
Bignum.a: (BIG_OBJ)
          ar ru Bignum.a $(BIG_OBJ)
          ranlib Bignum.a
```

Der Aufruf

```
make Bignum.a
```

hat folgende Konsequenzen:

1. Es werden alle Abhängigkeiten untersucht, die sich aus dem Makro

   ```
   $(BIG_OBJ)
   ```

   ergeben und eventuell die entsprechenden Dateien neu übersetzt.
2. Die Bibliothek wird aktualisiert.

7.9.4 Die Übersetzungsanweisungen

Eine wichtige Option zu Beginn des Makefiles ist die folgende:

```
.c.o $(CC) $(CFLAGS) -c $<
```

Mit dieser Anweisung erreicht man folgendes:

Soll ein Objekt erzeugt werden, so hängt dieses nur von der entsprechenden .c-Datei ab, es sei denn, es wird explizit eine andere Datei angegeben. Diese .c-Datei wird, falls sie jünger als das Objekt ist, mit den Compileroptionen

```
$(CFLAGS)
```

übersetzt. Das Makro `$<` ist ein Platzhalter für die Datei, die den Aufruf verursachte (siehe auch [10] S. 437).

7.9.5 Die Beschreibung von Abhängigkeiten

Im letzten Abschnitt wurden die Abhängigkeiten der einzelnen Objekte von ihren C, C++-Modulen diskutiert. Eine Änderung in einer Headerdatei kommt nur dann zum Tragen, wenn mit der .h-Datei auch die C, C++-Dateien übersetzt werden, die diese gebrauchen. Im Sinne der Makefiles sind also die C, C++-Module *abhängig* von den von ihnen gebrauchten .h-Dateien. Diese Abhängigkeit wird im Makefile wie folgt dargestellt:

```
<Dateiname>.o : <Headerfile1> <Headerfile2>
```

Soll ein Objekt (z.B. `$(OBJ)`) erzeugt werden, so werden durch die Beschreibung der Abhängigkeiten alle zur Erzeugung dieses Objektes notwendigen Dateien aufgelistet.

7.9.6 Hilfsoptionen in Makefiles

Um dem Benutzer eines Makefiles das Handling der Dateien zu erleichtern, sollten in jedem Makefile die folgenden Labels existieren:

```
# Es folgt die Option zum Saeubern
# aktuellen Verzeichnisses:

clean:
rm -f *.o *.a
```

Damit werden alle beim Compilieren entstandenen Objektfiles und Bibliotheksdateien vom aktuellen Verzeichnis gelöscht. Das zweite Label soll beim Installieren von Dateien in die richtigen Verzeichnisse helfen:

```
installlib:
cp Bignum.a $(LIBDIR)
ranlib ($LIBDIR)/Bignum.a
```

Der Aufruf

```
make installlib
```

kopiert die aktuelle Bibliothek in das dafür vorgesehene Verzeichnis. Natürlich sind an dieser Stelle beliebig viele Labels möglich. *Der Benutzer der Software sollte jedoch keinesfalls gezwungen sein, die Installation des Systems „von Hand" durchführen zu müssen.*

7.9.7 Überblick über die Bestandteile eines Makefiles

Ein Makefile sollte wie folgt gegliedert werden:

1. Kopf des Makefiles mit allen das Makefile betreffenden Informationen,
2. Makrodefinitionen für Pfade zu Verzeichnissen und der Definition der verwendeten Compiler.
3. Angabe der Abhängigkeiten zwischen Objekten und Quelldateien sowie zwischen Quelldateien und Headerdateien,
4. Angabe der Abhängigkeiten zur (evtl. teilweisen) Erzeugung eines Archivs (Bibliothek),
5. Labels zum Säubern von Verzeichnissen,
6. Labels zur Installation des Systems.

Jeder dieser Bereiche im Makefile sollte gut kommentiert werden, um Änderungen leichter zu machen. Besonderer Wert beim Erstellen eines Makefiles sollte auf die *Allgemeinheit* der Kommandos gelegt werden. Dies gilt insbesondere für Verzeichnisangaben, Namen und Umgebungsvariablen, die grundsätzlich mit Hilfe von Makros definiert werden sollten. In einem Makefile können auch vom Benutzer definierte „Environmentvariablen" benutzt werden. Im Anhang B ist ein kommentiertes Beispiel eines benutzbaren Makefiles zu finden.

Anhang A

Datentypen in einem Algebrasystem

- lange ganze Zahlen
- lange Gleitkommazahlen
- lange Rationale Zahlen
- p-adische Zahlen
- Komplexe Zahlen
- Permutationen
- Endliche Körper
- Restklassen modulo m
- Polynome in einer Veränderlichen
- Polynome in mehreren Veränderlichen
- Matrizen
- Vektoren
- Potenzreihen
- Mengen
- Graphen
- Strings
- Endliche Gruppen

A.1 Notationsbeispiele

Typ	**Typname**
Kurze ganze Zahlen	int
Lange ganze Zahlen	integer
Lange Gleitkommazahlen	real
Rationale Zahlen	rational
p-adische Zahlen	padic
Komplexe Zahlen	complex
Permutationen	permutation

A.2 Operationen

Folgende Tabelle gibt eine Übersicht über verbindliche Grundoperationen (soweit diese für die einzelnen Typen sinnvoll sind), die mit Zahlen möglich sein sollen und die entsprechenden C++-Operatoren, die dabei überladen werden sollen. In dieser Tabelle werden außerdem die Hilfsfunktionen aufgelistet, die standardmäßig in jeder Klasse enthalten sein sollten.

Operation	**Argumente**	**Ergebnis**	**Name**	**Operator**
Ausgabe auf stdout	stream,Zahl	keines	print	<<
Eingabe von stdin	stream,Zahl	keines	read	>>
Ausgabe auf Datei	Datei,Zahl	int	fwrite	keiner
Ausgabe auf String	char *,Zahl	int	swrite	keiner
Eingabe von Datei	Datei,Zahl	int	fread	keiner
Eingabe von String	char *,Zahl	int	sread	keiner
Zuweisung	Zahl,Zahl	void	as	=
Addition	Summe,Zahl,Zahl	void	add	+
Subtraktion	Differenz,Zahl,Zahl	void	sub	-
Negierung	Zahl,Zahl	void	neg	!
Multiplikation	Prod. Fakt1,Fakt2	void	mult	*
Division m. Rest	Quotient,Rest, Dividend,Divisor	void	div	/
Division o. Rest	Quot,Divt,Dior	void	quot	%
Rest b. Div.	Rest,Divt,Dior	void	rem	keiner
Potenz	Potenz,Basis,Exp.	void	pow	keiner
ggT	ggt,Zahl,Zahl	void	gcd	keiner
ggT mit Linkomb.	ggt,x,y,Zahl,Zahl	void	egcd	keiner
Vergleich	Zahl,Zahl	int	comp	==
Vergleich	Zahl,Zahl	int	neq	!=
Vergleich	Zahl,Zahl	int	leq	<=
Vergleich	Zahl,Zahl	int	geq	>=
Vergleich	Zahl,Zahl	int	lt	<
Vergleich	Zahl,Zahl	int	gt	>=
Inkrement	Zahl	void	inc	++
Dekrement	Zahl	void	dec	--
Zuweisung & Dekr.	Zahl,Zahl	void	mas	+=
Zuweisung & Inkr.	Zahl,Zahl	void	pas	-=
Quadratwurzel	Wurzel,Zahl	void	sqrt	keiner
Shift	Ergebnis,Zahl, Binärdarstellung	void	shl,shr	<<,>>
Logarithmus	Logarithmus,Zahl	void	log2	keiner

Die unter „Name" angegebenen Funktionsnamen sind die Realisierungen der C-Funktionen, die von den Operator-Funktionen aus aufgerufen werden sollen (Modularität). *Für die rechts stehenden binären Operatoren, die überladen werden sollen, sind beide Verknüpfungsrichtungen zu realisieren!*

Anhang B

Beispiel für ein Makefile

Das folgende Listing ist ein Beispiel für die Struktur eines Makefiles (siehe Kapitel 7). Es ist eine modifizierte Version eines Makefiles, das zum Generieren einer Arithmetik für ganze lange Zahlen an der Universität Heidelberg verwendet wurde und kann als Muster zur Erstellung von Makefiles an unserem Lehrstuhl übernommen werden:

```
####################################################
# Name: Makefile
# Autor:
# Datum:
# Aenderungen
# Programmsystem: Arithmetikpaket fuer lange
#                 ganze Zahlen
# Aufruf: make
#
####################################################

# Alle .c-Dateien werden zu Objekten
# gleichen Namens uebersetzt:
.c.o:
$(CC) -c $(CFLAGS) $<

# Compilernamen:
CC=cc
CPP=CC

# Compileroptionen:
CFLAGS=-O
```

```
# Zielobjekt:
SPARCOBJ=isparc.o isparc_vecmuladd.o isparc_cvecsubto.o \
isparc_PLACEdiv.o

CCPLUSOBJ=Int.o

IOBJ=iplace.o ivec.o imem.o iutil.o iadd.o imul.o idiv.o  \
     iio.o igcd.o $(SPARCOBJ) $(CCPLUSOBJ)


ROBJ=irandom.o timing.o

# Pfade:
LIBDIR=/home/merscher/libI
INCDIR=/home/merscher/libI

# Erzeugt wird folgende Bibliothek:
all: libI.a

# Abhaengigkeiten:

# Assembler:
iplace.o:  iplace.s
       as -o iplace.o iplace.s


iplace.s: iloc.h iint.h iplace.c isparc_inline.il
       $(CC) $(CFLAGS) -S iplace.c isparc_inline.il

# Abhaengigkeiten der Objekte von Headerfiles:
ivec.o: iloc.h iint.h

imem.o: iloc.h iint.h

iutil.o:  iint.h iloc.h

iadd.o: iint.h

imul.o:  iint.h

idiv.o:  iint.h
```

```
iio.o:  iint.h

igcd.o:  iint.h

irandom.o:  iint.h random.h timing.h

timing.o: timing.h

# C++-Objekte:
Int.o: Int.C iint.h Int.h
        $(CPP) $(CFLAGS) -c Int.C

# Archivierung, Option ,,ru'' ersetzt nur geaenderte
# Module:
libI.a: $(IOBJ) $(ROBJ)
        ar ru libI.a $(IOBJ) $(ROBJ) ; ranlib libI.a

# Installation:
install: libI.a
        cp -p libI.a $(LIBDIR)
        chmod 644 $(LIBDIR)/libI.a

installh: iint.h Int.h timing.h
        cp iint.h $(INCDIR)
        chmod 644 $(INCDIR)/iint.h
        cp timing.h $(INCDIR)
        chmod 644 $(INCDIR)/timing.h
        cp Int.h $(INCDIR)
        chmod 644 $(INCDIR)/Int.h

# Saeuberung:
clean:
        rm *.o iplace.s libI.a
```

Literaturverzeichnis

[1] S. Lippmann, *C++ Primer*, Addison/Wesley, first edition, 1987.

[2] S. Lippmann, *C++ Primer*, Addison/Wesley, second edition, 1989.

[3] M. Ellis, B. Stroustrup, *The annotated C++ Reference Manual*, Addison/Wesley, 1990.

[4] B. Eckel, *Using C++*, McGraw-Hill, 1989.

[5] K. Gorlen, S. Orlow, P.S. Plexico, *Data Abstraction and object-oriented Programming in C++*, John Wiley & Sons, 1989.

[6] G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991

[7] B.W. Kerninghan, D.M. Ritchie, *The C-Programming Language*, first edition, Prentice-Hall, 1978.

[8] Kerninghan, Pike, *The UNIX Programming Environment*, first edition, Prentice-Hall, 1984.

[9] B. Stroustrup, *Parameterized Types in C++*, Journal of object-oriented Programming, Ausgabe 4, 1990

[10] J. Gulbins, *UNIX Version 7*, Springer Compass, 1990

[11] B.W. Kerninghan, D.M. Ritchie, *Programmieren in C*, Hanser-Verlag, 1978.

[12] *Programmierrichtlinien in C, Hausstandard Targon*, Nixdorf Computer AG, 1988.

[13] *Programmer's Guide, SunOS 4.0, Nr. 9*, SUN Microsystems Germany, 1988.

[14] I. Pohl, *C++ for C-Programmers*, Benjamin/Cummings, 1988.

[15] Weiner and Pinson , *An Introduction to Object oriented Programming and C++*, Addison/Wesley, 1987. Benjamin/Cummings, 1988.

[16] T. Budd, *An Introduction to Object oriented Programming*, , Addison/Wesley, 1991.