

Konzepte von Programmiersprachen

Ein paar Stichpunkte zur Vorlesung

von Ulrich Grude

Skript für Vorlesungen an der
Technischen Fachhochschule Berlin
Fachbereich Informatik und Medien (FB VI)
Version 0.3, SS05

Inhaltsverzeichnis

Konzepte von Programmiersprachen.....	3
1. Ziele dieser Lehrveranstaltung.....	3
2. Literatur.....	3
3. Grundbegriffe der Programmierung.....	4
4. Bojendarstellung von Variablen und Konstanten.....	4
5. Typen in Java und in C++.....	5
6. Das Konzept eines Stapels.....	8
7. Das Konzept einer rekursiven Funktion.....	8
8. Funktionales Programmieren.....	9
9. Das Konzept eines Moduls.....	10
10. Das Konzept einer Klasse.....	11
11. Das Konzept einer generischen Einheit.....	11
12. Das Konzept einer nicht-schwachen Funktion.....	12
13. Das Konzept eines Zustands.....	13
14. Parameter-Übergabemechanismen.....	15
15. Anhang A: Beispiele für Bojen.....	17
15.1. Bojen für zwei Variablen definierter Typen.....	17
15.2. Bojen für zwei Zeiger-Variablen.....	17
15.3. Bojen für zwei Variablen mit gleichen Referenzen und identischem Wert.....	17
15.4. Bojen für zwei Variablen mit identischem Wert bzw. Zielwert.....	18
15.5. Bojen für Konstanten, Zeiger auf Konstanten und konstante Zeiger.....	18
15.6. Boje für eine Reihung.....	18
15.7. Bojen für ein Objekt.....	19
15.8. Eine Zeigervariable, die auf NULL zeigt.....	20
16. Anhang B: Ein paar Beispielprogramme.....	21
16.1. Ein Stapel-Modul.....	21
16.2. Eine Stapel-Klasse.....	23
16.4. Eine Stapel-Schablone.....	25
Die Skriptsprache Perl.....	27
1. Grundlegende Eigenschaften.....	27
2. Der Zwei-Sprachen-Programmierstil.....	27
3. Perl-Konstrukte, die dem Programmierer das Leben (manchmal) erleichtern.....	27
4. Einfache Beispiel-Programme.....	28
5. Kompliziertes hinter Einfachem verstecken (ties).....	30
6. Ein Perl-Programm zum Rechnen mit grossen Ganzzahlen.....	30
7. Eine grafische Benutzeroberfläche.....	32
8. Ein Tischrechner.....	33
9. Ein paar Internetadressen zu Perl.....	35
10. Bewertung von Perl.....	36
11. Zum Abschluss.....	36

Konzepte von Programmiersprachen

1. Ziele dieser Lehrveranstaltung

Anhand von Beispielen soll der Unterschied zwischen einem allgemeinen **Konzept** und seinen speziellen **Ausprägungen** in verschiedenen Programmiersprachen deutlich gemacht werden (z.B. das Konzept eines Typs und Typen in Java, Typen in C++ etc.)

Ein paar schon aus früheren Lehrveranstaltungen **bekannt**e Konstrukte und Konzepte (der Programmiersprachen Java und C++) sollen in einem **neuen Zusammenhang** dargestellt, verglichen, bewertet, diskutiert etc. werden.

Das Verständnis für die schon behandelten Programmiersprachen (Java und C++) soll **vertieft** werden.

Ein paar bisher **noch nicht behandelte** Konzepte und Konstrukte sollen beschrieben werden.

Das Erlernen **weiterer Programmiersprachen** soll **erleichtert** werden.

Programmiersprachen wurden **entwickelt** und werden **weiterentwickelt**. Einige Motive und Kräfte, die bei dieser Entwicklung wichtig waren und sind, sollen **diskutiert**, über zukünftige Entwicklungen soll ein bisschen **spekuliert** werden.

2. Literatur

David A. Watt

„**Programming Language Concepts and Paradigms**“

Prentice Hall, 1990, ca. 320 Seiten, bei Amazon 24,- bis 46,- US\$

Ein Klassiker zum Thema dieser Lehrveranstaltung. Sehr klar geschrieben, viele Konzepte werden behandelt. Sehr empfehlenswert.

Robert W. Sebesta

„**Concepts of Programming Languages**“, Sixth Edition

The Benjamin/Cummings Publishing Company, 1993, ca. 560 Seiten,

bei Amazon 15,- bis 112,- US\$

Ein bewährtes Buch (6. Auflage). Etwas konkreter und mit mehr Beispielen als das Buch von Watt. Sehr empfehlenswert.

Herbert L. Dershem, Michael J. Jipping

„**Programming Languages: Structures and Models**“

Wadsworth Publishing Company, 1990, ca. 400 Seiten, bei Amazon 1,60 bis 60,- US\$

Noch etwas konkreter als das Buch von Sebesta, mit vielen guten Fragen und Aufgaben (leider ohne Antworten und Musterlösungen), preiswert und empfehlenswert.

Bertrand Meyer

„**Introduction to the Theory of Programming Languages**“

Prentice Hall, 1990, ca. 450 Seiten, bei Amazon ca. 37,- US\$

Ein weiterführendes Buch über die Syntax und Semantik von Programmiersprachen und wie man sie "mathematisch in den Griff bekommen kann". Sehr klar geschrieben, aber auch sehr theoretisch und anspruchsvoll. Empfehlenswert für Leser, die sich intensiv mit dem Thema befassen und ihre Erkenntnisse vertiefen wollen.

3. Grundbegriffe der Programmierung

Programmieren als ein **Rollenspiel**. Die 5 Rollen und ihre charakteristischen Tätigkeiten:

1. **Programmierer**: Schreibt Programme, übergibt sie dem Ausführer
2. **Ausführer**: Prüft Programme, lehnt sie ab oder akzeptiert sie, führt Programme aus (auf Befehl des Benutzers)
3. **Benutzer**: Befiehlt dem Ausführer, Programme auszuführen. Ist für Ein- und Ausgabedaten zuständig.
4. **Kollege 1**: Verbessert, verändert, erweitert (kurz: wartet) die Programme des Programmierers.
5. **Kollege 2**: Ist daran interessiert, Programme des Programmierers wiederzuverwenden.

Die fünf wichtigsten **Grundkonzepte** von Programmiersprachen:

1. **Variablen** (Wertebehälter, deren Wert man beliebig oft verändern kann)
2. **Typen** (Baupläne für Variablen)
3. **Unterprogramme** (andere Bezeichnungen: Funktionen, Prozeduren, Methoden, ...)
4. **Module** (Behälter für Variablen, Unterprogramme, Typen, Module etc., der aus mindestens zwei Teilen besteht, einem öffentlichen [ungeschützten, sichtbaren] und einem privaten [geschützten, nicht sichtbaren] Teil).
5. **Klassen** (Eine Klasse ist ein Modul und ein Bauplan für Module).

Drei **Arten von Befehlen** (des Programmierers an den Ausführer):

1. **Vereinbarung** (declaration, "Erzeuge ...", z.B. eine Variable, ein Unterprogramm, einen Typ, ...)
2. **Ausdruck** (expression, "Berechne den Wert des Ausdrucks ...")
3. **Anweisung** (statement, "Tue die Werte ... in die Wertebehälter ...")

4. Bojendarstellung von Variablen und Konstanten

Eine **Variable** besteht konzeptuell aus **zwei** Teilen: Einer **Referenz** und einem **Wert**. Als **dritten** Teil **kann** eine Variable einen **Namen** haben (es gibt aber auch Variablen **ohne** Namen).

Den **Namen** legt der **Programmierer** fest, die **Referenz** in aller Regel der **Ausführer**.

Die sogenannte **Bojendarstellung** unterscheidet vor allem "**normale Werte**" in **rechteckigen** Kästchen und **Referenzen** (oder Zeiger) in **sechseckigen** Kästchen. Eine Zeigervariable hat eine **Referenz** (**sechseckiges** Kästchen) als **Wert**.

In manchen Programmiersprachen wird zwischen **unveränderbaren Variablen** und "**richtigen**" **Konstanten** unterschieden. Eine **richtige Konstante** hat **keine Referenz**, nur einen Namen und einen Wert. Eine **unveränderliche Variable** sieht (als Boje dargestellt) ganz ähnlich aus wie eine Variable, nur dass ihr Wert nicht verändert werden kann.

Mit Hilfe von Bojen kann man sich anschaulich klar machen, was bei der Ausführung eines Programms passiert, vor allem, wenn **komplizierte Zeigeroperationen** ausgeführt oder Parameter **per Zeiger** an ein Unterprogramm übergeben werden.

Ein paar Beispiel-Bojen findet man im Anhang A.

5. Typen in Java und in C++

In **Java** (Version 5.0) unterscheidet man folgende Arten von Typen:

Alle Java-Typen

Primitive Typen

(byte, char, short, int, long, float, double, boolean)

Referenztypen

Klassentypen

(z.B. String, ArrayList<String>, ArrayList, ElementType, ...)

Schnittstellentypen

(z.B. Serializable, Collection<String>, Collection, ...)

Reihungstypen

(z. B. int[], String[], int[][] , Serializable[][][], ...)

Bei den Klassentypen unterscheidet man folgende Unterarten:

Klassentypen

Einfache Klassentypen

(z. B. String, StringBuilder, Long, ...)

Parametrisierte Klassentypen

(z. B. ArrayList<String>, ArrayList<Long>, ...
HashMap<String, Long>, HashMap<String, String>, ...)

Rohe Klassentypen

(z. B. ArrayList, HashMap, ...)

Aufzählungstypen

(z. B. ElementType, RoundingMode, ThreadState, ...)

Weitgehend entsprechend man bei den Schnittstellentypen die folgenden Unterarten:

Schnittstellentypen

Einfache Schnittstellentypen

(z. B. Serializable, Runnable, ...)

Parametrisierte Schnittstellentypen

(z. B. Collection<String>, Collection<Long>, ...
Map<String, Long>, Map<String, String>, ...)

Rohe Schnittstellentypen

(z. B. Collection, Map, ...)

Im **C++-Standard** werden folgende Arten von Typen unterschieden:

Alle C++-Typen

Fundamentale Typen (fundamental types)

Arithmetische Typen (arithmetic types)

Ganzzahltypen (integral types or integer types)

Vorzeichenbehaftete Ganzzahltypen (signed integral types)

signed char, short int, int, long int

Vorzeichenlose Ganzzahltypen (unsigned integral types)

unsigned char, unsigned short int, unsigned int, unsigned long int

Sonstige Ganzzahltypen

bool, char, w_char_t

Gleitpunkttypen (floating point types)

float, double, long double

Der leere Typ (void)

Zusammengesetzte Typen (compound types)

Reihungstypen (array types)

Unterprogrammtypen (function types)

Zeigertypen (pointer types)

Referenztypen (reference types)

Klassentypen (class types)

Vereinigungstypen (union types)

Aufzählungstypen (enumeration types)

Elementzeigertypen (types of pointers to non-static class members)

Häufig ist es aber vor allem wichtig, zwischen den folgenden Arten von C++-Typen zu unterscheiden:

Alle C++-Typen

Definierte Typen

Vordefinierte Typen (built in types, predefined types)

Vom Programmierer definierte Typen (user defined types)

Konstruierte Typen

Reihungstypen

Unterprogrammtypen

Zeigertypen

Referenztypen

Elementzeigertypen

Leider wird diese Unterscheidung vom C++Standard **nicht unterstützt** (es fehlen Begriffe für die Typarten, die hier als **definierte Typen** und als **konstruierte Typen** bezeichnet werden).

Entsprechungen zwischen Typarten in **Java** und in **C++**:

C++-Typen und die zugehörigen **Bojen**:

Definierte Typen	Zeigertypen
z.B. int	z.B. int *
z.B. string	z.B. string *

Java-Typen und die zugehörigen **Bojen**:

Primitive Typen	Referenztypen
z.B. int	---
---	z.B. String

Zu einigen Typarten in **C++** gibt es **keine** Entsprechung in **Java**. Das spricht aber eher **für** Java als **gegen** Java.

In Java ist jeder Typ entweder ein **primitiver Typ** oder ein **Referenztyp**. Z.B. ist **int** ein primitiver Typ und **String** ist ein Referenztyp. In C++ gibt es von jedem Typ sozusagen **zwei Versionen**, die eine entspricht einem **primitiven Typ** und die andere einem **Referenztyp** in Java. Z.B. entsprechen dem einen Typ **int** in Java die beiden Typen **int** und **int *** in C++ und dem einen Java-Typ **String** entsprechen die beiden C++-Typen **string** und **string ***.

6. Das Konzept eines Stapels

War in in der Mathematik schon bekannt unter der Bezeichnung **Keller** (seit ?). Kam etwa 1960 in die Informatik im Zusammenhang mit **Algol60**. Die ersten höheren Programmiersprachen **Fortran** und **Cobol** kannten noch **keinen** Stapel. Ein Stapel ist ein Konzept für die **Implementierung** von **Variablen** und Konstanten.

Wie funktioniert ein Stapel? Wie werden die (Parameter und die) **lokalen Variablen** eines Unterprogramms **verwaltet** (erzeugt und wieder zerstört)?

Vorteile eines Stapels (Was ist an Algol60 besser als an Fortran?):

Nur die Variablen **aktiver** Unterprogramme belegen Speicher.

Rekursive Aufrufe von Unterprogrammen werden **möglich**.

Nachteile eines Stapels (Was ist an Fortran besser als an Algol60?):

7. Das Konzept einer rekursiven Funktion

Altes mathematisches Konzept, wurde spätestens ab ca. 1920 formal untersucht (z.B. als Grundkonzept des Lambda-Kalküls), lange vor der Erfindung elektronischer Computer.

Rekursive Gleichungen: Ein mächtiger und eleganter Formalismus für die **Definition von Funktionen**. Beispiel: Basis: Die natürliche Zahl **0** und die Nachfolgerfunktion **succ**. Damit kann man alle natürlichen Zahlen wie folgt darstellen:

0, succ(0), succ(succ(0)), succ(succ(succ(0))), ...

Seien **N** und **M** Variablen, die für beliebige natürliche Zahlen stehen.

Definition einer Funktion **add** durch (rekursive) Gleichungen:

add(N, 0) = N

add(N, succ(M)) = succ(add(N, M))

Definition einer Funktion **mult** durch (rekursive) Gleichungen:

mult(N, 0) = 0

mult(N, succ(M)) = add(M, mult(N, M))

Beispiele für rekursive Funktionen:

Die **Fakultäts**-Funktion rekursiv programmiert.

Wie viele Roboter können **m** Roboter in **n** Tagen bauen?

Auf wie viele Weisen kann man eine natürliche Zahl **als Summe** darstellen?

Einen **String invertieren** ("die Reihenfolge seiner Zeichen umkehren")

Das **Rösselsprungproblem** rekursiv lösen

Traversieren ("Durchklettern") eines binären Baumes

Man kann jede **Schleife** durch ein entsprechendes **rekursives Unterprogramm** ersetzen, z.B. so:

Eine Schleife:

```
1 for (int i=0; i<MAX; i++) {
2     machWas(i);
3 }
```

Vereinbarung einer entsprechenden rekursiven Prozedur:

```
4 void Schleife01(int i) {  
5     if (i>0) Schleife01(i-1);  
6     machWas(i);  
7 }
```

Ein Aufruf der rekursiven Prozedur:

```
8     Schleife01(MAX);
```

8. Funktionales Programmieren

Was bedeutet "**funktional Programmieren**"? Verschiedene Charakterisierungen:

- Nur mit **Funktionen** (ohne Seiteneffekt) programmieren (und **ohne Prozeduren**).
- Nur mit **unveränderbaren Variablen (Konstanten)** programmieren (und **ohne veränderbare Variablen** und **ohne Zuweisungs-Anweisung**)
- Nur mit **Vereinbarungen** und **Ausdrücken** programmieren (und **ohne Anweisungen**)

Funktionale Programmiersprachen: Lisp (in vielen Varianten), Miranda, Scheme, ..., SQL. All diese Sprachen sind "**im Wesentlichen** funktional" und enthalten nur **wenige Anweisungen** (z.B. für die Ein- und Ausgabe).

Die heute üblichen, **nicht-funktionalen** Sprachen (wie Fortran, Cobol, Algol60, Pascal, Ada, C, C++, Java, ...) bezeichnet man auch als **imperative Sprachen** oder als **prozedurale Sprachen**.

Auch in C++ kann man **funktional programmieren**, wenn man unbedingt will. Aber der funktionale Programmierstil wird von C++ nur **wenig** unterstützt. Grundregeln:

- Anstelle von **veränderbaren** Variablen nur **unveränderbare** Variablen (const) verwenden
- Anstelle von **Schleifen Rekursion** verwenden.

Vorteile des funktionalen Programmierstils:

- Programme sind tendenziell **leichter lesbar** (weil man beim "Ausführen im Kopf" keine Veränderungen von Variablen verfolgen muss).
- Die **Reihenfolge**, in der die einzelnen Befehle eines Programms ausgeführt werden, ist **weniger wichtig** als bei imperativen Programmen.
- Deshalb kann man Teile von funktionalen Programmen leichter **nebenläufig** zueinander ("gleichzeitig") **ausführen** als Teile von imperativen Programmen.
- Auf funktionale Programme kann ein Compiler zahlreiche **Optimierungen** anwenden, die bei imperativen Programmen nicht möglich sind.

Nachteile des funktionalen Programmierstils:

- Selbst bei kleinen **Veränderungen** einer großen Datenstruktur (z.B. einer Reihung) muss die ganze Datenstruktur **kopiert** werden.
- Für viele heute tätige Programmierer ist der funktionale Programmierstil **sehr ungewohnt**.

9. Das Konzept eines Moduls

Warum reichen **Unterprogramme** alleine nicht aus, um große Programme **gut zu strukturieren**?
Warum ist es häufig schwer, Unterprogramme (aus einem fertigen Programm in einem neuen Programm) **wiederverwenden**?

Ein **Modul** ist ein **Behälter** für **Variablen, Unterprogramme, ... etc.**, der aus mindestens **zwei** Teilen besteht, einem **sichtbaren Teil** und einem **unsichtbaren Teil**.

Wann und warum ist es wichtig, bestimmte Programmteile **unsichtbar** zu machen? Beispiele?

Die Idee eines **Moduls** kam wohl schon in den 60-er Jahren (des 20. Jahrhunderts) auf, wurde aber erst in den 70-er Jahren ausgearbeitet und in verbreitete Programmiersprachen eingebaut (z.B. **Modula** und **Ada**).

Module in Ada: **Pakete** (packages). Ein Paket besteht aus einer **Spezifikation** (einer Schnittstelle, die alle Informationen für die **Kollegen des Programmierers** enthält) und einem **Rumpf** (der alle Informationen für den **Ausführer** enthält).

Module in C: **Dateien, Variablen** und **Unterprogramme**, die in einem Modul (d.h. in einer Datei) vereinbart werden, gehören standardmäßig zum **sichtbaren Teil** des Moduls (auf sie kann in allen anderen Dateien des Programms zugegriffen werden). Nur wenn man Variable oder Unterprogramme mit **static** kennzeichnet, gehören sie zum **unsichtbaren Teil** des Moduls (auf sie kann dann in anderen Dateien des Programms **nicht** zugegriffen werden). **Konstanten** (mit **const** vereinbarte Variablen) gehören standardmäßig zum **unsichtbaren Teil** des Moduls, nur wenn man sie mit **extern** vereinbart, gehören sie zum **sichtbaren Teil**. Es ist weitgehend üblich, die Schnittstelle eines Moduls als **Kopfdatei** (.h-Datei, header file) zu realisieren.

Module in C++: **Dateien** mit einem **namenlosen Namensraum** (anonymous namespace) und **weiteren Vereinbarungen** (meist von Unterprogrammen) darin. Der **namenlose Namensraum** ist dann der **unsichtbare Teil** des Moduls und die **weiteren Vereinbarungen** bilden den **sichtbaren Teil**.

Module in Java und C++: Eine **Klasse**, die nur **Klassenelemente** (static members) enthält (und keine Objektelemente). Ein solcher Modul besteht nicht immer aus **zwei** Teilen, sondern kann aus bis zu **vier** Teilen mit unterschiedlicher Sichtbarkeit bestehen: **Überall sichtbarer Teil** (public), **weitgehend sichtbarer Teil** (protected), **wenig sichtbarer Teil** (ohne Modifizierer) und **unsichtbarer Teil** (privat).

Interessanter Unterschied zwischen Modulen in Ada und in C++/Java: In Ada kann man die beiden Teile eines Moduls **getrennt voneinander** schreiben, dem Ausführer übergeben ("compilieren") und verwalten. Die **Kollegen des Programmierers** lesen normalerweise nur den **sichtbaren Teil** des Moduls (die Spezifikation). In C++/Java muss man alle Teile eines Moduls an **einer** Stelle des Programms (in der Vereinbarung der betreffenden Klasse) vereinbaren. Ein Kollege, der eine Klassenvereinbarung liest, sieht nicht nur die Teile, die ihn etwas angehen (z.B. den **public**-Teil), sondern auch die anderen Teile (z.B. den **private**-Teil).

Abstrakte Variablen als Modul realisiert. Die abstrakten Variablen sind im **Rumpf** des Moduls (im unsichtbaren Teil) verborgen. Damit können die Kollegen **nicht direkt** auf diese Variablen zugreifen. Der **sichtbare Teil** des Moduls enthält **Unterprogramme**, mit denen man **indirekt** auf die abstrakten Variablen zugreifen und sie bearbeiten kann. Beispiel: Ein **Stapel**-Modul.

10. Das Konzept einer Klasse

Frühe Vorläufer in der Sprache **Simula**, einer Weiterentwicklung von **Algol60**, ca. 1967. Das heute verbreitete Konzept einer Klasse wurde im Zusammenhang mit der Sprache **Smalltalk** entwickelt, von 1970 bis 1980 (im berühmten Forschungslabor der Firma Xerox in Palo Alto, Californien. Heute gehört dieses Labor der Firma Lucent).

Probleme die man mit dem Konzept eines Moduls **nicht** (oder nur unbefriedigend) lösen kann?
Beispiel: Ein **Stapel-Modul** und eine **Stapel-Klasse**.

Eine **Klasse** ist gleichzeitig ein **Modul** und ein **Bauplan für Module**.

Die **Module**, die nach einem solchen Bauplan gebaut wurden, bezeichnet man auch als **Objekte** (oder als **Instanzen**). Z.B. ist ein **String-Objekt** ein Modul, welcher nach dem Bauplan **String** gebaut wurde.

Klassen können andere Klassen **beerben**.

Wichtige Unterschiede zwischen Klassen in **Java** und Klassen in **C++**: **Wie viele** Klassen kann eine C++-Klasse **beerben**? Wie ist das bei Java-Klassen? Welche Form (oder Struktur) hat ein **Diagramm aller Java-Klassen**? Welche Form hat ein **Diagramm aller C++-Klassen**?

11. Das Konzept einer generischen Einheit

Ada (1980) war die erste verbreitete Programmiersprache, in der man **generische Einheiten** programmieren konnte. Inzwischen gibt es generische Einheiten auch in **C++** (dort werden sie auch als **Schablonen**, templates, bezeichnet). In **Java** gibt es (leider! noch?) **keine** generischen Einheiten.

Probleme, die man mit dem Konzept einer Klasse **nicht** (oder nur unbefriedigend) lösen kann? Was ist z.B. unbefriedigend an dem rohen Java-Typ **Vector** (im Gegensatz zu den parametrisierten Typen **Vector<String>**, **Vector<Integer>** etc.)?

Gemeinsamkeiten zwischen **Unterprogrammen** und **generischen Einheiten** in **C++**:

Wenn man ein **Unterprogramm** (bzw. eine **generische Einheit**) **vereinbart**, kann man es (bzw. sie) mit **formalen Parametern** versehen. Wenn man ein **Unterprogramm aufruft** (bzw. eine generische **Einheit instanziiert**) kann man ihm (bzw. ihr) entsprechende **aktuelle Parameter** übergeben.

Unterschied 1 zwischen **Unterprogrammen** und **generischen Einheiten**:

Beim **Aufrufen** eines **Unterprogramms** kann man nur **Werte** (beschrieben durch Ausdrücke) als Parameter übergeben. Beim **Instanzieren** einer generischen Einheit kann man außer Werten auch **Typen** als Parameter übergeben.

Unterschied 2 zwischen **Unterprogrammen** und **generischen Einheiten**:

Aufrufe von **Unterprogrammen** werden während der **Ausführung** eines Programms ausgeführt. **Instanzierungen** einer **generischen Einheit** (und alle damit verbundenen Typprüfungen) werden schon bei der **Übergabe des Programms an den Ausführer** ("zur Compilezeit") durchgeführt.

In **C++** unterscheidet man zwei Arten von generischen Einheiten: **Unterprogrammsschablonen** und **Klassenschablonen**. Jede Instanz einer Unterprogrammsschablone (bzw. einer Klassenschablone) ist ein **Unterprogramm** (bzw. eine **Klasse**).

Mit Hilfe **generischer Einheiten** kann man für viele Probleme sehr **elegante** und **schnelle** Lösungen konstruieren. **Klassenhierarchien** mit vielen Schichten tendieren dazu, Programme

langsam zu machen.

Die **C++-Standardbibliothek** besteht hauptsächlich aus **generischen Einheiten**. Die **Java-Standardbibliothek** besteht dagegen aus **Klassen** und **Schnittstellen**.

Wichtigster Unterschied zwischen generischen Einheiten in C++ und in Java: Wenn man in einem C++-Programm z. B. die Typen **vector<int>**, **vector<string>** und **vector<vector<int>>** benutzt, werden dazu drei entsprechend modifizierte Kopien der Schablone **vector** in das Programm eingebaut. Wenn man in einem Java-Programm z. B. die Typen **Vector<Integer>**, **Vector<String>** und **Vector<Vector<String>>** verwendet, gibt es trotzdem nur **eine** Klasse **Vector** (und nicht drei Kopien davon).

Gute **generische Einheiten** zu programmieren ist besonders **anspruchsvoll** und **schwierig**, wenn es gelingt aber auch besonders effizient und befriedigend.

12. Das Konzept einer nicht-schwachen Funktion

Wurde etwa 1970 von dem Mathematiker Dana Scott als Grundkonzept einer "Mathematical Theory of Computation" eingeführt. Man kann damit u.a. die Bedeutung von **rekursiven Funktionen** auf sehr einsichtige und klare Weise erklären und definieren.

Undefinierte Werte: Zu jedem Datentyp, so kann man sich vorstellen, gehört auch ein **undefinierter Wert**. Dieser Wert, so sagt man, wird von einer Funktion geliefert, die keinen definierten Wert liefert, weil sie z.B. eine Endlosschleife enthält.

Beispiel für ein Funktion, die immer den **undefinierten Wert** des Typs **int** liefert:

```
1 int unDefInt() {
2     while (true) ; // Endlosschleife
3     return 1;      // Dieser Befehl wird nie erreicht
4 }
```

Wenn man die Funktion **unDefInt** aufruft, liefert sie **keinen** (definierten) **int**-Wert zurück. Deshalb sagt man: Die Funktion **unDefInt** liefert den **undefinierten int-Wert**. Wir sagen auch: Der Ausdruck **unDefInt()** bezeichnet den **undefinierten int-Wert**.

Viele Funktionen liefern **nicht immer**, aber doch **manchmal** den undefinierten Wert als Ergebnis, je nachdem, mit welchen Parametern man sie aufruft. Hier ein Beispiel für eine solche Funktion:

```
1 int manchMal(short s) {
2     if (s % 2 == 0) {
3         while (true) ; // Eine Endlosschleife
4         return 0;      // Dieser Befehl wird nie erreicht
5     } else {
6         return 1;
7     }
8 }
```

Wenn man die Funktion **manchMal** auf eine **ungerade** Zahl anwendet, liefert sie den **int**-Wert **1** als Ergebnis. Wendet man sie auf eine **gerade** Zahl an dann liefert sie (so sagt man) den **undefinierten int-Wert**.

Angenommen, **g** ist eine Funktion mit einem **int**-Parameter. Dann kann man **g** auch auf den undefinierten **int**-Wert anwenden, indem man als aktuellen Parameter z.B. **unDefInt()** oder **manchMal(12)** angibt. Frage: Welchen Wert hat der Ausdruck **g(unDefInt())** oder der Ausdruck **g(manchMal(12))** ?

Definition: Eine Funktion heißt **schwach**, wenn sie den **undefinierten Wert** (ihres Ergebnistyps) liefert, sobald mindestens **einer** ihrer Parameter **undefiniert** ist.

In den meisten Programmiersprachen (z.B. in Java und C++) kann der Programmierer **nur schwache** Funktionen programmieren. Das hat folgenden Grund: Wenn ein **Funktionsaufruf** ausgeführt wird, dann werden **zuerst** die Werte der **aktuellen Parameter** berechnet und **danach** wird der **Rumpf** des Unterprogramms ausgeführt. Falls die Berechnung eines Parameterwertes **ewig** dauert (d.h. falls ein Parameterwert **undefiniert** ist), dauert die gesamte Ausführung des Funktionsaufrufs ebenfalls **ewig** (d.h. der Aufruf liefert den **undefinierten Wert**).

In vielen Sprachen gibt es ein paar **vordefinierte Funktionen** (oder ähnliche Konstrukte), die **nicht schwach** sind. In Java und in C++ sind das z.B. die logischen Operatoren **&&** (und) und **||** (oder), denn ein Ausdruck wie z.B. **false && unDefBool()** bezeichnet den Wert **false** (und **nicht** den undefinierten bool-Wert). Entsprechend bezeichnet ein Ausdruck wie **(x == y) || unDefBool()** den Wert **true**, falls x und y gleich sind.

if-Befehle sind in allen Sprachen **nicht-schwache** Konstrukte. In Java und C++ gibt es sogar ein **Ausdrucks-if** (conditional operator), welches alle wichtigen Eigenschaften einer **nicht-schwachen Funktion** hat. Ein Ausdruck wie **(x != y) ? unDefInt() : 27** bezeichnet den Wert **27** bzw. den **undefinierten int-Wert**, je nachdem ob x und y gleich oder ungleich sind.

Was passiert, wenn man einen undefinierten Wert mit sich selbst vergleicht? D. h. welchen Wert bezeichnet ein Ausdruck wie **(unDefInt() == unDefInt())**? Wie könnte man (unabhängig von den Beschränkungen heute üblicher Programmiersprachen) eine **nicht-schwache Multiplikations-Funktion** definieren? Wie könnte man eine **und-Funktion** definieren, die **"echt stärker"** ist als die **&&-Funktion** in Java und C++? Entsprechend für die **oder-Funktion ||**? Wie würde sich eine **starke Multiplikations-Funktion** verhalten? Das Ausdrucks-if ist bezüglich seines **ersten** Parameters (der Bedingung vom Typ **bool**) **schwach**. Könnte man das irgendwie ändern? Wie würde sich ein **starkes Ausdrucks-if** verhalten?

In der Sprache Algol60 konnte der Programmierer selbst **nicht-schwache Funktionen** (z.B. nicht-schwache **Multiplikations-Funktionen**, **und-Funktionen**, **oder-Funktionen** etc.) programmieren. Allerdings konnte man auch in Algol60 **keine starke** Multiplikation, und-Funktion oder-Funktion oder if-Funktion programmieren.

13. Das Konzept eines Zustands

Die Beschreibung einer Programmiersprache (z.B. in einem Standard-Dokument) muss vor allem die **Syntax** und die **Semantik** der Sprache beschreiben. (**Syntax**: "Was darf der Programmierer hinschreiben und was nicht?". **Semantik**: "Was bedeuten die einzelnen Programme, was bewirken sie, was richten sie an?").

Die **Syntax** einer Programmiersprache wird heute meistens durch eine **kontextfreie Grammatik** und ein paar **zusätzliche Regeln** in natürlicher Sprache beschrieben. Beispiele für solche zusätzlichen Regeln: "Bevor man ein Unterprogramm **aufrufen** darf, muss man es **vereinbaren**" oder "Die Anzahl der **aktuellen** Parameter im **Aufruf** eines Unterprogramms muss übereinstimmen mit der Anzahl der **formalen** Parameter in der **Vereinbarung** des Unterprogramms". Syntax-Beschreibungen sind heutzutage (im Jahr 2000) meistens recht **gut**, d.h. genau und verständlich.

Die **Semantik** der meisten Programmiersprachen ist dagegen nur sehr **informell** und **unvollständig** beschrieben.

Das Konzept eines **Zustands** wurde entwickelt, um die **Semantik** von Programmiersprachen (ins-

besondere von imperativen, Algol-ähnlichen Sprachen) **unabhängig** von einer bestimmten Rechnerarchitektur und doch **präzise** zu beschreiben.

Wenn ein Programm ausgeführt wird, befindet es sich in jedem Moment in einem bestimmten **Zustand**. Indem der Ausführer die einzelnen Befehle des Programms ausführt, ändert er diesen Zustand Schritt für Schritt.

Ein **Zustand** besteht aus einer **Umgebung** (environment) und einem abstrakten **Speicher** (storage in UK, memory in USA).

Als Beispiel soll das folgende Programm ausgeführt und dabei sein Zustand dargestellt und schrittweise verändert werden:

Ein Beispielprogramm:

```

1 int main () {
2     int const carl    = 17;
3     int             ilse = 33;
4     int *           zai  = &ilse;
5     int             ralf[3] = {12, 78, 56};
6     *zai = carl + 3;
7     ...
8 } // main

```

Wenn der Ausführer dieses Programm bis einschließlich Zeile 4 ausgeführt hat, sieht der Zustand der Ausführung wie folgt aus:

Umgebung:

Name	Art	Der Name steht für
main	Funktion	int main() {int const carl=17; int ilse=33; ... }
carl	int Konst.	17
ilse	int Var.	[1]
zai	int* Var.	[2]
ralf	int * Var.	[3]

Speicher:

Zeiger	Der Zeiger zeigt auf
[1]	33
[2]	[1]
[3]	12
[4]	78
[5]	56

14. Parameter-Übergabemechanismen

Wie **Parameter** an Unterprogramme **übergeben** werden ist sehr wichtig für die **Effizienz** vieler Programme.

Seit 1960 (als die Entwicklung der ersten höheren Programmiersprache Fortran ungefähr begann) wurden **sehr viele** verschiedene Parameter-Übergabemechanismen entwickelt. Viele davon sind inzwischen wieder **vergessen**. An einige sollte man sich ab und zu wieder mal **erinnern** (z.B. immer dann, wenn übliche Prozessoren wieder um den Faktor 100 schneller geworden sind).

Die Sprachen C++ und Java verwenden im Kern nur **einen** Parameter-Übergabemechanismus: **Übergabe per Wert**. Leider ist die übliche englische Bezeichnung **call by value** sprachlich ziemlich missglückt, besser wäre **pass by value**. Mit Hilfe von Referenztypen (in Java) bzw. Zeigertypen oder Referenztypen (in C++) kann man mit der Übergabe **per Wert** auch eine **Übergabe per Referenz** erreichen.

Kein einzelner Parameter-Übergabemechanismus ist für **alle** Situationen **optimal**. Eine Übergabe **per Wert** ist z.B. dann **ungünstig**, wenn ein Unterprogramm **mehrere** Parameter hat, aber in bestimmten Fällen **nicht alle** davon benötigt werden, etwa so:

```

1 void proz1(int p1, int p2) {
2     if (p1 % 2 == 0) {
3         ... // Hier wird p2 nicht benutzt
4     } else {
5         ... // Hier wird p2 benutzt
6     }
7 }

8 void main() {
9     ...
10    proz1(n, aufWendig() + 3);
11    ...

```

Falls **n** gerade ist, wird in Zeile 10 die Funktion **aufWendig** aufgerufen, aber ihr Ergebnis wird nicht benutzt.

Die Parameterübergabe **per Name** funktioniert etwa so: Als aktuellen Parameter darf man einen beliebig komplizierten **Ausdruck** (des betreffenden Typs) angeben. Dieser Ausdruck wird aber nicht **vor** der Ausführung des Unterprogramm-Rumpfes ausgewertet (wie bei der Übergabe **per Wert**), sondern erst **während** der Ausführung, und zwar jedesmal, wenn der Parameter **benutzt** wird. Falls der Parameter gar **nicht benutzt** wird, wird der entsprechende Ausdruck auch **nicht ausgewertet**. Für das obige Beispiel-Unterprogramm **proz1** wäre eine Übergabe des zweiten Parameters **p2 per Name** günstig (der erste Parameter könnte ebenfalls **per Name** oder aber **per Wert** übergeben werden).

In **Algol60** konnte man Parameter wahlweise **per Name** oder **per Wert** übergeben. Mit dem Übergabemechanismus **per Name** kann man bestimmte **nicht-schwache** Funktionen programmieren.

Statt **per Name** sollte der Mechanismus besser **per Ausdruck** heißen. Weil dieser Parameter-Übergabemechanismus relativ **schwer zu implementieren** war, wurde er nach Algol60 in keine andere Sprache mehr aufgenommen. Inzwischen hat es allerdings erhebliche Fortschritte auf dem Gebiet des Compilerbaus gegeben.

In welchen Situationen ist eine Übergabe per Name **nachteilig**? Wie könnte man den Übergabemechanismus verändern, um diese Nachteile zu **beseitigen**?

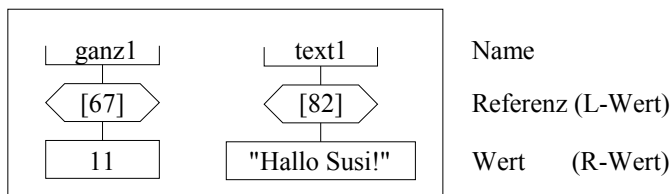
Bei **verteilten Programmen** (bei denen die einzelnen Programmteile von **verschiedenen Rechnern** ausgeführt werden) ergeben sich bei der Übergabe von Parametern an Unterprogramme ganz neue Probleme. Eine Übergabe **per Zeiger** wird problematisch oder unmöglich, wenn der aufrufende Programmteil von **einem** Rechner und das aufgerufene Unterprogramm von einem **anderen** Rechner ausgeführt wird. Eine Übergabe **per Wert** ist bei verteilten Programmen weniger problematisch.

15. Anhang A: Beispiele für Bojen

In den folgenden Beispielen werden (in der Sprache C++) jeweils ein paar Variablen **vereinbart** und dann **als Bojen dargestellt**.

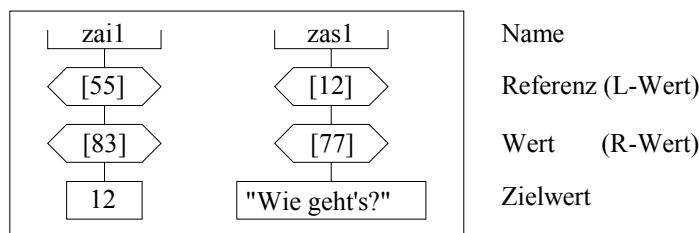
15.1. Bojen für zwei Variablen definierter Typen

```
1 int    ganz1 = 11;
2 string text1 = "Hallo Susi!";
```



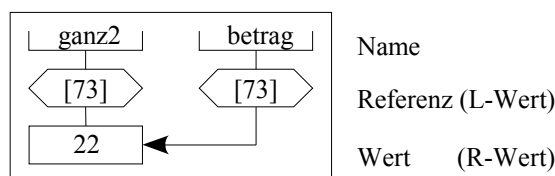
15.2. Bojen für zwei Zeiger-Variablen

```
3 int    * zai1 = new int(12);           // Zeiger auf int-Variable
4 string * zasi = new string("Wie geht es?"); // Zeiger auf string-Variable
```



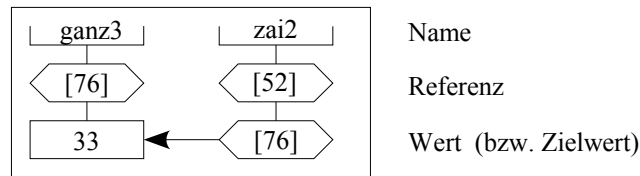
15.3. Bojen für zwei Variablen mit gleichen Referenzen und identischem Wert

```
5 int    ganz2 = 22;
6 int & betrag = ganz2;
```



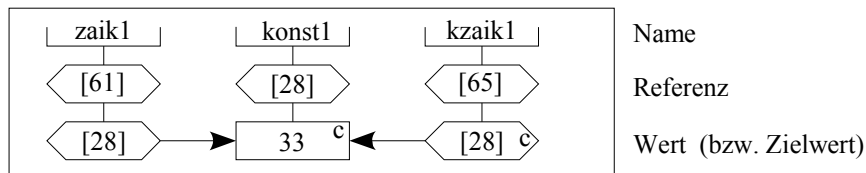
15.4. Bojen für zwei Variablen mit identischem Wert bzw. Zielwert

```
7 int ganz3 = 33;
8 int * zai2 = &ganz3; // Zeiger auf int
```



15.5. Bojen für Konstanten, Zeiger auf Konstanten und konstante Zeiger

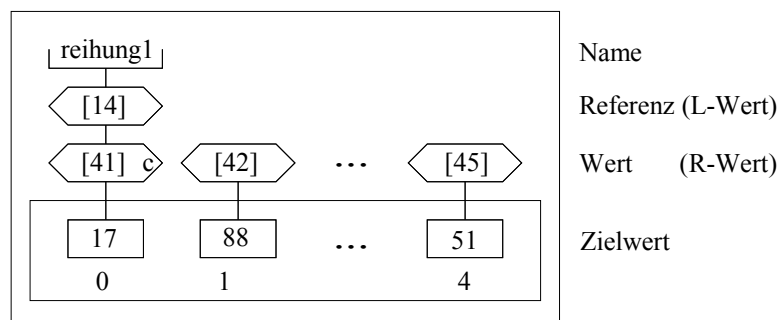
```
9 const int konst1 = 33;
10 const int * zaik1 = &konst1; // Zeiger auf int-Konst.
11 const int * const kzaik1 = &konst1; // konst. Zeiger auf int-Konst.
```



Der kleine Buchstabe **c** kennzeichnet Kästchen, auf die der Programmierer eigentlich Schreibzugriff hat, deren Inhalt er aber trotzdem nicht verändern darf. Obwohl sie unveränderbar sind, werden die **Referenzen** von Variablen **nicht** mit einem **c** gekennzeichnet, weil der Programmierer **nie Schreibzugriff** auf sie hat.

15.6. Boje für eine Reihung

```
12 int reihung1[5] = {17, 88, 35, 67, 51};
```



15.7. Bojen für ein Objekt

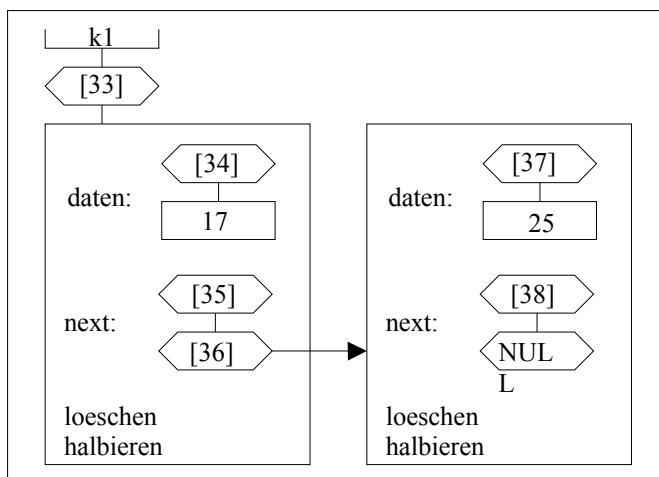
Die Objekte einer Klasse sollte man je nach Zusammenhang "ausführlich und vollständig" oder "ein bisschen vereinfacht" darstellen. Betrachten wir als Beispiel die folgenden Vereinbarungen:

```

13 struct Knoten {
14     // Klassen-Elemente:
15     Knoten(int daten=0, Knoten * next=NULL) : daten(daten), next(next) {}
16
17     // Objekt-Elemente:
18     int     daten;
19     Knoten * next;
20     int loeschen() {int erg=daten; daten=0; return erg;}
21     void halbiere() {daten /= 2;}
22 }; // struct Knoten
23
24 Knoten k1(17, new Knoten(25, NULL));
25

```

Die Klasse **Knoten** enthält 1 Klassen-Element (einen **Konstruktor**), 2 Objekt-Attribute (**daten** und **next**) und 2 Objekt-Methoden (**loeschen** und **halbieren**). Hier eine "ausführliche und vollständige" Darstellung der Variablen **k1** als Boje:



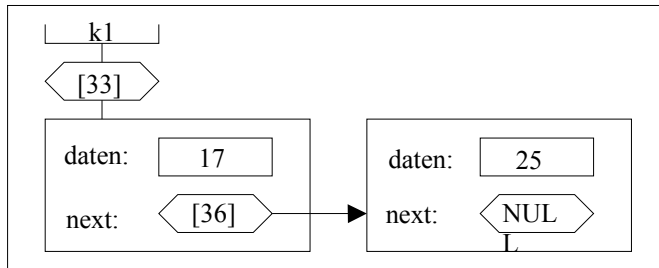
Man sieht hier 2 Knoten-Variablen. Die erste hat einen **Namen** (**k1**) und die **Referenz** [33], die zweite hat keinen einfachen Namen und die **Referenz** [36]. Diese zweite Knoten-Variable kann man mit dem zusammengesetzten Namen ***(k1.next)** bezeichnen.

Die **Attribute** einer Objekt-Variablen sind selbst auch **Variablen** und bestehen (wie alle Variablen) aus einer **Referenz** und einem **Wert**. Diese Attribut-Variablen haben aber **keine einfachen Namen**. Man kann sie über die zusammengesetzten Namen **k1.daten**, **k1.next**, **k1.next->daten** und **k1.next->next** bezeichnen. Die Variable **k1.daten** hat im Beispiel die **Referenz** [34] und den **Wert** 17, die Variable **k1.next->next** hat die **Referenz** [38] und den Wert NULL etc..

Dem Namen **daten** ist **keine** (eindeutige) **Referenz** zugeordnet. Daraus folgt, dass ein Ausdruck wie **&daten** ("Referenz von Daten") verboten und **daten** somit **kein Variablen-Name** ist. Deshalb darf **daten** auch nicht in einem "Paddelboot" stehen wie z.B. der Variablen-Name **k1** (dem eindeutig die Referenz [33] zugeordnet ist). Entsprechendes gilt auch für den Namen **next**. Man bezeichnet **daten** und **next** als **Attribut-Namen** oder als **relative Namen** (weil sie nur **relativ** zu einer bestimmten Variablen eine Bedeutung haben), um sie von **Variablen-Namen** zu

unterscheiden..

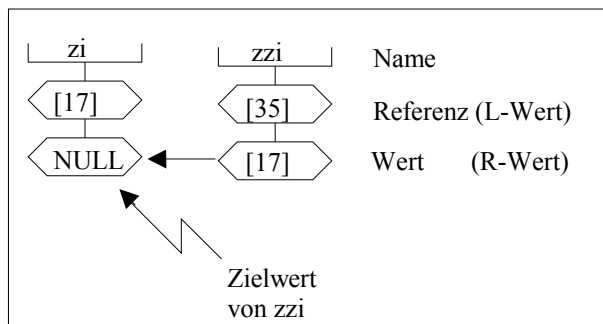
In obiger Bojen-Darstellung der beiden Knoten-Variablen sind auch die **Objekt-Methoden** (**loeschen** und **halbieren**) eingezeichnet. Die kann man meistens **weglassen**. Die **Referenzen der Attribute** ([34], [35], [36], [37]) kann man ebenfalls **weglassen**, wenn man nicht gerade Ausdrücke wie **&k1.daten**, **&k1.next**, **&k1.next->daten** oder **&k1.next->next** erklären will. Hier eine vereinfachte Bojen-Darstellung derselben Variablen wie oben:



15.8. Eine Zeigervariable, die auf NULL zeigt

Im folgenden Beispiel zeigt die Zeigervariable **zzi** auf NULL:

```
26 int *   zi = NULL; // Eine Zeigervariable mit dem Wert NULL
27 int * * zzi = &zi; // Eine Zeigervariable, die auf NULL zeigt
```



Man beachte den feinen Unterschied: Die Variable **zi** **hat den Wert NULL** und hat somit **keinen** Zielwert. Die Variable **zzi** (vom Typ **Zeiger auf Zeiger auf int**) hat den **Wert [17]** und den **Zielwert NULL**, d.h. **zzi "zeigt auf NULL"**.

16. Anhang B: Ein paar Beispielprogramme

16.1. Ein Stapel-Modul

```

1 // Datei StapelM.h
2 /* -----
3 Deklarationsteil (sichtbarer Teil) des Moduls StapelM. Stellt seinen
4 Benutzern folgende Größen für die Bearbeitung eines char-Stapels zur
5 Verfügung: eine Konstante STAPEL_GROESSE, zwei Ausnahmeklassen (Unterlauf,
6 Ueberlauf) und drei Unterprogramme (push, pop, top).
7 Der Stapel selbst ist im Rumpf des Moduls (im unsichtbaren Teil, in der
8 Datei StapelM.cpp) verborgen und damit vor direkten Zugriffen geschuetzt.
9 ----- */
10 #ifndef StapelM_h // Diese Datei (StapelM.h) wird hoechstens einmal in jede
11 #define StapelM_h // andere Datei kopiert (mit #include).
12
13 #include <string>
14 using namespace std;
15 // -----
16 const unsigned STAPEL_GROESSE = 6; // Absichtlich ziemlich klein
17 // -----
18 class Unterlauf {
19 public:
20     string melde(); // Liefert den Text einer Fehlermeldung
21 }; // class Unterlauf
22 // -----
23 class Ueberlauf {
24 public:
25     Ueberlauf(char c = '?'); // Standard- und allgemeiner Konstruktor
26     string melde(); // Liefert den Text einer Fehlermeldung
27 private:
28     const char ueberZaehlig; // Zeichen, welches nicht mehr auf den
29                             // Stapel passt.
30 }; // class Ueberlauf
31 // -----
32 // Unterprogramme zum Bearbeiten des Stapels (der im Rumpf des Moduls
33 // StapelM verborgen ist):
34 void push(char c) throw (Ueberlauf);
35 // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgelost.
36 // Sonst wird der Inhalt von c auf den Stapel gelegt.
37 void pop() throw (Unterlauf);
38 // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgelost.
39 // Sonst wird das oberste Element vom Stapel entfernt.
40 char top() throw (Unterlauf);
41 // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgelost.
42 // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
43 // bleibt dabei unveraendert).
44 // -----
45
46 #endif // ifndef StapelM_h

47 // Datei StapelM.cpp
48 /* -----
49 Rumpf (unsichtbarer Teil, Definitionsteil) des Moduls StapelM.
50 ----- */
51 #include "StapelM.h"
52 // -----
53 // Der Stapel und seine Indexvariable efi (erster freier Index):
54 static char stapel[STAPEL_GROESSE];
55 static unsigned efi = 0; // Anfangs ist der Stapel leer
56 // -----
57 // Definitionen der Konstruktoeren und Unterprogramme der geschachtelten
58 // Klassen Unterlauf und Ueberlauf:
59
60 string Unterlauf::melde() {
61     return "Der Stapel ist leer!";

```

```
62 } // Unterlauf::melde
63 // -----
64 Ueberlauf::Ueberlauf(char c) : ueberZaehlig(c) {}
65
66 string Ueberlauf::melde() {
67     const string TEXT1 = "Das Zeichen ";
68     const string TEXT2 = " passt nicht mehr auf den Stapel!";
69     return TEXT1 + ueberZaehlig + TEXT2;
70 } // Ueberlauf::melde
71 // -----
72 // Definitionen der Unterprogramme zum Bearbeiten des Stapels:
73 void push(char c) throw (Ueberlauf) {
74     if (efi >= STAPEL_GROESSE) {
75         throw Ueberlauf(c);
76     }
77     stapel[efi++] = c;
78 } // push
79 // -----
80 void pop() throw (Unterlauf) {
81     if (efi < 0) {
82         throw Unterlauf();
83     }
84     efi--;
85 } // pop
86 // -----
87 char top() throw (Unterlauf){
88     if (efi < 1) {
89         throw Unterlauf();
90     }
91     return stapel[efi-1];
92 } // top
93 // -----
```

16.2. Eine Stapel-Klasse

```

1 // Datei StapelK.h
2 /* -----
3 Deklaration der Klasse Stapel. Jedes Objekt dieser Klasse ist ein Stapel,
4 auf den man char-Werte legen kann. Die Groesse des Stapels kann man beim
5 Vereinbaren des Objekts angeben (Standardwert ist 6).
6 ----- */
7 #ifndef StapelK_h
8 #define StapelK_h
9
10 #include <string>
11 using namespace std;
12 // -----
13 class Stapel {
14 public:
15     // -----
16     // Ein Standard- und allgemeiner Konstruktor:
17     Stapel(const unsigned groesse = 6);
18     // -----
19     // Destruktor, muss sein, weil privatStapel ein Zeiger ist:
20     ~Stapel();
21     // -----
22     // Objekte der folgenden beiden Klassen werden als Ausnahmen geworfen:
23
24     class Unterlauf {
25     public:
26         string melde();           // Liefert den Text einer Fehlermeldung
27     }; // class Unterlauf
28
29     class Ueberlauf {
30     public:
31         Ueberlauf(char c = '?'); // Standard- und allgemeiner Konstruktor
32         string melde();         // Liefert den Text einer Fehlermeldung
33     private:
34         const char ueberZaehlig; // Zeichen, welches nicht mehr auf den
35                                     // Stapel passt.
36     }; // class Ueberlauf
37     // -----
38     // Unterprogramme zum Bearbeiten des Stapels privatStapel
39     void push(char c) throw (Ueberlauf);
40     // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgelost.
41     // Sonst wird der Inhalt von c auf den Stapel gelegt.
42     void pop()          throw (Stapel::Unterlauf);
43     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgelost.
44     // Sonst wird das oberste Element vom Stapel entfernt.
45     char top()         throw (Stapel::Unterlauf);
46     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgelost.
47     // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
48     // bleibt dabei unveraendert).
49     // -----
50 private:
51     const unsigned GROESSE;      // Wird vom Konstruktor initialisiert
52     char * const privatStapel; // Zeigt auf Reihung mit GROESSE Komponenten
53     unsigned efi;                // erster freier Index von privatStapel
54 // -----
55 }; // class Stapel
56
57 #endif // ifndef StapelK_h

```

```
58 // Datei StapelK.cpp
59 /* -----
60 Implementierung der Klasse Stapel (und der geschachtelten Klassen
61 Stapel::Unterlauf und Stapel::Ueberlauf).
62 ----- */
63 #include "StapelK.h"
64 #include <string>
65 using namespace std;
66
67 // -----
68 // Definitionen von Konstruktor und Destruktor der Klasse Stapel:
69 Stapel::Stapel(const unsigned G)
70     : GROESSE(G), efi(0), privatStapel(new char[GROESSE]) {
71 }
72
73 Stapel::~Stapel() {delete [] privatStapel;}
74 // -----
75 // Definitionen fuer die geschachtelten Klassen Unterlauf und Ueberlauf:
76
77 Stapel::Ueberlauf::Ueberlauf(char c) : ueberZaehlig(c) {}
78 // -----
79 string Stapel::Unterlauf::melde() {
80     return "Der Stapel ist leer!";
81 } // melde
82
83 string Stapel::Ueberlauf::melde() {
84     const string TEXT1 = "Das Zeichen ";
85     const string TEXT2 = " passt nicht mehr auf den Stapel!";
86     return TEXT1 + ueberZaehlig + TEXT2;
87 } // melde
88 // -----
89 // Definitionen der Unterprogramme zum Bearbeiten des Stapels:
90 void Stapel::push(char c) throw (Ueberlauf) {
91
92     if (efi >= GROESSE) {
93         throw Ueberlauf(c);
94     }
95     privatStapel[efi++] = c;
96 } // push
97 // -----
98 void Stapel::pop()          throw (Unterlauf) {
99     if (efi < 0) {
100         throw Unterlauf();
101     }
102     efi--;
103 } // pop
104 // -----
105 char Stapel::top() throw (Unterlauf){
106     if (efi < 1) {
107         throw Unterlauf();
108     }
109     return privatStapel[efi-1];
110 } // top
111 // -----
112 // Damit sind alle Elemente der Klasse Stapel definiert
```

16.3.

16.4. Eine Stapel-Schablone

```

1 // Datei Stapels.h
2 /* -----
3 Deklaration der Schablone Stapel. Jede Instanz dieser Schablone ist eine
4 Klasse. Jedes Objekt einer solchen Klasse ist ein Stapel, auf den man Werte
5 des Typs ElemTyp legen kann. Diesen ElemTyp kann man beim Instanziiieren
6 der Schablone angeben.
7 ----- */
8 #ifndef Stapels_h
9 #define Stapels_h
10
11 #include <string>
12 using namespace std;
13 // -----
14 template <typename ElemTyp> // Statt class ist auch typename erlaubt
15 class Stapel {
16 public:
17     // -----
18     // Ein Standard- und allgemeiner Konstruktor:
19     Stapel(const unsigned groesse = 6);
20     // -----
21     // Destruktor, muss sein, weil privatStapel ein Zeiger ist:
22     ~Stapel();
23     // -----
24     // Objekte der folgenden beiden Klassen werden als Ausnahmen geworfen:
25
26     class Unterlauf {
27     public:
28         string melde(); // Liefert den Text einer Fehlermeldung
29     }; // class Unterlauf
30
31     class Ueberlauf {
32     public:
33         Ueberlauf(ElemTyp e); // Allgemeiner Konstruktor
34         string melde(); // Liefert den Text einer Fehlermeldung
35     private:
36         const
37         ElemTyp ueberZaehlig; // Element, welches nicht mehr auf den
38                               // Stapel passt.
39     }; // class Ueberlauf
40     // -----
41     // Unterprogramme zum Bearbeiten des Stapels privatStapel
42     void push(ElemTyp c) throw (Ueberlauf);
43     // Falls der Stapel voll ist, wird die Ausnahme Ueberlauf ausgeloeset.
44     // Sonst wird der Inhalt von c auf den Stapel gelegt.
45     void pop() throw (Stapel::Unterlauf);
46     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloeset.
47     // Sonst wird das oberste Element vom Stapel entfernt.
48     ElemTyp top() throw (Stapel::Unterlauf);
49     // Falls der Stapel leer ist, wird die Ausnahme Unterlauf ausgeloeset.
50     // Sonst wird eine Kopie des obersten Elementes geliefert (der Stapel
51     // bleibt dabei unveraendert).
52     // -----
53 private:
54     const unsigned GROESSE; // Wird vom Konstruktor initialisiert
55     ElemTyp * privatStapel; // Zeigt auf Reihung mit GROESSE Komponenten
56     unsigned efi; // erster freier Index von privatStapel
57 // -----
58 }; // class Stapel
59 /* -----
60 Diese Datei Stapels.h enthaelt die *Deklaration* der Schablone Stapel. Die
61 Datei Stapels.cpp enthaelt die *Definitionen* aller Methoden und Konstruk-
62 toren dieser Schablone. Unter dem Inklusions-Modell (ohne Schluesselwort
63 export) muessen in jeder Datei, in der die Schablone instanziiert wird,
64 die Deklaration der Schablone und alle zugehoerigen Definitionen enthalten
65 sein. Deshalb wird im folgenden hier die Definitionsdatei Stapels.cpp

```

```

66 inkludiert. Eine Anwendungsdatei muss dann nur noch diese Deklarationsdatei
67 Stapels.h inkludieren (und nicht etwa beide, Stapels.h und Stapels.cpp).
68 ----- */
69 #include "Stapels.cpp" // Um in Anwendungsdateien include-Befehle zu sparen
70
71 #endif // ifndef Stapels_h

72 // Datei Stapels.cpp
73 /* -----
74 Implementierung der Klassenschablone Stapel (und der geschachtelten
75 Klassen Stapel::Unterlauf und Stapel::Ueberlauf).
76 ----- */
77 #include <string>
78 using namespace std;
79
80 // -----
81 // Definitionen von Konstruktor und Destruktor der Klasse Stapel:
82 template <typename ElemTyp>
83 Stapel<ElemTyp>::Stapel(const unsigned G)
84     : GROESSE(G), efi(0), privatStapel(new ElemTyp[GROESSE]) {
85 }
86
87 template <class ElemTyp>
88 Stapel<ElemTyp>::~~Stapel() {delete [] privatStapel;}
89 // -----
90 // Definitionen fuer die geschachtelten Klassen Unterlauf und Ueberlauf:
91
92 template <class ElemTyp>
93 Stapel<ElemTyp>::Ueberlauf::Ueberlauf(ElemTyp e) : ueberZaehlig(e) {}
94 // -----
95 template <class ElemTyp>
96 string Stapel<ElemTyp>::Unterlauf::melde() {
97     return "Der Stapel ist leer!";
98 } // melde
99
100 template <class ElemTyp>
101 string Stapel<ElemTyp>::Ueberlauf::melde() {
102     return "Der Stapel ist voll!";
103 } // melde
104 // -----
105 // Definitionen der Unterprogramme zum Bearbeiten des Stapels:
106 template <class ElemTyp>
107 void Stapel<ElemTyp>::push(ElemTyp e) throw (Ueberlauf) {
108
109     if (efi >= GROESSE) {
110         throw Ueberlauf(e);
111     }
112     privatStapel[efi++] = e;
113 } // push
114 // -----
115 template <class ElemTyp>
116 void Stapel<ElemTyp>::pop() throw (Unterlauf) {
117     if (efi < 0) {
118         throw Unterlauf();
119     }
120     efi--;
121 } // pop
122 // -----
123 template <class ElemTyp>
124 ElemTyp Stapel<ElemTyp>::top() throw (Unterlauf){
125     if (efi < 1) {
126         throw Unterlauf();
127     }
128     return privatStapel[efi-1];
129 } // top
130 // -----
131 // Damit sind alle Elemente der Klassenschablone Stapel definiert

```

Die Skriptsprache Perl

1. Grundlegende Eigenschaften

Wenn ein Programmierer ein Perl-Programm geschrieben hat, kann er es direkt von einem Perl-Interpreter ausführen lassen (ohne sich um das Compilieren und Binden seines Programms zu kümmern). Typische Perl-Interpreter compilieren ein Programm (zumindest teilweise), bevor sie es ausführen, aber so, dass der Programmierer "nichts davon merkt".

Perl gibt es für **alle verbreiteten Plattformen** (z.B. für Linux, Solaris und andere Unix-Varianten, für den Macintosh, Dos, alle Windows-Varianten, OS/2 etc.). Die Perl-Interpreter auf den verschiedenen Plattformen **stimmen gut überein**, so dass Perl-Programme gut **portierbar** sind (etwa so gut wie Java-Programme, deutlich besser als C++-Programme). Perl besteht aus einer **mittel-grossen Kernsprache** und zahlreichen Spracherweiterungen (für verschiedene Anwendungsgebiete), die im Internet in Form von **Perl-Modulen** frei zur Verfügung stehen. In Perl kann man insbesondere **Internetverbindungen** aufbauen und kontrollieren und **graphische Benutzeroberflächen** (Grabos, mit Fenstern, Knöpfen, Menüs etc.) programmieren. Perl eignet sich **nicht** für die **hardwarenahe** Programmierung oder für Anwendungen, bei denen die **Ausführungseffizienz** im Vordergrund steht.

2. Der Zwei-Sprachen-Programmierstil

Ein Anwendungssystem, welches in diesem Stil programmiert wurde, besteht aus zwei Arten von Bausteinen, die hier als **Module** und **Skripte** bezeichnet werden. Die Module programmiert man in einer **typischeren, Ausführungs-effizienten** Sprache (z.B. in C++), die Skripte dagegen in einer **Programmierer-effizienten** Sprache (z.B. in Perl). Die Module lösen die abstrakten Grundprobleme des Systems. Die Skripte dienen dazu, die Module miteinander zu **verbinden** und enthalten alle Befehle, die man vermutlich häufig ändern und anpassen will (z.B. die graphische Benutzeroberfläche). Perl und ähnliche Skriptsprachen werden manchmal auch als **Klebstoff-Sprachen** (glue languages) bezeichnet, weil man damit **Module zusammenkleben** kann.

3. Perl-Konstrukte, die dem Programmierer das Leben (manchmal) erleichtern

Elementare Typen wie **integer, float, boolean** etc. spielen keine wichtige Rolle. Im Prinzip kann **jede** elementare **Variable jeden** elementaren **Wert** enthalten.

Stringwerte wie "123" und **Zahlenwerte** wie 123 werden **automatisch** ineinander **umgewandelt**, wenn es sinnvoll ist (z.B. in Befehlen wie "**123**" + 17 oder **print 123;**).

Es gibt nur **zwei** zusammengesetzte Typen: **Reihungen** (arrays) und **assoziative Tabellen** (hashes). Die Komponenten einer Reihung müssen **nicht** zum selben Typ gehören. Auf die Komponenten einer Reihung greift man mit **Indizes** zu (ganz ähnlich wie in Java und C). Eine assoziative Reihung besteht aus **Einträgen**. Jeder Eintrag besteht aus einem **Schlüssel** und einem damit verbundenen (assozierten) **Wert**. Der Schlüssel und der Wert können im Prinzip beliebige Datenelemente sein. Auf die Werte in einer assoziativen Tabelle greift man also mit Schlüsseln zu. Verbunde (records) kann man leicht mit assoziativen Tabellen "nachempfinden".

Reihungen und **Tabellen** kann man jederzeit **verlängern**. Hat man z.B. eine Reihung der Länge 10, dann kann man ohne weiteres der 20. Komponente dieser Reihung einen Wert zuweisen. Die Reihung wird dadurch automatisch auf die Länge 20 vergrößert.

In vielen Perl-Befehlen kann man **Parameter** angeben (wie in anderen Sprachen auch). In Perl kann

man diese Parameter in vielen Fällen auch **fortlassen** und der Befehl bezieht sich dann automatisch auf bestimmte **Standardvariablen**. Ist z.B. **@reihung01** eine Reihung, dann kann man mit dem Befehl **shift @reihung01**; die erste Komponente daraus entfernen. Ohne Parameter entfernt der Befehl **shift**; die erste Komponente aus der Reihug **@_** (deren Name im wesentlichen nur aus einem Unterstrich besteht).

Variablen braucht man **nicht zu vereinbaren**, man kann sie gleich benutzen. Da dadurch jeder (aus Versehen) falsch geschriebene Variablen-Name zur Erzeugung einer neuen Variablen führt, kann der Programmierer sich (mit dem Befehl **use strict**;) selbst dazu verpflichten, alle Variablen zu vereinbaren. Sehr empfehlenswert bei nicht-trivialen Programmen.

Die **Parameter** eines Unterprogramms brauchen **nicht** formal beschrieben oder vereinbart zu werden. Wenn ein Unterprogramm aufgerufen wird, bekommt es eine (im Prinzip beliebig lange) **Liste von aktuellen Parametern**. Die Länge dieser Liste kann von Aufruf zu Aufruf auch **verschieden** sein. Nur durch spezielle Angaben und zusätzliche Prüfungen kann der Programmierer erreichen, dass ein bestimmtes Unterprogramm mit einer **bestimmten** Anzahl von Parametern aufgerufen werden **muss**.

Viele Befehle, die in anderen Sprachen einfach als **Fehler** gelten, sind in Perl **erlaubt** und bewirken etwas, was in vielen Fällen **sinnvoll** ist. Eine Zuweisung zwischen zwei **Reihungsvariablen** wie z.B. **@r01 = @r02**; ist in vielen Sprachen erlaubt (und bewirkt in Perl, dass der Variablen **@r01** eine **Kopie** der Reihung **@r02** zugewiesen wird). Eine Zuweisung einer **Reihungsvariablen** an eine **elementare Variable** ist in vielen Sprachen **verboten**. Dagegen ist in Perl z.B. die Zuweisung **\$n = @r01**; erlaubt und bewirkt, dass der Variablen **\$n** die **Länge** der Reihung **@r01** zugewiesen wird.

4. Einfache Beispiel-Programme

Das unvermeidliche Hallo-Programm ist in Perl deutlich kürzer und einfacher als in vielen anderen Sprachen. Es besteht im einfachsten Fall nur aus einer Zeile:

```
1 print "Hallo Welt!\n";
```

Angenommen, das folgende Perl-Programm steht in einer Datei namens **prog01.pl**:

```
2 $muster = shift;  
3 while (<>) {print if !$muster/};  
4 print "\n";
```

Dann kann man dieses Programm z.B. so aufrufen:

```
c:\user> perl prog01.pl abc dat1.txt dat2.txt prog01.pl
```

Hier ist **perl** der Name des Perl-Interpreters, **prog01.pl** der Name des zu interpretierenden Programms, **abc** eine Zeichenkette und **dat1.txt**, **dat2.txt** und **prog01.pl** sind Dateinamen (**prog01.pl** ist die Datei, in der das Perl-Programm selbst drinsteht). Das Programm **prog01.pl** wird hier also mit 4 **Kommandozeilenparametern** (**abc** bis **prog01.pl**) aufgerufen. Dadurch passiert folgendes:

Die Zeichenkette **abc** wird in der Variablen **\$muster** gespeichert (der **shift**-Befehl ohne Parameter entfernt die erste Komponente aus der Reihung der Kommandozeilenparameter und liefert diesen ersten Parameter als Ergebnis).

Die drei übrigen Kommandozeilenparameter (**dat1.txt** bis **prog01.pl**) werden als Dateinamen

interpretiert, die entsprechenden Dateien werden eine nach der anderen geöffnet und zeilenweise gelesen (dies geschieht auf Grund des Befehls **while** ($\langle \rangle$)). Jede Zeile, in der das Muster **abc** (der Inhalt der Variablen **\$muster**) vorkommt, wird zum Bildschirm ausgegeben (dies geschieht auf Grund des Befehls **{print if /\$muster/}**).

Der Bildschirmzeiger (cursor) wird zum Anfang der nächsten Zeile vorgeschoben (durch den Befehl **print "\n"**).

Um das gleiche Problem statt in **Perl** z.B. in **C++** oder in **Java** zu lösen, müsste man **deutlich mehr Befehle** hinschreiben.

Hier noch ein einfaches Perl-Programm, in dem eine **Reihung** verwendet wird:

```
1 # Datei Rueckwaerts.pl
2 # -----
3 # Erwartet als ersten (und einzigen) Kommandozeilen-Parameter den Pfad
4 # einer (Text-) Datei. Liest alle Zeilen der Datei in eine Reihung und
5 # gibt sie "in umgekehrter Reihenfolge" aus (zuerst die letzte Zeile der
6 # Datei, dann die vorletzte Zeile, ..., zuletzt die erste Zeile)
7 # -----
8 # Falls das Oeffnen der Datei nicht klappt, Abbruch mit Fehlermeldung:
9 open IN, $ARGV[0] or die "Kann $ARGV[0] nicht oeffnen!\n";
10
11 # Alle Zeilen der Eingabedatei IN in die Reihung @dieGanzeDatei lesen:
12 @dieGanzeDatei = <IN>;
13
14 # Die Komponenten der Reihung @dieGanzeDatei ausgeben:
15 for ($meinIndex = @dieGanzeDatei - 1; $meinIndex >= 0; $meinIndex--) {
16     print $dieGanzeDatei[$meinIndex];
17 };
18 # -----
```

Man sieht hier, dass der Befehl zum **Lesen** aus einer Datei (in Zeile 12) wie eine **Zuweisung** notiert wird. Um alle Zeilen einer Datei zu lesen ist hier **keine** Schleife erforderlich, die Angabe einer Reihung als Ziel des Lesebefehls genügt. In diesem Programm wurden die Variablen **\$dieGanzeDatei** und **\$meinIndex** **nicht vereinbart**, sondern einfach **benutzt**.

5. Kompliziertes hinter Einfachem verstecken (ties)

Eine **assoziative Tabelle** (hash) enthält **Einträge**. Jeder Eintrag besteht aus einem **Schlüssel** und einem **Wert**. Ist `@TelefonNrn` eine assoziative Tabelle und ist `"Meier"` ein geeigneter Schlüssel, dann bezeichnet `$TelefonNrn->"Meier"` den Wert, der in der Tabelle mit dem Schlüssel "Meier" verbunden ist. Das könnte z.B. die Telefon-Nr von Frau Meier sein.

Unter Windows ist die **Registatur** (registry) eine Datenstruktur, die Ähnlichkeit mit einer **assoziativen Tabelle** hat. Der Perl-Modul `Win32::TieRegistry` bindet die Windows-Registatur an eine Perl-assoziative-Tabelle namens `@Registry`. Über diese assoziative Tabelle kann der Perl-Programmierer bequem auf die Windows-Registatur zugreifen, wie das folgende Perl-Skript demonstriert:

```

1 # Datei Registry01.pl
2 use Win32::TieRegistry;
3 # -----
4 # Beispiel fuer Zugriffe auf die Windows-Registry.
5 # -----
6 $beGrenzerZeichen = $Registry->Delimiter("/");
7
8 $colorInfo        = $Registry->{"HKEY_LOCAL_MACHINE/Software/"
9   "Microsoft/Windows/CurrentVersion/explorer/CSSFilters/ColorInfo"};
10
11 $javaDir          = $Registry->{'HKEY_LOCAL_MACHINE/Software/'
12   'JavaSoft/Java Development Kit/1.3.0/JavaHome'};
13
14 print "beGrenzerZeichen: ", $beGrenzerZeichen, "\n";
15 print "colorInfo:       ", $colorInfo,          "\n";
16 print "javaDir:         ", $javaDir,            "\n";
17 # -----
18 # Ausgabe des Perl-Skripts Registry01.pl:
19 #
20 # beGrenzerZeichen: \
21 # colorInfo:       {D3D6C6B1-DDCF-11D0-9048-00A0C90640B8}
22 # javaDir:         c:\jdk
23 # -----

```

Nebenbei: Vor Einführung der Registatur (registry) musste man manchmal mehrere Stunden warten, bis ein Windows-Betriebssystem abstürzte. **Mit** der Registatur kann man ein System jetzt fast beliebig schnell zum Absturz bringen. Vor allem aber kann man schon durch kleine Eingriffe in die Registatur ein Windows-Betriebssystem so beschädigen, dass es danach nicht mehr ordnungsgemäß hochfährt und auf umständliche Weise repariert werden muss.

Allgemein kann man in Perl eine **Variable** (insbesondere eine assoziative Tabelle) an ein **Paket** (mit Unterprogrammen darin) binden. Wenn man dann lesend bzw. schreibend auf die Variable zugreift, werden entsprechende Unterprogramme in dem Paket aufgerufen und sind dafür zuständig, einen geeigneten "Lese-Wert" zu liefern bzw. den "Schreib-Wert" irgendwie abzuspeichern. Mit dieser Technik kann man z.B. eine beliebige Stelle im Internet mit einer Perl-Variablen verbinden. Wenn man im Perl-Skript dann auf die Variable zugreift, greift man tatsächlich auf die entsprechende Stelle im Internet zu.

6. Ein Perl-Programm zum Rechnen mit grossen Ganzzahlen

```

1 # Datei BigNat02.pl
2 # -----
3 # Liest wiederholt zwei im Prinzip beliebig grosse Ganzzahlen ein und gibt
4 # sie gefolgt von ihrer Summe und ihrem Produkt in lesbarer Form aus.
5 # -----

```

```

6 use integer; # 7/2 ist gleich 3 und nicht gleich 3.33333
7 use strict; # Alle Variablen muessen mit vereinbart werden (z.B. mit my)
8 # -----
9 sub machLesbar($;$) {
10 # Erwartet einen String von Dezimalziffern und optional eine Ganzzahl
11 # sollLaenge. Liefert einen Ergebnisstring, der mindestens alle Ziffern
12 # enthaelt (je 3 durch einen Untestrich "_" getrennt) und der, falls
13 # noetig, vorn mit Blanks auf die sollLaenge verlaengert wurde.
14 # Beispiele:
15 # machLesbar("12345", 8) liefert " 12_345"
16 # machLesbar("4321") liefert "4_321"
17
18 my ($ziffern, $sollLaenge, $anzZiffern, $mindLaenge, $sollLaenge);
19 my ($erg, $iErg, $iZiff);
20
21 $ziffern = $_[0]; # Parameter1
22 $sollLaenge = $_[1]; # Parameter2
23 $anzZiffern = length($ziffern); # Anzahl Ziffern
24 $mindLaenge = (4*$anzZiffern-1)/3; # Laenge inklusive Unterstriche
25 $sollLaenge = $sollLaenge > $mindLaenge ? $sollLaenge : $mindLaenge;
26 $erg = " " x $sollLaenge; # Ergebnisstring wird initialisiert
27
28 $iErg = length($erg);
29 foreach $iZiff (reverse(-length($ziffern)..-1)) {
30     if ($iZiff%3 == -1 && $iZiff != -1) {substr($erg, --$iErg, 1) = "_";}
31     substr($erg, --$iErg, 1) = substr($ziffern, $iZiff, 1);
32 }
33 return $erg;
34 } # machLesbar
35 # -----
36 sub add($$) {
37 # Erwartet zwei Strings von Dezimalziffern. Addiert die dadurch darge-
38 # stellten Zahlen und liefert ihre Summe (als String von Dezimalziffern)
39 my ($s1, $s2, $erg, $maxLaenge, $i, $summe, $ziffer, $imSinn);
40
41 $s1 = $_[0]; # Parameter1
42 $s2 = $_[1]; # Parameter2
43
44 $maxLaenge = length($s1) > length($s2) ? length($s1) : length($s2);
45 foreach $i (reverse(-$maxLaenge..-1)) {
46     $summe = substr($s1, $i, 1) + substr($s2, $i, 1) + $imSinn;
47     $ziffer = $summe % 10;
48     $imSinn = $summe / 10;
49     substr($erg, $i, 1) = $ziffer;
50 } # foreach $i
51
52 # Eventuell den letzten Uebertrag vor dem bisherigen Ergebnis einfüegen:
53 if ($imSinn != 0) {$erg = $imSinn.$erg;}
54
55 return $erg;
56 } # sub add
57 # -----
58 sub mult($$) {
59 # Erwartet zwei Strings von Dezimalziffern. Multipliziert die dadurch
60 # dargestellten Zahlen und liefert ihr Produkt (als String von Dezimal-
61 # ziffern).
62 my ($s1, $s2, $erg, $i1, $i2, $ie, $produkt, $ziffer, $imSinn);
63
64 $s1 = $_[0]; # Parameter1
65 $s2 = $_[1]; # Parameter2
66
67 foreach $i1 (reverse(-length($s1)..-1)) {
68     foreach $i2 (reverse(-length($s2)..-1)) {
69         $ie = $i1+$i2+1; # Index fuer den Ergebnis-String $erg
70         $produkt = substr($s1, $i1, 1) * substr($s2, $i2, 1) +
71             substr($erg, $ie, 1) + $imSinn;
72         $ziffer = $produkt % 10;

```

```

73     $imSinn = $produkt / 10;
74     substr($erg, $ie, 1) = $ziffer;
75     } # foreach $i2
76   } #foreach $i1
77
78   # Eventuell den letzten Uebertrag vor dem bisherigen Ergebnis einfüegen:
79   if ($imSinn != 0) {$erg = $imSinn.$erg;}
80   return $erg;
81 } # sub mult
82 # -----
83 # Jetzt werden wiederholt zwei Strings von Dezimalziffern eingelesen,
84 # addiert und multipliziert. Die eingelesenen Zahlen, ihre Summe und ihr
85 # Produkt werden in lesbarer Form ausgegeben. Wenn der Benutzer eine 0
86 # eingibt (oder nur auf Return drueckt) wird das Programm beendet.
87 my ($ein1, $ein2, $laenge);
88 $ein1 = 1;
89 while ($ein1) {
90   print "Eine Ganzzahl (0 zum Beenden)? "; chomp($ein1 = <STDIN>);
91   last if ($ein1 == 0);
92   print "Eine Ganzzahl (0 zum Beenden)? "; chomp($ein2 = <STDIN>);
93   last if ($ein1 == 0);
94   $laenge = 4 * (length($ein1) + length($ein2)) / 3;
95
96   print "Zahl1:   ", machLesbar($ein1,           $laenge), "\n";
97   print "Zahl2:   ", machLesbar($ein2,           $laenge), "\n";
98   print "Summe:   ", machLesbar(add ($ein1, $ein2), $laenge), "\n";
99   print "Produkt: ", machLesbar(mult($ein1, $ein2), $laenge), "\n";
100 } # while
101 # -----

```

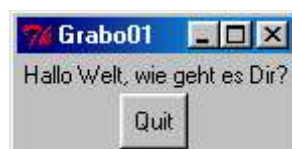
7. Eine grafische Benutzeroberfläche

```

1 # Datei Grabo01.pl
2
3 use strict;                               # Variablen muessen vereinbart werden
4 use Tk;                                     # Den Grabo-Modul Tk einbinden
5
6 my $haupt = MainWindow->new;               # Das Hauptfenster erzeugen
7
8 my $etikett = $haupt->Label(                 # Ein Etikett erzeugen
9   -text => "Hallo Welt, wie geht es Dir?"
10 )->pack;                                     # und sichtbar machen.
11
12 my $knopf = $haupt->Button(                  # Einen Knopf erzeugen,
13   -text => 'Quit',                           # mit Beschriftung versehen,
14   -command => [$haupt => 'destroy']        # mit einem Befehl verbinden
15 )->pack;                                     # und sichtbar machen
16
17 MainLoop;                                  # Auf Ereignisse warten

```

Dieses Programm erzeugt auf dem Bildschirm folgendes Fenster:



Der Benutzer kann das Programm beenden, indem er auf den Quit-Knopf oder auf den Fenster-Schliessen-Knopf (oben rechts, der mit dem "x") klickt.

8. Ein Tischrechner

Hier ein umfangreicheres Perl-Skript, welches eine Grabo (graphische Benutzeroberfläche) erzeugt:

```

1 # Datei Rechner01.pl
2 # -----
3 # Ein kleiner Tischrechner mit Grabo (graphischer Benutzeroberflaeche),
4 # programmiert in Perl/Tk. Damit kann man Ganzzahlen und Bruchzahlen
5 # addieren, multiplizieren, subtrahieren und dividieren.
6 # -----
7 use Tk;
8 use strict; # Ab jetzt muessen Variablen *vereinbart* werden.
9 # -----
10 # Konstanten, die die Groesse der graphischen Elemente festlegen:
11 my $BB = 10; # Breite der button-Elemente
12 my $BL = 15; # Breite der label-Elemente
13 my $BE = 20; # Breite der entry-Elemente
14 my $PY = 3; # Polsterung in y-Richtung
15 # -----
16 # Sechs Variablen vereinbaren und initialisieren:
17 my ($ein1V, $ein2V, $summV, $prodV, $diffV, $quotV);
18 vonVorn(); # Initialisiert die sechs Variablen
19 # -----
20 sub beRechne {
21     # Berechnet aus den zwei Eingaben $ein1V und $ein2V die vier Ausgaben
22     # $summV, $prodV, $diffV und $quotV:
23
24     $summV = $ein1V+$ein2V;
25     $prodV = $ein1V*$ein2V;
26     $diffV = $ein1V-$ein2V;
27
28     # Wenn Dividend gleich 0 dann wird Quotient zu "not a number":
29     $quotV = ($ein2V != 0) ? ($ein1V / $ein2V) : "NaN";
30
31 } # sub beRechne
32 # -----
33 sub vonVorn {
34     # Initialisiert die sechs Variablen $ein1V, $ein2V, $summV, $prodV,
35     # $diffV und $quotV mit einer "Beispiel-Berechnung":
36
37     $ein1V = 9; # Beispiel fuer eine Eingabe
38     $ein2V = 4; # Beispiel fuer eine Eingabe
39     beRechne; # Berechne 9+4, 9*4, 9-4 und 9/4
40 }; # sub vonVorn
41 # -----
42 # Das Hauptfenster erzeugen und sichtbar machen:
43 my $haupt = MainWindow->new(-title => 'Rechner01 (in Perl/Tk)');
44
45 # Zwei Knoepfe (button-Elemente):
46 my $quitB = $haupt->Button(-text => 'Quit',
47     -command => [$haupt => 'destroy'], -width => $BB)->pack;
48 my $vonVornB = $haupt->Button(-text => 'Von vorn',
49     -command => sub{vonVorn()}, -width => $BB)->pack;
50
51 # Sechs Hilfsrahmen (frame-Elemente) zum Zusammenfassen und Anordnen von
52 # je einem Etikett (label-Element) und einer Eingabezeile (entry-Element):
53 my $ein1F = $haupt->Frame->pack(-pady=>$PY); # fuer $ein1L und $ein1E
54 my $ein2F = $haupt->Frame->pack(-pady=>$PY); # fuer $ein2L und $ein2E
55 my $summF = $haupt->Frame->pack(-pady=>$PY); # fuer $summL und $summE
56 my $prodF = $haupt->Frame->pack(-pady=>$PY); # fuer $prodL und $prodE
57 my $diffF = $haupt->Frame->pack(-pady=>$PY); # fuer $diffL und $diffE
58 my $quotF = $haupt->Frame->pack(-pady=>$PY); # fuer $quotL und $quotE
59
60 # Sechs Etiketten (label-Elemente):
61 my $ein1L = $ein1F->Label(-text=>'Eingabe z1:', -width=>$BL)
62     ->pack(-side => 'left');
63 my $ein2L = $ein2F->Label(-text=>'Eingabe z2:', -width=>$BL)

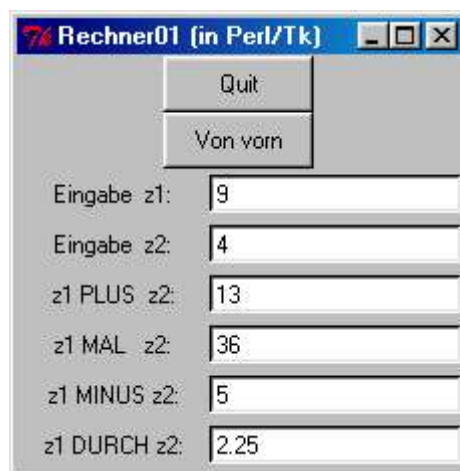
```

```

64   ->pack(-side => 'left');
65 my $summL = $summF->Label(-text=>'z1 PLUS  z2:', -width=>$BL)
66   ->pack(-side => 'left');
67 my $prodL = $prodF->Label(-text=>'z1 MAL   z2:', -width=>$BL)
68   ->pack(-side => 'left');
69 my $diffL = $diffF->Label(-text=>'z1 MINUS z2:', -width=>$BL)
70   ->pack(-side => 'left');
71 my $quotL = $quotF->Label(-text=>'z1 DURCH z2:', -width=>$BL)
72   ->pack(-side => 'left');
73
74 # Sechs Eingabezeilen (entry-Elemente):
75 my $ein1E = $ein1F->Entry(-textvariable => \$ein1V, -width => $BE)->pack;
76 my $ein2E = $ein2F->Entry(-textvariable => \$ein2V, -width => $BE)->pack;
77 my $summE = $summF->Entry(-textvariable => \$summV, -width => $BE)->pack;
78 my $prodE = $prodF->Entry(-textvariable => \$prodV, -width => $BE)->pack;
79 my $diffE = $diffF->Entry(-textvariable => \$diffV, -width => $BE)->pack;
80 my $quoteE = $quotF->Entry(-textvariable => \$quoteV, -width => $BE)->pack;
81
82 # Falls in $ein1E oder $ein2E eine Taste losgelassen wird (weil der Be-
83 # nutzer gerade ein Zeichen eingegeben hat) werden alle Ausgaben neu
84 # berechnet und angezeigt:
85 $ein1E->bind("<KeyRelease>", sub {beRechne});
86 $ein2E->bind("<KeyRelease>", sub {beRechne});
87
88 # Auf Ereignisse warten und die Ereignisse (wenn sie eintreten) behandeln:
89 MainLoop;
90 # -----
91 # Ende der Datei Rechner01.pl

```

Dieses Programm erzeugt auf dem Bildschirm folgendes Fenster:



In diesem **Fenster** sind 2 **Knöpfe** (Button-Elemente) und 6 unsichtbare **Rahmen** (Frame-Elemente) **untereinander** angeordnet. In jedem der 6 Rahmen ist ein **Etikett** (Label-Element) und eine **Eingabezeile** (Entry-Element) **nebeneinander** angeordnet. Z.B. enthält der oberste Rahmen ein Etikett mit der Aufschrift **Eingabe z1:** und eine Eingabezeile, die anfangs den Wert **9** enthält.

Jede **Eingabezeile** ist mit einer **Textvariablen** verbunden (z.B. ist die Eingabezeile **.ein1E** mit der Variablen **ein1V** verbunden). Das Shellprogramm sorgt dafür, dass die Eingabezeile auf dem Bildschirm und die interne Textvariable **immer den gleichen Wert** enthalten. Wenn die Variable verändert wird (z.B. durch einen **set**-Befehl im Skript), wird ihr neuer Wert sofort und automatisch in der Eingabezeile angezeigt und wenn der Benutzer Daten in die Eingabezeile eintippt werden diese Daten (als String) sofort und automatisch in die interne Variable übertragen.

Die obersten beiden Eingabezeilen (neben denen die Etiketten **Eingabe 1:** und **Eingabe 2:** stehen) werden in diesem Skript zum **Einlesen** von zwei Zahlen verwendet. Die übrigen vier Eingabezeilen (mit den Etiketten **z1 PLUS z2:**, **z1 MAL z2:**, **z1 MINUS z2:** und **z1 DURCH z2:**) dienen nur zum **Ausgeben** der entsprechenden Rechenergebnisse.

In den Zeilen 20 bis 31 wird ein Unterprogramm namens **beRechne** vereinbart. Mit dem **bind**-Befehl in Zeile 85 wird der Befehl **beRechne** an das Ereignis **KeyRelease** in der Eingabezeile **\$ein1E** gebunden. Dieser **bind**-Befehl hat folgende Wirkung: Jedesmal, wenn der Benutzer ein Zeichen in die Eingabezeile 1 eintippt, wird das Unterprogramm **beRechne** aufgerufen. Eine ganz entsprechende Wirkung (für die Eingabezeile 2) hat der **bind**-Befehl in Zeile 86.

9. Ein paar Internetadressen zu Perl

1. <http://www.activestate.com/>
2. <http://www.perl.com/>
3. <http://language.perl.com/>
4. <http://search.cpan.org/>
5. <ftp://ftp-stud.fht-esslingen.e/pub/Mirrors/CPAN>

Hier findet man ein gutes, **kostenloses** und leicht zu installierendes **Perl-System** zum runterladen. Zu diesem System gehören zahlreiche **Perl-Module** und eine umfangreiche **Dokumentation** (in HTML).

Eine Seite des Verlags **O'Reilly**, der viele Bücher über Perl verlegt. Auch hier findet man das neuste Perl-System zum runterladen, Antworten auf häufig gestellte Fragen (**FAQs**, frequently asked questions), Artikel zu Perl und Verbindungen zu anderen Perl-Seiten im Internet. Die Seite von O'Reilly verfügt über eine **Suchfunktion** (wenn man reinkommt gleich oben rechts).

Die gleiche Seite wie 2.

CPAN ist das **Comprehensive Perl Archive Network**. Dies ist eine Adresse innerhalb dieses Netzwerkes, in dem man alles über Perl finden (und beliebig lange herumsuchen) kann.

Diese Seite ist ein Teil des **CPAN** und der zugehörige Server steht in Esslingen. Viele Orte in Deutschland liegen näher an Esslingen als an New York oder Californien.

10. Bewertung von Perl

Typischerweise wird man zum Programmieren einer graphischen Benutzeroberfläche in Perl **deutlich weniger** Zeit benötigen, als wenn man die Sprache **Java** benützt und **viel weniger** Zeit, als wenn man **C++** verwendet. Auf der anderen Seite ist ein Perl-Skript bei der Ausführung **deutlich langsamer** als ein entsprechendes Java-Programm und **viel langsamer** als ein entsprechendes C++-Programm. In vielen Fällen merkt der Benutzer allerdings nichts von diesem Geschwindigkeitsunterschied, weil es dabei nur um kleine Bruchteile von Sekunden geht (und der Benutzer selbst z.B. zum Eintippen eines einzigen Zeichens eine Zehntelsekunde oder noch länger braucht).

Perl steht auf allen verbreiteten Plattformen (Unix, Windows, Mac) zur Verfügung. **Perl**-Skripte (insbesondere solche, die graphische Benutzeroberflächen realisieren) sind fast so gut (zwischen verschiedenen Plattformen) **portierbar** wie **Java**-Programme und somit wesentlich **leichter portierbar** als entsprechende **C/C++**-Programme.

Perl hat einen sehr ähnlichen Leistungsumfang wie **Tcl/Tk**. Im Vergleich zu **Tcl/Tk** beruht **Perl** auf **mehr Grundkonzepten** und hat eine **Syntax**, die der von **C/C++** **ähnelt**. Beide Skriptsprachen werden von der **Open-Source-Bewegung** unterstützt.

11. Zum Abschluss

Die Sprache **Perl** wurde ursprünglich von dem Amerikaner **Larry Wall** geschaffen, der auch heute noch sehr **Perl**-aktiv ist. Er geht davon aus, dass Programmieren auch **Spass** machen sollte und bezeichnet Programmierer gern als **Zauberer** (magicans) und Programme als **Zaubersprüche** (magic spells). Bestimmte Programmierprobleme löst man in **Perl**, indem man Variablen mit dem Befehl **bless segnet**, Unterprogramme können sich mit dem **croak**-Befehl über falsche oder fehlende Parameter **krächzend beschweren** oder mit dem **confess**-Befehl **beichten**, dass etwas schief gelaufen ist. **Perl** ist eine sehr ernst zu nehmende Sprache, kann aber gleichzeitig auch grossen **Spass** machen.