

B++ Programmierhandbuch

August 2019

© 2019 Ralf-Erik Ebert, Berlin

Die in diesem Text verwendeten Bezeichnungen von Hard- und Software-Produkten sowie Firmennamen sind in der Regel – auch ohne besondere Kennzeichnung – eingetragene Warenzeichen und sollten als solche behandelt werden.

Der hier veröffentlichte Text wurde mit großer Sorgfalt erarbeitet. Dennoch können Fehler und Irrtümer nicht ausgeschlossen werden. Für entsprechende Hinweise und Verbesserungsvorschläge ist der Autor jederzeit dankbar.

Die im Text verwendeten Beispiele dienen ausschließlich der Illustration und dem leichteren Verständnis des Textes.

Der Autor übernimmt keinerlei Haftung für durch die Verwendung der Software B++, dieser Dokumentation und der darin enthaltenen Code-Beispiele entstehende Schäden.

Änderungen, die der Korrektur, der Weiterentwicklung der Software B++ sowie der Ergänzung ihrer Dokumentation dienen, behält sich der Autor vor.

Inhalt

1 VORWORT.....	7
2 GRUNDLAGEN.....	8
2.1 Was ist B++?.....	8
2.2 Architekturübersicht.....	9
2.3 Die B++ Distribution.....	10
2.4 Verwenden von B++ auf der Kommandozeile.....	11
2.5 Umgebungsvariablen.....	12
3 PROGRAMMSTRUKTUR.....	13
4 SYNTAKTISCHE GRUNDELEMENTE.....	15
4.1 Kommentare.....	15
4.2 Bezeichner.....	15
4.3 Literale.....	15
4.3.1 Zeichenliterale.....	16
4.3.2 Zeichenkettenliterale.....	16
4.3.3 Ganzzahlige Literale.....	17
4.3.4 Gleitkommaliterale.....	18
4.3.5 Erweiterte Literale.....	18
4.3.6 Literale Ausdrücke.....	19
4.3.7 Benannte Literale.....	19
4.3.8 Vordefinierte Literale.....	19
4.4 Definitionen, Anweisungen und Blöcke.....	20
4.5 Verarbeitungsanweisungen.....	23
4.5.1 Die #include-Anweisung.....	23
4.5.2 Die #use-Anweisung.....	24
4.5.3 Die #import-Anweisung.....	24
4.5.4 Die Anweisungen #deflit, #define, #literal, #undeflit und #undef.....	24
4.5.5 Die #defvar-Anweisung.....	25
4.5.6 Die Anweisungen #ifdef, #ifndef, #if, #elif, #else und #endif.....	26
4.5.7 Kennzeichnen literaler Ausdrücke mit #().....	26
4.5.8 Die #endsrc-Anweisung.....	26
5 VARIABLEN UND DATENTYPEN.....	28
5.1 Variablen.....	28
5.2 Datentypen.....	29
5.2.1 Werttypen.....	30
5.2.2 Referenztypen.....	32
5.3 Statisch typisierte Variablen.....	36
5.4 Initialisierung von Variablen.....	37
5.5 Implizite und explizite Typkonvertierungen.....	38
5.6 Vordefinierte Variablen.....	39
6 OPERATOREN.....	41
6.1 Arithmetische Operatoren.....	41
6.2 Vergleichsoperatoren.....	42
6.3 Logische Operatoren.....	43
6.4 Bitoperatoren.....	43
6.5 Zuweisungsoperatoren.....	44
6.6 Inkrement- und Dekrement-Operatoren.....	45
6.7 Sonstige Operatoren.....	46
6.8 Hierarchie der Operatoren.....	49
7 SCHLÜSSELWÖRTER UND RESERVIERTE BEZEICHNER.....	50

8 STEUERSTRUKTUREN.....	53
8.1 Bedingungen (if-else).....	53
8.2 Fallunterscheidungen (switch-case).....	53
8.3 Wiederholungen (Schleifen).....	54
8.3.1 Die while-Anweisung.....	54
8.3.2 Die do-while-Anweisung.....	54
8.3.3 Die for-Anweisung.....	55
8.3.4 Geschachtelte Schleifen.....	56
8.3.5 Die Anweisungen break und continue.....	56
8.4 Fehlerbehandlung (try – catch – throw).....	57
9 FUNKTIONEN.....	58
9.1 Funktionsdefinition.....	58
9.2 Aufruf von Funktionen.....	60
9.3 Variable Parameterlisten und Vorgabeparameter.....	60
9.4 Statische Variablen in Funktionen.....	61
9.5 Funktionszeiger.....	62
9.6 Überladen von Funktionen.....	62
9.7 Anonyme Funktionen und Funktionslitterale.....	63
10 OBJEKTORIENTIERTE PROGRAMMIERUNG.....	65
10.1 Aufbau einer Klasse.....	65
10.2 Inline-Definition von Klassen.....	67
10.3 Konstruktoren und Destruktoren.....	68
10.4 Zugriff auf Member-Funktionen und -Variablen innerhalb einer Klasse.....	71
10.5 Virtuelle Methoden.....	71
10.6 Erweiterte Syntax des Operators ->.....	73
10.7 Methodenzeiger und Methodenlitterale.....	74
10.8 Partielle Klassen.....	75
10.9 Ersetzen von Member-Funktionen (Redefinition).....	77
10.10 Definieren von Operatoren.....	77
10.10.1 Definieren des Funktionsoperators.....	78
10.10.2 Definieren des Indexoperators.....	79
10.10.3 Definieren des Punktoperators.....	80
10.10.4 Definieren der Inkrement- und Dekrement-Operatoren.....	81
10.10.5 Definieren des Dereferenzierungsoperators.....	83
10.10.6 Definieren der weiteren unären Operatoren.....	83
10.10.7 Indirekte Definition von Vergleichsoperatoren mit den Methoden compare bzw. equals.....	83
10.10.8 Binäre Infix-Operatoren.....	84
10.10.9 Operatoren zur Typumwandlung.....	86
11 VORDEFINIERT FUNKTIONEN.....	88
11.1 Standardfunktionen für Container-Datentypen.....	88
11.1.1 Container-Konstruktorfunktionen.....	88
11.1.2 Allgemeine Containerfunktionen.....	92
11.1.3 Vektor-Funktionen.....	95
11.1.4 Dictionary-Funktionen.....	99
11.2 Typkonvertierungen und RTTI.....	101
11.2.1 Funktionen zur Typkonvertierung.....	101
11.2.2 RTTI-Funktionen.....	106
11.3 Ein-/Ausgabe.....	111
11.4 Zeichenkettenfunktionen.....	119
11.5 Datum und Uhrzeit.....	125
11.6 Mathematische Funktionen.....	127
11.7 Systemfunktionen.....	130
11.7.1 MS-DOS-Funktionen.....	138
11.7.2 Windows-Funktionen.....	142

11.8 MS-DOS-Grafikfunktionen.....	144
11.9 Sonstige Funktionen.....	146
12 REGULÄRE AUSDRÜCKE.....	154
12.1 Pattern-Syntax.....	154
12.2 Die vordefinierte Klasse RegEx.....	157
12.2.1 Methoden.....	158
12.2.2 Operatoren.....	166
13 SPEICHERVERWALTUNG.....	169
13.1 Weak References und die unären Operatoren & bzw. *.....	170
13.2 Explizite Speicherverwaltung.....	174
14 INTERNE DATENSTRUKTUREN.....	176
14.1 Globale Symboltabellen und Typen von Symbolen.....	176
14.2 Struktur von Klassen und Objekten.....	176
14.3 Struktur von Funktionen und Aufbau des Bytecodes der VM.....	177
14.4 Organisation des Stacks der VM.....	183
15 ERWEITERN VON B++.....	185
15.1 B++-Bibliotheksmodule.....	185
15.2 Erweitern von B++ mit dynamischen Bibliotheken.....	187
15.2.1 Allgemeine Funktionsbibliotheken.....	187
15.2.2 B++-spezifische Bibliotheken.....	187
15.2.3 Dynamische Bibliotheken für MS-DOS.....	195
15.3 Verwenden dynamischer Bibliotheken.....	203
15.3.1 Allgemeine Funktionsbibliotheken.....	203
15.3.2 B++-spezifische Bibliotheken.....	207
16 EINBETTEN VON B++.....	208
17 QUELLEN.....	209

1 Vorwort

B++ hat seinen Ursprung in BOB [1], einem Ansatz für eine minimalistische objektorientierte (und C++-ähnliche) Sprache, der von DAVID MICHAEL BETZ entwickelt und 1991 im Dr. Dobbs Journal [2] veröffentlicht wurde.

Um die Sprache für den praktischen Einsatz tauglich zu machen, waren einige Ergänzungen des Sprachumfangs selbst, insbesondere eine Speicherverwaltung, sowie die Bereitstellung einer Reihe elementarer vordefinierter Funktionen unumgänglich. Das Resultat erhielt den Namen BOB+ und stellte im Wesentlichen eine Erweiterung von BOB dar, die in weiten Teilen auf dem Originalcode nach [1] aufbaute. Im Fokus der Entwicklung von BOB+ standen zunächst minimal ausgestattete MS-DOS-basierte Systeme. Deshalb wurde besonderer Wert auf minimalen Platzbedarf - sowohl auf dem Datenträger als auch im Arbeitsspeicher - gelegt. Für die Verwendung auf solchen Minimalsystemen wird BOB+ noch weiter gepflegt¹.

B++ ist vor allem als leicht integrierbare Skriptsprache zur benutzerdefinierten Erweiterung/Steuerung von Applikationen, zum schnellen Erstellen kleinerer Hilfsprogramme sowie zum Prototyping gedacht.

B++ ist nicht einfach eine Weiterentwicklung von BOB+, sondern geht unter Beibehaltung der grundlegenden Eigenschaften der Sprache selbst und weitgehender Quellkompatibilität zu BOB+ von anderen Entwurfszielen aus:

- Tauglichkeit zur Einbettung in Programme als Skriptsprache
- Übertragbarkeit auf verschiedene Betriebssysteme
- einfache Erweiterbarkeit durch „native“ Module²
- qualifizierte Fehlerbehandlung für stabilen Code
- einfache Möglichkeiten zur Verwendung systemspezifischer Funktionen
- weitgehend automatische Speicherverwaltung
- Möglichkeit der Integration in Entwicklungswerkzeuge

Die Realisierung dieser Ziele machte eine völlig neue interne Architektur erforderlich, die auch eine vollständige Neuimplementierung nach sich zog.

Das hier vorliegende Handbuch beinhaltet einen Überblick über B++, allgemeine Hinweise zum Einsatz, die Beschreibung der Sprache aus Anwendersicht, eine Referenz der vordefinierten Funktionen sowie Vorgehensweisen zur Erweiterung von B++ und zur Einbettung als Skriptsprache in andere Applikationen.

Das Handbuch ist keine allgemeine Einführung in die Programmierung.

Vielmehr wird versucht, B++ in übersichtlicher und knapper Form darzustellen. Dabei müssen grundlegende Programmierkenntnisse in einer anderen – vorzugsweise C-ähnlichen – Sprache sowie Grundkenntnisse der objektorientierten Programmierung vorausgesetzt werden.

¹ Eine Online-Dokumentation mit einigen Zusätzen für den Atari-Portfolio findet man im POFO-Wiki (www.pofowiki.de). In dem Zusammenhang geht ein besonderer Dank an Volker Hamann, der sich der Mühe unterzogen hat, das Handbuch von BOB+ entsprechend aufzubereiten und dort verfügbar zu machen.

² Gemeint sind hier in einer anderen Sprache (z. B. C/C++ oder Pascal) implementierte Funktionen und Objektklassen für B++.

2 Grundlagen

2.1 Was ist B++?

B++ ist eine Hybridsprache in mehrfacher Hinsicht.

Ihrem Konzept nach ist es zunächst eine prozedurale Sprache mit objektorientierten Erweiterungen. Funktoren und anonyme Funktionen, die ebenfalls Bestandteil von B++ sind, erlauben darüber hinaus auch die Verwendung funktionaler Paradigmen.

B++ ist weder ein „echter“ Compiler noch ein „echter“ Interpreter. Der Quellcode eines Programms wird zunächst in einen Bytecode übersetzt, der anschließend in einer Virtuellen Maschine interpretiert wird. Dies erlaubt eine sehr einfache Portierung von B++ auf unterschiedliche Zielplattformen und die Ausführung vorhandener B++-Programme auf jeder unterstützten Plattform³.

Wie für Skriptsprachen üblich, erlaubt es B++ mit dynamisch typisierten Variablen zu arbeiten. Daneben steht auch eine statische Typisierung (siehe Abschnitt 5.3) zur Verfügung, wobei die Typprüfung teilweise durch den Compiler und teilweise durch das Laufzeitsystem erfolgt. Beide Formen können auch gemischt verwendet werden.

Syntaktisch (und bis zu einem gewissen Grad auch semantisch) lehnt sich B++ stark an C/C++ an, mit einigen Elementen aus anderen Skriptsprachen, insbesondere JavaScript. Der Name „B++“ weist auf die Anlehnung an C++ und auf den „Vorfahr“ BOB+ hin.

B++ ist eine vom Kern her überschaubare Sprache mit nur 30 Schlüsselwörtern (vgl. Kapitel 7), die in ihren Grundzügen schnell und einfach erlernbar ist. Gleichzeitig ist es aber auch eine sehr mächtige und ausdrucksstarke Sprache, die Objektorientierung und Modularisierung ebenso unterstützt wie Zugriffe auf Systemfunktionen.

Die nachfolgende (unvollständige) Aufzählung beinhaltet einige wesentliche Sprachmerkmale von B++, die weiter unten noch ausführlicher dargestellt werden.

- Unterstützung von Gleitpunkt-Arithmetik (Datentypen `float` und `double`) auf allen Plattformen, wenn nötig per Emulation⁴
- explizit verwendbare Integer-Datentypen von 8 bis 64 Bit Breite, jeweils mit und ohne Vorzeichen, wobei 32 Bit die Standardgröße ist⁵
- integrierte Datentypen für Zeichenketten, Puffer, Vektoren (dynamische Arrays) und Dictionaries (assoziative Container)
- ganzzahlige numerische Literale in dezimaler, binärer, oktaler oder hexadezimaler Schreibweise
- LiteralDarstellungen für Vektoren und Dictionaries
- Funktions- und Methodenzeiger
- anonyme Funktionen und Methoden
- variable Parameterlisten und Vorgabeparameter für Funktionen
- polymorphe Klassen, automatische Destruktoren und initialisierte statische Member-Variablen
- partielle Klassen
- benutzerdefinierte Operatoren für Objekte
- Typinformationen zur Laufzeit (RTTI)
- Zugriff auf Systemfunktionen
- integrierte Serialisierung beliebiger Daten

³ Letzteres gilt zumindest, so lange keine plattformspezifischen API-Funktionen direkt aufgerufen werden.

⁴ Dies trifft für DOS-Plattformen zu, sofern kein numerischer Koprozessor verfügbar ist.

⁵ Dies gilt für alle Plattformen – wenn nötig wird die 64-Bit-Arithmetik emuliert.

- Verwendung vorkompilierter Bytecode-Bibliotheken
- dynamisches Laden von Bibliotheken zur Laufzeit
- wahlweise explizite Speicherverwaltung oder automatische Speicherverwaltung mit Referenzzählung
- optional statische Typisierung

Hinzu kommt eine Vielzahl vordefinierter Funktionen (vgl. Kapitel 11), welche die Aufgaben einer Standardbibliothek erfüllen.

2.2 Architekturübersicht

Den Kern von B++ bildet die Klassenbibliothek `bpplib`, deren Grobstruktur in Abbildung 1 dargestellt ist. Anwendungsprogramme - zu denen auch die Kommandozeilenversion von B++ selbst gehört - und Erweiterungen benutzen diese Bibliothek über deren C- bzw. C++-Schnittstelle.

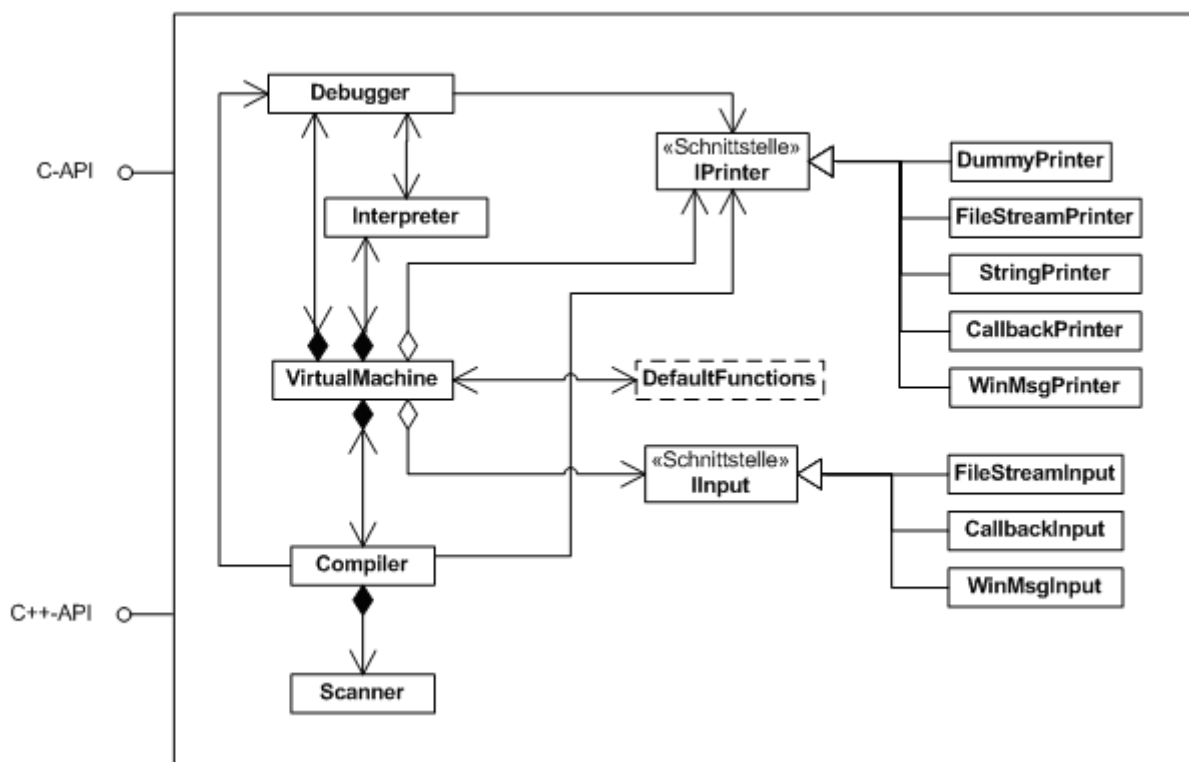


Abbildung 1 Grobstruktur von B++

Von zentraler Bedeutung innerhalb der Bibliothek ist die Virtuelle Maschine (Klasse `VirtualMachine`), die alle zentralen Datenstrukturen verwaltet und die Steuerung aller wesentlichen Abläufe übernimmt.

Der Compiler arbeitet im Single-Pass-Verfahren nach der Methode des rekursiven Abstiegs mit direkter Codeerzeugung⁶. Der Bytecode-Interpreter ist als Kellerautomat realisiert.

Für die Ein- und Ausgabe von Textdaten wird mit Hilfe der Schnittstellen `IInput` und `IPrinter` von den Gegebenheiten der konkreten Anwendung abstrahiert. Einige Implementierungen dieser Schnittstellen bieten häufig benötigte Standardvarianten an und sollen so insbesondere die Integration von B++ als Skriptsprache in Anwendungen erleichtern.

Der B++-Kern verwendet keine globalen Datenstrukturen. Dadurch ist es problemlos möglich, bei Bedarf auch mehrere virtuelle Maschinen gleichzeitig zu betreiben.

⁶ Dies bedeutet einerseits sehr kurze Übersetzungszeiten, was für Skriptsprachen ein wichtiger Aspekt ist, andererseits aber den Verzicht auf automatische Optimierungen.

2.3 Die B++ Distribution

B++ ist derzeit für folgende Plattformen verfügbar:

- MS-DOS ab Version 3.0⁷
- Windows-Desktop 32 Bit (95, 98, ME, NT, 2000, XP, Vista, Win7/8/10)
- Windows-Desktop 64 Bit (Vista, Win 7/8/10)
- Linux (nur als Sourcecode)

Zur vollständigen Distribution gehören

- ein Installer (Setup) für die Windows-Desktop-Versionen
- ein ZIP-Archiv mit den Binärdateien für die DOS-Version sowie einigen Beispielen
- ein ZIP-Archiv mit den gesamten B++-Quellen

Dabei beinhalten die B++-Quellen zusätzlich Projektdateien für Borland C++ (DOS), Visual Studio 2005 und 2015 (Windows Desktop), CodeLite (Linux) sowie Embedded Visual Tools 3.0 (Windows CE)⁸.

Den Windows-Versionen liegt außerdem eine einfache IDE bei, deren Aufgabe es ist, neben der Erleichterung des Erstellens von B++-Programmen zu zeigen, wie sich B++ in andere Applikationen einbetten lässt.

Kern aller Distributionen ist die Bibliothek `bpplib`. Sie beinhaltet die Virtuelle Maschine mit allen erforderlichen weiteren Komponenten (Compiler, Interpreter, vordefinierte Funktionen usw.). Sie liegt in den Windows-Versionen als dynamische Bibliothek (DLL) und in der DOS-Version als statische C++-Bibliothek (LIB) vor. Für Linux kann sie wahlweise als dynamische oder statische Bibliothek erzeugt werden.

Daneben gibt es für alle Plattformen eine Kommandozeilenversion `bp2.exe`⁹, die im Grunde nicht mehr ist als eine Minimalanwendung der Bibliothek mit einer Kommandozeilen-Schnittstelle. Sie erlaubt die direkte Verwendung von B++ von einer Konsole aus bzw. die Einbindung von B++-Programmen in Kommando-Scripts (Batch-Dateien). Von der Kommandozeilenversion wiederum existiert eine modifizierte Variante `bp2r.exe`, die sich mit einem Bytecode-Modul von B++ zu einem eigenständig ausführbaren Programm binden lässt. In den Windows-Desktop-Versionen gibt es eine zusätzliche Variante `bp2rw.exe`, die sich von `bp2r.exe` dadurch unterscheidet, dass sie keine Konsole enthält. Sie ist zum Erstellen 'typischer' Windows-Anwendungen gedacht. Um die verschiedenen Windows-Versionen parallel verwenden zu können, werden den Namen der ausführbaren Dateien spezifische Suffixe angehängt, wobei `u` für eine UNICODE-Variante und `_64` für eine 64-Bit-Variante steht.

Für die DOS-Version existiert eine Variante mit dem Suffix `_386`, die mindestens einen 80386-Prozessor voraussetzt. Sie kann dessen 32-Bit-Registersatz nutzen und ist somit schneller als die Standardvariante, die dafür auch auf einen 8088-Prozessor läuft. Beide Varianten nutzen einen vorhandenen numerischen Koprozessor oder eine Emulation (falls kein solcher Prozessor vorhanden ist) für Gleitkommaoperationen.

Zu den Windows-Versionen gehört außerdem eine Erweiterungsbibliothek `bp2class`, die vor allem zur Demonstration der Implementierung von Erweiterungen gedacht ist. Sie beinhaltet einige allgemeine Klassen (String, Container, Emulation von Strukturen) sowie Basisklassen zur Windows-GUI-Programmierung. Darauf aufbauend wird der Kern eines Windows-GUI-Frameworks als B++-Quelltext bereitgestellt. Zur Demonstration des Umgangs damit dient ein Beispiel, das einen Notepad-Clone in B++ implementiert.

⁷ Dies schließt auch DIP-DOS des Atari Portfolio mit ein, welches zwar die Versionsnummer 2.11 meldet, jedoch in weiten Teilen DOS 3.x entspricht.

⁸ Im Laufe der Entwicklung entstand zunächst eine Zwischenversion mit dem Namen BOB+2, die auch Windows CE/Mobile unterstützte. Da diese Plattform inzwischen weitgehend ausgestorben ist, wird sie nicht mehr aktiv unterstützt, es existieren aber noch Projektdateien, die dies prinzipiell ermöglichen.

⁹ In der Linux-Version fehlt die Dateierweiterung `.exe`

Die Referenzdokumentation von B++ liegt im HTML-Help-Format (`bpp.chm`) vor.

2.4 Verwenden von B++ auf der Kommandozeile

Von der Kommandozeile wird B++ über den Aufruf von `bp2.exe` (bzw. `bp2u.exe`, `bp2_64.exe` oder `bp2u_64.exe` als Varianten) gestartet. Um Aufrufe von beliebiger Stelle zu vereinfachen, sollte der Pfad zu dieser Datei (bzw. diesen Dateien) in die PATH-Umgebungsvariable aufgenommen werden. Für den eigentlichen Aufruf gilt dann folgende Syntax:

```
bp2 [-i][-d][-l][-v][-t][-c][-e][-E][-L rtname] [-Dsymbol[=value][...]]  
    [sourcefile [...]] [-r objfile [...]] [-o outfile] [# userarg [...]]
```

Dabei haben die einzelnen Optionen folgende Bedeutung:

- `-i` (info) Anzeige von Copyright und Funktionsnamen
- `-d` (debug) Ausgabe des erzeugten Bytecodes beim Übersetzen
- `-l` (line info) Namen von Quelldateien und Zeilennummern werden zu Debugging-Zwecken in den Bytecode mit aufgenommen.
- `-v` (variables) Namen lokaler Variablen werden zu Debugging-Zwecken in den Bytecode mit aufgenommen.
- `-t` (trace) Ablaufverfolgung des Bytecodes bei der Ausführung
- `-c` (compile) Die angegebenen Quellen werden nicht ausgeführt, sondern nur in Bytecode übersetzt.
- `-e` (make exe) Erzeugen eines eigenständig lauffähigen Programms (angegebene Quellen werden übersetzt und das Resultat mit dem Standard-Runtime-Modul gebunden)
- `-E` (make exe) Erzeugen eines eigenständig lauffähigen Programms ohne Konsole (angegebene Quellen werden übersetzt und das Resultat mit dem Standard-`--Runtime-Modul` ohne Konsole gebunden). Die Option existiert nur in den Windows-Desktop-Versionen.
- `-L` (link exe) Erzeugen eines eigenständig lauffähigen Programms (angegebene Quellen werden übersetzt und das Resultat mit dem in `rtname` angegebenen Runtime-Modul¹⁰ gebunden).
- `-D` (define symbol) Für die nachfolgende Übersetzung wird ein Symbol `symbol`, optional mit dem Wert `value` definiert. Das so definierte Symbol ist ein benanntes Literal (4.3.7) des Typs `string`, falls ein Wert angegeben wurde, sonst hat es den Wert `null`.
- `-r` (read / run) Lesen und ggf. Ausführen von vorkompilierten Bytecode-Modulen
- `-o` (output) Name der auszugebenden Datei. Die Option wird nur ausgewertet, wenn auch `-c`, `-e`, `-E` oder `-L` gesetzt ist. Fehlt die Angabe, so wird als Standardwert „out.bpm“ bzw. „out.exe“¹¹ verwendet.

Quelldateien und einzulesende Module können ohne Dateiendungen angegeben werden. In solchen Fällen werden implizit die Erweiterungen „.bpm“ bzw. „.bpm“ angenommen.

Bei Angabe mehrerer Quelldateien werden diese in der angegebenen Reihenfolge in Bytecode übersetzt, bevor die eigentliche Programmausführung beginnt. Die zu ladenden vorkompilierten Module werden *vor* dem Kompilieren der Quellen (und ebenfalls in der angegebenen Reihenfolge) gelesen.

¹⁰ Dies ist eine Verallgemeinerung der Optionen `-e` bzw. `-E`, die die explizite Angabe des Runtime-Moduls erlaubt. Somit können ausführbare Programme für andere Zielplattformen erzeugt werden. Dabei ist `rtname` der vollständige Name des Runtime-Moduls, dem optional ein relativer oder absoluter Pfad vorangestellt sein kann.

¹¹Unter Unix/Linux wird kein „.exe“ angehängt, der Standard-Ausgabename ist also einfach „out“.

B++ erlaubt zusätzlich die Angabe einer Liste von Benutzer-Argumenten, die an das auszuführende Programm weitergegeben werden. Diese Liste wird durch ein Hash-Symbol (#) eingeleitet und bildet das Ende der Kommandozeile. Innerhalb eines B++-Programms kann mit den vordefinierten Funktionen `getargs()` bzw. `getusrargs()` auf die gesamte bzw. die Benutzer-Argumentliste zugegriffen werden (vgl. Abschnitt 11.9).

Die vollständige Argumentliste wird beim Aufruf von der Kommandozeile auch der Eintrittsfunktion `main` übergeben, so dass mit Hilfe der Funktionen `argcnt()`, `arg(i)` bzw. `argv()` ebenfalls ein Zugriff auf die Argumente möglich ist.

Im Gegensatz zu BOB+ verwendet B++ einen dynamischen Stack, dessen Größe nur durch den verfügbaren Speicherplatz begrenzt ist. Eine explizite Festlegung der Größe¹² mit Hilfe einer Umgebungsvariablen ist deshalb weder erforderlich noch möglich.

2.5 Umgebungsvariablen

B++ kann zur Laufzeit Erweiterungen in Form von dynamischen Bibliotheken (üblicherweise mit der Erweiterung `.dll`) laden. Die Suche nach solchen Bibliotheken erfolgt über den in der Umgebungsvariablen `PATH` angegebenen Suchpfad¹³.

Für die Suche nach Quelldateien, die über die Verarbeitungsanweisung `#include` (vgl. Abschnitt 4.5.1) eingebunden werden, wird der in der Umgebungsvariablen `BPPINC` angegebene Suchpfad verwendet.

Die Umgebungsvariable `BPPLIB` dient als Suchpfad für zu ladende vorkompilierte Module.

Wie in `PATH` können auch in `BPPINC` und `BPPLIB` mehrere Pfade, jeweils durch ein Semikolon (;) getrennt, hinterlegt werden.

¹² Diese Möglichkeit gibt es für BOB+.

¹³ Abhängig vom jeweiligen Betriebssystem und dessen Regeln zum Laden dynamischer Bibliotheken, erfolgt zusätzlich auch eine Suche in anderen Verzeichnissen, z. B. im aktuellen Verzeichnis, im Verzeichnis des Hauptprogramms oder in Standard-Systemverzeichnissen

3 Programmstruktur

Jedes B++-Programm besteht – wie in C – aus einer Menge von Funktionen (siehe Kapitel 9). Einsprungpunkt ist normalerweise¹⁴ die Funktion `main()`. Neben reinen Funktionen können auf der obersten Strukturebene zusätzlich Verarbeitungsanweisungen (siehe Abschnitt 4.5), Variablendefinitionen (vgl. Kapitel 5) und Klassendeklarationen (vgl. Kapitel 10) stehen. Anders als in C können mit Hilfe der Verarbeitungsanweisung `#defvar` globale Variablen innerhalb der Implementierung von Funktionen angelegt werden. Wie in C und C++ ist es *nicht* möglich, Funktionen *lokal*, d. h. innerhalb anderer Funktionen zu definieren.

Grundsätzlich muss *jeder Bezeichner* vor der ersten Bezugnahme darauf *deklariert* werden.

Das nun folgende unvermeidliche „Hello World“-Beispiel, das absichtlich etwas umständlicher als nötig ausgeführt ist, zeigt den typischen Programmaufbau:

```
1 /* B++ Hello World example */
2
3 // write a string to console
4 void printString(var str)
5 {
6     var n = strlen(str);
7     for (var i=0; i<n; ++i)
8         putc(str[i], stdout);
9 }
10
11 var main()
12 {
13     printString("Hello World - this is B++\n");
14     return 0;
15 }
```

Beispiel 1 Hello-World-Programm in B++

Wie man sieht, besteht eine große syntaktische Ähnlichkeit zu C/C++.

Zunächst ist zu sagen, dass B++ formatfrei ist. Das bedeutet, dass Einrückungen, Leerzeilen, Zeilenumbrüche usw. keine syntaktische Bedeutung haben. Sie dienen lediglich der besseren Lesbarkeit des Quelltextes. Die Länge einer einzelnen Quelltextzeile ist nicht begrenzt, der leichten Lesbarkeit des Textes wegen empfiehlt es sich allerdings, keine unnötig langen Zeilen zu verwenden.

Zeilen 1 und 3 in obigem Beispiel enthalten Kommentare. Danach wird eine Funktion `printString` definiert (siehe Abschnitt 9), die eine Zeichenkette im Parameter `str` übernimmt und deren Inhalt zeichenweise auf die Konsole ausgibt. Hierbei verwendet sie die vordefinierten Funktionen¹⁵ `strlen` (Länge der Zeichenkette bestimmen) und `putc` (Dateiausgabe eines Zeichens) sowie die ebenfalls vordefinierte Variable `stdout`, die einen Verweis auf die Standardausgabe – üblicherweise die Konsole – darstellt.

In den Zeilen 6 und 7 werden mit Hilfe des Schlüsselworts `var` die *lokal* (d. h. innerhalb der Funktion) gültigen Variablen `i` und `n` definiert.

Das Schlüsselwort `var` wird im Beispiel noch in zwei anderen Zusammenhängen benutzt: In Zeile 11 kennzeichnet es, dass die Funktion `main` einen Wert zurückgibt. In Zeile 4 steht es vor dem Parameter `str` im Funktionskopf als Ersatz für einen Typbezeichner, wie er z. B. in C geschrieben werden müsste. Vor dem Funktionskopf in Zeile 4 steht außerdem das Schlüsselwort `void`, das hier kennzeichnet, dass die Funktion `printString` *keinen* Wert zurückgibt.

In diesen Zusammenhängen sind die Schlüsselwörter `var` und `void` nicht zwingend erforderlich

¹⁴ Dies gilt zumindest für die Kommandozeilenversion. Eine Anwendung, die die virtuelle Maschine von B++ integriert, kann prinzipiell jede beliebige Funktion als Einsprungpunkt aufrufen.

¹⁵ siehe hierzu Kapitel 11

und könnten demnach auch weggelassen werden. Ihr Zweck besteht u. a. auch darin, für B++-Quellen automatische Dokumentationsgeneratoren, wie z. B. Doxygen [5], einsetzen zu können, die eigentlich für die Syntax von C bzw. C++ ausgelegt sind. Außerdem können sie die Verständlichkeit des Quelltextes verbessern. Bei der Kennzeichnung einer Funktion mit `void` verhindert der Compiler zudem die Rückgabe eines Wertes, bei Angabe von `var` muss dagegen mit `return` tatsächlich ein Wert zurückgegeben werden.

Die Einsprungfunktion `main` ruft nun `printString` mit der auszugebenden Zeichenkette als Argument auf.

Hier ist auf die besondere Bedeutung des Rückgabewertes der `main`-Funktion hinzuweisen. Da es innerhalb des Programms keinen Aufrufer dieser Funktion gibt, wird der Wert an das Betriebssystem selbst (unter DOS und in der Windows-Konsole als Errorlevel) zurückgegeben¹⁶. Daraus ergibt sich die Einschränkung, dass der Rückgabewert nur eine Ganzzahl sein kann. Aus diesem Grund prüft B++ nach Beendigung der `main`-Funktion zunächst den Typ des Rückgabewertes¹⁷. Handelt es sich um einen Integer-Wert, so wird dieser, andernfalls 0 zurückgegeben.

Noch C-ähnlicher sieht das „Hello World“-Beispiel aus, wenn man die optionale statische Typisierung verwendet:

```
/* B++ Hello World example */

// write a string to console
void printString(string str)
{
    int n = strlen(str);
    for (int i=0; i<n; ++i)
        putchar(str[i], stdout);
}

int main()
{
    printString("Hello World - this is B++\n");
    return 0;
}
```

Beispiel 2 Hello-World-Programm mit statischer Typisierung

Hier wird statt des Schlüsselwortes `var` ein Typbezeichner verwendet (im Beispiel `string` und `int`). Neben der verbesserten Lesbarkeit lässt sich damit ein großer Teil ggf. nötiger Gültigkeitsprüfungen vom Quelltext – und damit der Verantwortlichkeit des Programmierers – an den Compiler bzw. die Laufzeitumgebung delegieren, was im besten Fall zu kürzerem Quellcode und besseren (stabileren) Programmen führt – um den Preis eines etwas ungünstigeren Laufzeitverhaltens – siehe dazu Abschnitt 5.3.

¹⁶ Das gilt nur für die Kommandozeilenversion. Eine Anwendung, die B++ integriert, kann mit dem Wert natürlich machen, was sie will.

¹⁷ Genau genommen wird nicht der Rückgabewert, sondern der letzte auf dem internen Stack liegende Wert verwendet. Wird `main` nicht mit einer `return`-Anweisung verlassen, so ist der an das Betriebssystem übergebene Wert mehr oder weniger zufällig.

4 Syntaktische Grundelemente

In diesem Kapitel werden die syntaktischen Grundelemente erklärt, die für das Schreiben von B++-Programmen von Bedeutung sind. Dabei handelt es sich um keine vollständige formale Syntaxbeschreibung. Vielmehr sollen die wichtigsten Sprachelemente im Vergleich zu C/C++ dargestellt werden.

4.1 Kommentare

B++ kennt drei Arten von Kommentaren:

- **Kommentare in C-Stil:**
Solche Kommentare haben die Form `/* Kommentartext */`.
Sie können sich über mehrere Zeilen erstrecken, dürfen jedoch nicht verschachtelt werden.
- **Kommentare im C++-Stil:**
Diese Kommentare werden in der Form `// Kommentartext` notiert. Sie reichen jeweils bis zum Ende der aktuellen Quelltextzeile.
- **Mit @ bzw. #! eingeleitete Kommentare:**
Diese Form des Kommentars dient der Einbettung von B++-Quellen in Shell-Scripts (Batch-Dateien unter DOS). Für B++ selbst ist sie identisch mit den C++-Kommentaren. Da das Zeichen @ in Batch-Scripts (bzw. die Kombination #! in Unix-Shell-Scripts) aber eine gültige Kommandozeile einleitet, deren Ausgabe lediglich unterdrückt wird, lässt sich diese Art des Kommentars verwenden, um B++-Code in solche Scripts einzubinden. Ein Beispiel hierzu findet sich in Abschnitt 4.5.8.

4.2 Bezeichner

Für Bezeichner gelten im Wesentlichen die gleichen Regeln wie in C bzw. C++.

Bezeichner bestehen aus einer Folge von Buchstaben (ohne Umlaute und ß) und Ziffern bzw. dem Unterstrich (`_`) sowie dem Dollar-Symbol¹⁸ (`$`). Sie dürfen nicht mit einer Ziffer beginnen, die Groß- bzw. Kleinschreibung ist signifikant.

In B++ gibt es weder eine Begrenzung der Länge von Bezeichnern noch der Anzahl der signifikanten Zeichen.

Beispiele für Bezeichner:

```
MyClass  
myValue  
_anInternalVariable  
p1  
$A
```

Generell sollten Bezeichner so gewählt werden, dass sie das spätere Lesen eines Quelltextes erleichtern. Das bedeutet insbesondere, dass Bezeichner möglichst einen direkten Schluss auf ihren Verwendungszweck zulassen sollten.

4.3 Literale

Literale im engeren Sinne sind direkt im Quellcode notierte konstante Werte. Hierzu gehören zunächst die Notationen von Werten der verschiedenen elementaren Datentypen (vgl. Abschnitt 5.2). Hinzu kommen in B++ erweiterte Formen zur Initialisierung von Vektoren und Dictionaries (siehe 5.2.2) sowie Funktionslitterale (4.3.5) und literale Ausdrücke (4.3.6).

¹⁸ Das Dollar-Symbol ist auch in JavaScript-Bezeichnern gültig, zum C++-Standard gehört es nicht, wird aber von den meisten Compilern akzeptiert. Deshalb wurde es auch in B++-Bezeichner mit aufgenommen.

4.3.1 Zeichenliterale

Ein Zeichenliteral ist ein einzelnes Zeichen mit einem Zeichencode zwischen 0 und 255¹⁹, das in Hochkommata geklammert ist, z. B. `'A', 'x', 'ä', '7', '\n'`.

Das letzte Zeichen in obiger Aufzählung weist dabei eine Besonderheit auf. Es repräsentiert kein druckbares Zeichen, sondern ein Steuerzeichen. Die Darstellung spezieller Zeichen wird stets mit einem Backslash (`\`), oft auch *Escape*-Zeichen genannt, eingeleitet. Tabelle 1 listet die von B++ interpretierten Escape-Codes mit ihrer Bedeutung auf.

Escape-Code	Bedeutung	Zeichencode (Hex)
<code>\0</code>	NUL-Byte	00
<code>\a</code>	Alarmton	07
<code>\b</code>	Backspace	08
<code>\f</code>	Seitenvorschub	0C
<code>\n</code>	Zeilenvorschub	10
<code>\r</code>	Wagenrücklauf	0D
<code>\t</code>	Tabulator horizontal	09
<code>\v</code>	Tabulator vertikal	0B
<code>\\</code>	Backslash	5C
<code>\'</code>	Apostroph	27
<code>\"</code>	Anführungszeichen	22

Tabelle 1 Von B++ unterstützte Escape-Codes

Wie C/C++ (und nach denselben Regeln) erlaubt auch B++ die Verwendung von *Escape-Sequenzen*, um den Code nicht druckbarer Zeichen als Zeichenliterale zu notieren. Dabei kann der Code hexadezimal oder oktal angegeben werden. Ein Konstrukt wie `\xFF` oder `\377` entspricht also dem Zeichen mit dem Code 255.

Zusätzlich ist es möglich, in einer Escape-Sequenz den Wert binär anzugeben. In diesem Fall wird sie mit `\0b` eingeleitet. Das Zeichen mit dem Code 255 könnte also auch als `\0b11111111` angegeben werden.

Darüber hinaus werden analog zu C++11 auch die Escape-Sequenzen `\uXXXX` (UTF-8-Zeichen) bzw. `\UXXXX` (UTF-16-Zeichen) interpretiert, wobei `XXXX` für eine Folge von Hexadezimalziffern steht. Da B++ aber keine Multibyte-Zeichen kennt, werden die beiden Sequenzen wie die `\xxxxx`-Sequenz behandelt, also einfach als hexadezimaler Zahlenwert, der als Zeichencode interpretiert wird. In Unicode-Builds entspricht das dem Code eines 2-Byte-Zeichens (UTF-16).

Intern werden Zeichenliterale wie ganze Zahlen (Typ `int`) behandelt, wobei der Zeichencode dem Zahlenwert entspricht. Dementsprechend können sie im Quelltext überall dort verwendet werden, wo auch eine ganze Zahl stehen kann.

4.3.2 Zeichenkettenliterale

Zeichenkettenliterale sind Folgen beliebiger druckbarer Zeichen oder Escape-Codes (siehe Tabelle 1) bzw. Escape-Sequenzen, die in Anführungszeichen eingeschlossen sind. Kommt das Anführungszeichen selbst in einer Zeichenkette vor, so muss es als Escape-Code angegeben werden. Zeichenkettenliterale können beliebig lang sein.

¹⁹ In der Unicode-Version von B++ ist der größtmögliche Zeichencode FFFFH.

Beispiele für Zeichenkettenlitterale:

```
"Hello World!"  
"Er sagte \"Ich bin müde\"."   
"Adresse:\n\tName:\n\tStrasse:\n\tOrt:\n"  
"c:\\prog\\bp2\\bpp -c hello -o hello.bpm"  
"Dieser String sagt auf der Konsole\7 'Piep'!"
```

Wie in C++ können Zeichenkettenlitterale direkt verkettet werden. Dies erlaubt das einfache Verteilen langer Zeichenkettenlitterale auf mehrere Zeilen.

Beispiel:

```
var s = "Dieser String "  
        "gehört zusammen."
```

Anders als in C/C++ sind Zeilenumbrüche innerhalb von Zeichenkettenlitteralen ohne Backslash (\) am Zeilenende zulässig. Steht ein Backslash am Zeilenende, wird der Zeilenumbruch in der resultierenden Zeichenkette unterdrückt.

Beispiel:

```
"Man glaubt es kaum:  
    Diese Zeichenkette geht  
        über mehrere Zeilen \  
und merkt sich auch die Einrückungen  
    und Umbrüche."
```

B++ erlaubt auch die Verwendung von „Raw-Strings“, analog zu C++11. Ein Raw-String ist dabei eine Folge beliebiger Zeichen, deren Inhalt nicht interpretiert wird.

Die Notation erfolgt wie in C++11, also nach der Syntax

```
rawstring ::= 'R"' [tag] '(' { any_character } ')' [tag] '"' .
```

Dabei ist das optionale *tag* ein beliebiger Bezeichner, der als erweitertes Endekennzeichen verwendet wird, falls die Zeichenfolge ')' '"' im Text selbst vorkommt.

Beispiele:

```
var s1 = R"(Hier werden \n oder \0xFF und " nicht interpretiert.  
Auch Zeilenumbrüche werden einfach übernommen.)";  
  
var s2 = R"end( Wenn die Kombination )" im Text vorkommt,  
benötigt man ein Tag zur Kennzeichnung des Endes.)end";
```

4.3.3 Ganzzahlige Litterale

Ganzzahlige Litterale sind Darstellungen ganzer Zahlen in dezimaler, binärer, oktaler oder hexadezimaler Schreibweise, denen optional ein Vorzeichen (+ oder -)²⁰ vorangestellt sein kann. Dabei werden Oktalzahlen – wie in C – dadurch gekennzeichnet, dass die erste Ziffer eine Null ist und alle weiteren Ziffern im Bereich zwischen 0 und 7 liegen. Binärzahlen beginnen mit 0b, Hexadezimalzahlen mit 0x. Alle anderen Zahlen sind Dezimalzahlen²¹. Standardmäßig werden ganzzahlige Litterale intern als 32-Bit-Integer (Typ `int`) abgebildet und haben somit einen Wertebereich von -2^{31} bis $+2^{31}-1$. Überschreitet der Wert eines Literals diesen Bereich, erfolgt die interne Darstellung als 64-Bit-Integer (Typ `long`), Wertebereich -2^{63} bis $+2^{63}-1$.

Wie in C++ können dem Literal die Suffixe U und L (auch kombiniert, Groß- bzw. Kleinschreibung wird nicht berücksichtigt) angehängt werden, um die interne Darstellung als vorzeichenlose Kon-

²⁰Genau genommen gehört das Vorzeichen nicht zum Literal, sondern ist ein einstelliger Operator, der dem Literal vorangestellt ist.

²¹Eine ganze Dezimalzahl kann nicht mit der Ziffer 0 beginnen!

stante bzw. in 64 Bit Breite festzulegen.

Wie in C++11 ist es zulässig, in die Notation Hochkommata zur Trennung von Zifferngruppen (mit dem Ziel besserer Lesbarkeit) einzufügen.

Beispiel 3 zeigt einige mögliche Varianten der Notation:

```
17                // 32-Bit-Integer in Dezimaldarstellung
-234567L          // 64-Bit-Integer in Dezimaldarstellung
0x7FFFFFFFUL      // vorzeichenloser 64-Bit-Integer, hexadezimal
021U              // vorzeichenloser 32-Bit-Integer, oktal
0b1000'1001'0001 // 32-Bit-Integer in Binärdarstellung mit Zifferngruppen
0x2dff0           // 32-Bit-Integer in Hexadezimaldarstellung
```

Beispiel 3 Notation ganzzahliger numerischer Literale

4.3.4 Gleitkommalliterale

Gleitkommalliterale sind Darstellungen reeller²² Zahlen nach der Syntax

`[+|-] digit[.digit {digit}][e|E [+|-] digit{digit}]`.

Dabei ist *digit* eine Dezimalziffer. Die interne Darstellung erfolgt standardmäßig mit doppelter Genauigkeit (8 Byte, Typ `double`), wodurch die Genauigkeit auf 15 Stellen der Mantisse begrenzt ist und der Wertebereich zwischen $-1.7 \cdot 10^{308}$ und $1.7 \cdot 10^{308}$ liegt. Mit dem Suffix `F` (bzw. `f`) wird explizit die Verwendung einfacher Genauigkeit (4 Byte, Typ `float`, Wertebereich zwischen $-3.4 \cdot 10^{38}$ und $3.4 \cdot 10^{38}$ bei 7 Stellen der Mantisse) festgelegt.

Beispiele für Gleitkommalliterale:

```
3.141593          // double
-123.4567E23      // double, Exponentialdarstellung
123e-4F           // float, Exponentialdarstellung, keine Nachkommastellen
-1.234567e-2f     // float, Exponentialdarstellung
2.5E+6            // double, mit optionalem + vor Exponenten
3E14              // double, Exponentialdarstellung, keine Nachkommastellen
```

Beispiel 4 Notation von Gleitkommalliteralen

4.3.5 Erweiterte Literale

Variablen der Container-Datentypen `vector` und `dictionary` können mit Hilfe von Literalen initialisiert werden. Außerdem können Literale überall dort verwendet werden, wo eine konstante Instanz eines solchen Containers benötigt wird. Die Notation dieser Literale wird im Zusammenhang mit den entsprechenden Datentypen (Abschnitt 5.2.2) beschrieben.

Funktionsliterale erlauben das Erstellen sogenannter *anonymer Funktionen*, die einer Variablen als Wert zugewiesen, als temporäre *inline*-Funktionen oder ähnlich den aus C/C++ bekannten Funktionsmakros verwendet werden können. Ihre ausführliche Erläuterung erfolgt in Abschnitt 9.7.

²²Wegen der endlichen Länge der Mantisse in der internen Darstellung ist das nicht ganz korrekt. Eigentlich ist nur eine Teilmenge der rationalen Zahlen exakt darstellbar.

4.3.6 Literale Ausdrücke

B++ erlaubt das Bilden von *literalen Ausdrücken*. Damit sind Ausdrücke gemeint, in denen alle Operanden selbst Literale darstellen. In solchen Ausdrücken können alle Operatoren (siehe Kapitel 6) verwendet werden, die auf konstante Operanden anwendbar sind.

Literale Ausdrücke werden nicht zur Laufzeit, sondern direkt vom Compiler ausgewertet. Ihr Ergebnis ist wiederum ein Literal. Sie können in einem Programm überall dort verwendet werden, wo syntaktisch ein konstanter Wert zulässig ist.

Darüber hinaus sind sie im Zusammenhang mit der bedingten Übersetzung (vgl. 4.5.6) zur Auswertung von Bedingungen von Bedeutung.

4.3.7 Benannte Literale

Literale kann ein symbolischer Name (ein Bezeichner) zugewiesen werden. Auf diese Weise lassen sich symbolische Konstanten oder – mit Hilfe von Funktionsliteralen – *inline-Funktionen* erzeugen. Die Definition benannter Literale erfolgt mit den Verarbeitungsanweisungen `#deflit` bzw. `#literal`. Sie wird in Abschnitt 4.5.4 ausführlicher beschrieben.

Hinweis:

Weil der Compiler jede einzelne Quelldatei als separate Übersetzungseinheit auffasst, sind die Bezeichner benannter Literale nur innerhalb der Quelldatei gültig, in der sie definiert wurden. Will man z. B. so definierte symbolische Konstanten in mehreren Übersetzungseinheiten verwenden, sollten sie deshalb in einer eigenen (Header-)Datei definiert und diese Datei mit `#include` in die jeweiligen Quelldateien eingebunden werden.

4.3.8 Vordefinierte Literale

B++ kennt keinen speziellen Boolean-Typ, jedoch existieren die vordefinierten Literale `true` (entspricht dem Integer-Wert 1) und `false` (entspricht dem Integer-Wert 0), die zur besseren Lesbarkeit für Wahrheitswerte verwendet werden können.

Die vordefinierten Literale `UNICODE`, `WIN32`, `WINCE`, `MSDOS`, `UNIX`, `LINUX` und `BPPI64`, geben darüber Auskunft, für welche Plattform der gerade verwendete B++-Compiler selbst übersetzt wurde und ob es sich um die Unicode-Version handelt. Diese Literale haben den Wert 1 (bzw. `true`), wenn beim Kompilieren die jeweilige Gegebenheit zutrifft, und 0 (bzw. `false`), wenn nicht.

Als die (leider einzigen) standardisierten Fehlerbezeichner der C-Bibliothek sind die Literale `EDOM` (Domain Error) und `ERANGE` (Range Error) vordefiniert. Sie können im Zusammenhang mit numerischen Funktionen (siehe Abschnitt 11.6) bei unzulässigen Argumentwerten oder Zahlbereichsüberschreitungen auftreten.

Zur Vereinfachung der Arbeit mit Typinformationen sind benannte Literale als symbolische Namen der von B++ unterstützten Datentypen (vgl. Abschnitt 5.2) vordefiniert. Tabelle 2 enthält dazu eine vollständige Übersicht.

Die Spalte „Literal“ enthält darin die symbolischen Namen, „Typ-ID“ ist der jeweils zugehörige Zahlenwert, der z. B. von den vordefinierten Funktionen `gettype` und `gettypename` (siehe Abschnitt 11.2) verwendet wird. Die in der Spalte „Datentyp“ **fett und unterstrichen** gesetzten Wörter können als *Typbezeichner* für die *explizite Typisierung* von Variablen verwendet werden – siehe Abschnitt 5.3.

Ebenfalls vordefiniert sind die Literale `LC_ALL` (0), `LC_COLLATE` (1), `LC_CTYPE` (2), `LC_MONETARY` (3), `LC_NUMERIC` (4): und `LC_TIME` (5) für die Kategorien der `setlocale`-Funktion (siehe Abschnitt 11.7)

Darüber hinaus existieren in Anlehnung an den C99-Standard die vordefinierten Literale `__func__` (string), `__FILE__` (string) und `__LINE__` (int) zur Angabe des aktuellen Methoden- oder Funktionsnamens, des Dateinamens bzw. der aktuellen Zeilennummer während

der Übersetzung. Die Werte dieser Literale werden vom Scanner in Abhängigkeit von der aktuellen Position im Quelltext gebildet.

Literal	Typ-ID	Datentyp
BVT_NULL	0	<u>null</u>
BVT_VOID	1	kein Wert (Ergebnistyp für Funktionen)
BVT_AUTO	2	uninitialisierte <u>auto</u> -Variable
BVT_SBYTE	64	<u>sbyte</u>
BVT_BYTE	65	<u>byte</u>
BVT_SHORT	68	<u>short</u>
BVT_USHORT	69	<u>ushort</u>
BVT_INT	72	<u>int</u>
BVT_UINT	73	<u>uint</u>
BVT_INTPTR	74	<u>intptr</u>
BVT_LONG	76	<u>long</u>
BVT_ULONG	77	<u>ulong</u>
BVT_FLOAT	104	<u>float</u>
BVT_DOUBLE	108	<u>double</u>
BVT_FILE	128	<u>FILE</u>
BVT_CODE	129	<u>function</u> (nativ)
BVT_VAR	130	Variable/Element in dictionary
BVT_OBJECT	160	<u>object</u>
BVT_STRONGOBJECT	161	Objektvariable mit fester Klasse
BVT_CLASS	168	<u>class</u>
BVT_STRING	176	<u>string</u>
BVT_BUFFER	177	<u>buffer</u>
BVT_CHARBUFFER	178	<u>charbuffer</u>
BVT_VECTOR	180	<u>vector</u>
BVT_BYTECODE	181	<u>function</u> (B++)
BVT_LOCALVARINFO	182	Variableninformation (intern)
BVT_DICTIONARY	184	<u>dictionary</u>
BVT_OBJECTREF	224	Referenz auf <u>object</u>
BVT_CLASSREF	232	Referenz auf <u>class</u>
BVT_STRINGREF	240	Referenz auf <u>string</u>
BVT_BUFFERREF	241	Referenz auf <u>buffer</u>
BVT_CHARBUFFERREF	242	Referenz auf <u>charbuffer</u>
BVT_VECTORREF	244	Referenz auf <u>vector</u>
BVT_BYTECODEREF	246	Referenz auf <u>function</u> (B++)
BVT_DICTIONARYREF	248	Referenz auf <u>dictionary</u>

Tabelle 2 Benannte Datentyp-Literale

4.4 Definitionen, Anweisungen und Blöcke

Wie bereits im Kapitel 3 erwähnt, besteht ein B++-Programm auf der obersten Quelltextebene im Wesentlichen aus Klassendeklarationen (siehe Kapitel 10), Variablen- und Funktionsdefinitionen. Hinzu kommen noch Verarbeitungsanweisungen – siehe 4.5.

Funktionsdefinitionen bestehen aus einem *Kopf*, der u. a. den Namen des jeweiligen Codeobjekts beinhaltet, sowie einem *Block*, der den eigentlichen Inhalt einschließt.

Ein Block ist eine Zusammenfassung von aufeinander folgenden *Anweisungen* und (optional) *Variablendefinitionen*.

In B++ wird ein Block – wie u. a. in C/C++ – durch geschweifte Klammern ({ und }) begrenzt.

Jede in einem Block enthaltene Anweisung oder Definition muss – ebenfalls wie in C/C++ – mit einem Semikolon (;) abgeschlossen²³ werden. Überall dort, wo syntaktisch eine Anweisung vorgesehen ist, kann wiederum ein Block stehen. Dies ist besonders im Zusammenhang mit den Steuerstrukturen (siehe Kapitel 8) von Bedeutung.

Nachfolgend werden die beschriebenen Elemente anhand eines etwas ausführlicheren Beispiels veranschaulicht. Es handelt sich dabei um ein Programm, das die zeilenweise Eingabe von Zeichenketten (z. B. Namen) erwartet, sie in einer sortierten Liste speichert und am Ende den Listeninhalt wieder ausgibt. Das Beispiel geht über das bisher Besprochene hinaus, indem von Elementen (Klassen, Funktionen, Steuerstrukturen) Gebrauch gemacht wird, deren Erläuterung Gegenstand späterer Abschnitte ist. Es ist jedoch ein „typisches“ B++-Programm und sollte – einige Programmierkenntnisse in anderen Sprachen vorausgesetzt – leicht nachvollziehbar sein.

```

1 /* sorted stringlist example */
2
3 // =====
4 // Class for single entry of list
5 class ListEntry
6 {
7 // private elements
8 private:
9     // value of the entry
10    string _strValue;
11    // reference to next entry
12    ListEntry _next = null;
13 // public elements
14 public:
15     // constructor
16    ListEntry(string str) { _strValue = str; }
17    // gets current value
18    string getValue() { return _strValue; }
19    // gets reference to next entry
20    ListEntry getNext() { return _next; }
21    // sets reference to next entry
22    void setNext(ListEntry entry) { _next = entry; }
23 };
24 // =====
25 // Declaration of List class
26 class List
27 {
28     // private elements
29 private:
30     // pointer to first ListEntry object
31     ListEntry _first = null;
32
33     // public elements
34 public:
35     // constructor (does nothing)
36     List() {}
37     // adds entry to list
38     ListEntry addEntry(string str);
39     // prints entries to console
40     void printList();
41 };

```

²³ Das ist ein feiner aber wichtiger Unterschied zu Sprachen wie z. B. PASCAL, wo das Semikolon die Anweisungen nicht abschließt, sondern *trennt*. In B++ muss das Semikolon auch hinter der letzten Anweisung eines Blocks oder vor einem Schlüsselwort geschrieben werden.

```

42 // =====
43 // Implementation of List class
44
45 // add new entry to sorted list
46 ListEntry List::addEntry(string str) {
47     // define local variable newentry
48     ListEntry newentry(str);           // create new entry
49
50     if (!_first)                       // nothing to compare in empty list
51         _first = newentry;             // single statement
52     else
53     {                                   // statement block
54         if (strcmp(_first->getValue(),str) > 0)
55         {
56             // place new entry at start of list
57             newentry->setNext(_first); // -> operator references a
58                                     // member function
59             _first = newentry;
60         }
61         else
62         {
63             // Walk through the list until correct position found.
64             // Local Variable p references current list element.
65             for(ListEntry p=_first;p->getNext();p=p->getNext())
66             {
67                 if (strcmp(p->getNext()->getValue(),str) > 0)
68                 {
69                     newentry->setNext(p->getNext());
70                     break;           // insertion is done so return
71                 }
72             }
73             p->setNext(newentry);     // close chain of entries
74         }
75     }
76     return newentry; // return reference to new entry
77 }
78
79 // print sorted values to console
80 void List::printList() {
81     ListEntry p = _first;    // Local variable references list element.
82     while ( p != null )
83     {
84         print(p->getValue(), "\n");
85         p = p->getNext();
86     }
87 }
88 // =====
89 // programs entry point
90 void main() {
91     List nameList;
92     // read text lines from console until empty line was read
93     while (true)    // loops forever
94     {
95         print("Name: ");
96         string s = gets();
97         if (strlen(s) == 0)
98             break;    // break on empty string
99         nameList->addEntry(s);
100     }
101     print("---- sorted ----\n");
102     nameList->printList();
103 }

```

Beispiel 5 Sortierte Liste mit Ein- und Ausgabe

Zur Verwaltung der eingegebenen Daten wird eine einfach verkettete Liste (Klasse `List`) definiert, deren Elemente Objekte der Klasse `ListEntry` sind. Jedes Element enthält neben dem Verweis auf seinen Nachfolger (`_next`) die als Datenwert gespeicherte Zeichenkette `_strValue`. Die Liste selbst besitzt einen Verweis auf das erste Listenelement (`_first`).

Das Einfügen neuer Elemente in die Liste erfolgt mit der Methode `List::addEntry`, die aus der übergebenen Zeichenkette ein neues Listenelement erzeugt und in die vorhandene Liste so einfügt, dass eine aufsteigende Sortierung gewährleistet ist.

Die Methode `printList` der Klasse `List` iteriert einfach über die Liste und gibt die einzelnen Zeichenketten zeilenweise auf die Konsole aus.

Das Hauptprogramm (Funktion `main`, Zeile 90) legt in der lokalen Variablen `nameList` eine Instanz der Klasse `List` an. Anschließend werden in einer Endlosschleife (Zeilen 93 bis 100) so lange Namen von der Konsole gelesen und in die Liste eingefügt, bis eine leere Zeichenkette eingegeben wurde. Zum Schluss erfolgen die Ausgabe der sortierten Liste (Zeile 102) und die Rückkehr zum Betriebssystem.

Die Implementierung der Klasse `ListEntry` erfolgt hier vollständig „*inline*“, d. h. direkt innerhalb der Klassendefinition. Bei der Klasse `List` werden die umfangreicheren Methoden `addEntry` und `printList` separat implementiert. Beide Varianten sind gleichwertig, die Syntax zur separaten Implementierung entspricht (weitgehend) der von C++.

4.5 Verarbeitungsanweisungen

Verarbeitungsanweisungen werden – im Gegensatz zu allen anderen Anweisungen – vom Compiler während der Übersetzung des Quellcodes und nicht vom Bytecode-Interpreter ausgeführt.

Obgleich B++ keinen Präprozessor besitzt, haben sie eine gewisse Ähnlichkeit zu Präprozessorbefehlen in C/C++ und werden auch so notiert. Insbesondere bedeutet dies, dass sie **nicht** mit einem Semikolon abgeschlossen werden.

Zudem können Verarbeitungsanweisungen – mit Ausnahme der Kennzeichnung eines literalen Ausdrucks (siehe 4.5.7) - nicht innerhalb eines Ausdrucks notiert werden.

Nachfolgend werden die in B++ verfügbaren Verarbeitungsanweisungen im Einzelnen beschrieben.

4.5.1 Die `#include`-Anweisung

Mit der Verarbeitungsanweisung `#include` wird eine Quelldatei in den aktuell übersetzten Quelltext eingebunden, als ob ihr Inhalt an der Stelle der `#include`-Anweisung notiert worden wäre.

Wie in C bzw. C++ gibt es zwei Notationsformen:

```
#include "filename.ext" bzw.
```

```
#include <filename.ext>.
```

Der Unterschied zwischen den beiden Notationsformen besteht in der Art und Weise, auf die die Suche nach der einzubindenden Datei erfolgt.

Bei der ersten Variante erfolgt die Suche zunächst im aktuellen Verzeichnis (dem Verzeichnis der Datei, die die `#include`-Anweisung enthält), anschließend in allen Verzeichnissen der Dateien, die übergeordnete `#include`-Anweisungen enthalten, und schließlich entlang des über die Umgebungsvariable `BPPINC` (vgl. Abschnitt 2.5) spezifizierten Suchpfades. Bei der zweiten Variante entfällt die Suche in „übergeordneten“ Verzeichnissen.

In beiden Fällen kann vor dem Dateinamen ein relativer oder absoluter Verzeichnispfad angegeben werden.

4.5.2 Die #use-Anweisung

Die #use-Anweisung bindet eine vorkompilierte Bytecode-Bibliothek in das aktuelle Programm ein. Die in der Bibliothek enthaltenen Symbole werden damit in das Programm übernommen. Das bedeutet, dass beim Übersetzen eines Programms, das mit #use eine Bibliothek einbindet, Bytecode entsteht, der den kompletten Inhalt der Bibliothek mit enthält.

Verwendung: #use "filename.ext"

Dem Dateinamen kann ein relativer oder absoluter Pfad vorangestellt sein. B++ sucht die angegebene Datei zunächst im aktuellen Verzeichnis (dem Verzeichnis der Datei, die die #use-Anweisung enthält). Hat die Suche dort keinen Erfolg, so werden zusätzlich die in der Umgebungsvariablen BPPLIB angegebenen Verzeichnisse durchsucht.

Anwendungsbeispiel:

Eine Bibliothek soll den Namen `tstfunc.bpp` haben und eine Funktion mit dem Namen `testfunc` exportieren.

```
/* using #use example - module tstfunc.bpp */
void testfunc(var n)
{
    print("testfunc(",n,") from tstfunc.bpp\n");
}
```

Die Bibliothek wird nun in Bytecode übersetzt:

```
bp2 -c tstfunc -o tstfunc.bpm
```

... und von nachfolgendem Programm benutzt:

```
/* using #use example - main module */
#use "tstfunc.bpm"

void main()
{
    testfunc("Hello World");
}
```

Beispiel 6 Verwendung der #use-Anweisung

4.5.3 Die #import-Anweisung

Die #import-Anweisung erlaubt das Einbinden dynamischer B++-Erweiterungsbibliotheken (DLLs; siehe Abschnitt 15.2.2) zur Übersetzungszeit.

Verwendung: #import "dateiname.ext"

Auch hier kann dem Dateinamen ein absoluter oder relativer Pfad vorangestellt sein.

Die Suche nach der zu ladenden Bibliothek erfolgt zuerst im aktuellen Verzeichnis (dem Verzeichnis der Datei, die die #import-Anweisung enthält) und anschließend im durch die Umgebungsvariable PATH spezifizierten Suchpfad²⁴.

4.5.4 Die Anweisungen #deflit, #define, #literal, #undeflit und #undef

Diese Verarbeitungsanweisungen dienen der Definition benannter Literale (vgl. 4.3.7) bzw. der Aufhebung der Definition eines benannten Literals.

²⁴Unter Windows werden die spezifischen Regeln zur Suche nach DLLs verwendet. Demnach wird auch im Verzeichnis des aufrufenden Programms sowie im Windows-Verzeichnis gesucht.

Mit

```
#deflit <identifizier> <litexpr> bzw.
```

```
#literal <identifizier> <litexpr>
```

wird ein benanntes Literal mit dem angegebenen Bezeichner definiert. Als zugeordneter Wert kann dabei ein beliebiger literaler Ausdruck stehen.

Der Unterschied zwischen beiden Formen besteht darin, dass `#literal` nur dann wirksam wird, wenn unter dem angegebenen Namen noch keine Literaldefinition vorhanden ist, während `#deflit` eine vorhandene Definition ggf. überschreibt.

`#define` ist ein Alias für `#deflit`, der der vereinfachten Dokumentation mit Doxygen und ähnlichen Hilfsmitteln dient.

Mit `#undeflit <identifizier>` kann eine bestehende Definition eines benannten Literals aufgehoben werden. Auch hierzu existiert mit `#undef` ein Alias.

Beispiele:

```
#deflit PI 3.141593

#literal TRUE 1
#literal FALSE 0

#literal HIWORD function(x) { return (x >> 16) & 0xFFFF;}
#literal LOWORD function(x) { return x & 0xFFFF;}
```

Beispiel 7 Verwenden von `#deflit` und `#literal`

Die letzten beiden Zeilen definieren zwei Funktionen als Literale, die das höherwertige bzw. niederwertige 16-Bit-Wort aus einem 32-Bit-Wort extrahieren. Diese Form ist mit Präprozessor-Funktionsmakros in C bzw. C++ vergleichbar. Der wesentliche Unterschied besteht darin, dass es sich hier um „echte“ (anonyme) Funktionen handelt, nicht um Präprozessormakros.

4.5.5 Die `#defvar`-Anweisung

Mit `#defvar` können initialisierte globale Variablen innerhalb und außerhalb von Funktionen definiert werden.

Verwendung:

```
#defvar <varname> <value>
```

Dabei ist `<varname>` ein gültiger Bezeichner und `<value>` der zugehörige Wert. Auch hier ist zu beachten, dass die Initialisierung des Variablenwerts zur Übersetzungs- und nicht zur Laufzeit erfolgt. Deshalb können für `value` nur Literale (bzw. literale Ausdrücke) angegeben werden.

Beispiele:

```
#defvar Version "2.1.0"
#defvar bufSize 2048
```

Hinweis:

Im Unterschied zur Definition benannter Literale (vgl. 4.3.7 bzw. 4.5.4), deren Auswertung erst bei ihrer Verwendung erfolgt, werden hier tatsächlich „echte“ Variablen angelegt und mit dem Wert des als Parameter angegebenen Literals initialisiert. Die Sichtbarkeit dieser Variablen ist *nicht* auf die jeweilige Übersetzungseinheit beschränkt.

4.5.6 Die Anweisungen `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` und `#endif`

Mit Version 1.2 unterstützt B++ die bedingte Übersetzung von Quelltextabschnitten in einer Notation, die weitestgehend der des C-Präprozessors entspricht.

Tabelle 3 gibt eine Übersicht über Syntax und Bedeutung der Anweisungen zur bedingten Übersetzung.

Anweisung	Bedeutung
<code>#ifdef <i>literalname</i></code>	Übersetzung, falls <i>literalname</i> definiert ist
<code>#ifndef <i>literalname</i></code>	Übersetzung, falls <i>literalname</i> nicht definiert ist
<code>#if <i>literal_expression</i></code>	Übersetzung, falls <i>literal_expression</i> den Wert <i>true</i> ergibt
<code>#elif <i>literal_expression</i></code>	bedingter Alternativzweig; Übersetzung, falls <i>literal_expression</i> den Wert <i>true</i> ergibt
<code>#else</code>	unbedingter Alternativzweig
<code>#endif</code>	Ende des bedingten Übersetzungsabschnitts

Tabelle 3 Übersicht der Anweisungen zur bedingten Übersetzung

Darin meint *literalname* den Bezeichner eines benannten Literals (vgl. 4.3.7) und *literal_expression* einen literalen Ausdruck (vgl. 4.3.6)

Innerhalb eines literalen Ausdrucks kann wiederum der Pseudo-Operator

```
defined(literalname)
```

verwendet werden, um die Existenz eines benannten Literals zu prüfen. Dabei ist die Klammerung von *literalname* optional.

4.5.7 Kennzeichnen literaler Ausdrücke mit `#()`

Der Compiler von B++ führt keine automatische Optimierung von Code durch. Das bedeutet u. a., dass die Auswertung von Ausdrücken im Normalfall zur Laufzeit erfolgt, sofern sie nicht zur statischen Initialisierung von Variablen verwendet werden – auch dann, wenn ihre Operanden Literale (und damit konstant) sind.

Mit Hilfe der Verarbeitungsanweisung

```
#(<expr>)
```

kann ein Ausdruck `<expr>` explizit als literaler Ausdruck gekennzeichnet und so seine Berechnung durch den Compiler (d. h. während der Programmübersetzung) erzwungen werden, was zu kompakterem und vor allem schnellerem Code führt.

4.5.8 Die `#endsrc`-Anweisung

Die `#endsrc` Anweisung kennzeichnet das (vorzeitige) Ende eines B++-Quelltextes, oder anders ausgedrückt, jeglicher Text hinter `#endsrc` wird vom Compiler ignoriert. Der Anwendungszweck besteht vor allem in der Einbindung von B++-Quellen in Batch-Scripts (vgl. 4.1).

Auch hierzu ein Beispiel. Das folgende Programm ist ein Batch-Script für MS-DOS mit dem Namen `graph.bat`, das B++-Quelltext enthält:

```
@bp2 graph.bat
@goto end
// global settings
#define sizeX 240
```

```

#define sizeY 64
#define graphMode 4
#define textMode 7
main() {
    var px=sizeX;
    var py=sizeY;
    setscrmode(graphMode);           // switch to graph mode
    var ms = timer();
    var n=0;
    for (var x=0;x<px;x+=2) {
        ++n;
        var x2= px -x;
        line(x,0,x2,py-1,1);
    }
    for (var y=0;y<py;y+=2) {
        ++n;
        var y2= py-1 -y;
        line(0,y,px-1,y2,2);
    }
    ms = timer() - ms;
    var r = n*1000 / ms;
    gets();
    setscrmode(textMode);           // switch back to text mode
    print(r, " lines per second\n");
}
#endsrc
:end

```

Beispiel 8 Script-Einbettung von B++-Quellen

Das Programm²⁵ füllt einen Bildschirmbereich (mit `sizeX` und `sizeY` festgelegt) mit Linien und misst die dafür benötigte Zeit. Am Ende wird die Zeichengeschwindigkeit (Linien je Sekunde) ausgegeben.

Die ersten beiden Zeilen enthalten Anweisungen für den DOS-Kommandoprozessor. Das vorangestellte `@`-Zeichen unterdrückt unter DOS das Bildschirmecho und kennzeichnet die Zeilen unter B++ als Kommentare. Der B++-Quelltext endet mit der `#endsrc` Anweisung.

Die letzte Zeile gehört wieder dem Kommandoprozessor – es ist die das Programmende kennzeichnende Sprungmarke.

²⁵ Dieses Beispiel ist nur in DOS-Umgebungen lauffähig, da es systemspezifische Funktionen zur Grafikansteuerung verwendet.

5 Variablen und Datentypen

B++ ist primär eine Sprache, die gewöhnlich als *schwach* oder *dynamisch* typisiert bezeichnet wird. Etwas genauer bedeutet dies:

In B++ haben nicht die Variablen, sondern allein die Werte einen Datentyp.

Die optionale statische Typisierung wird erreicht, indem Variablen mit der Restriktion versehen werden, dass sie nur Werte vom selben Typ aufnehmen dürfen, wie der Wert, mit dem sie initialisiert wurden.

5.1 Variablen

Variablen sind im Grunde einfach Ablageorte für einen beliebigen Wert, die einen weitgehend frei wählbaren Bezeichner tragen.

In B++ müssen Variablen *grundsätzlich vor ihrer ersten Verwendung* definiert werden.

Wird eine Variable auf der „äußersten“ Programmebene – also außerhalb von Klassen oder Funktionen – definiert, so ist ihr Gültigkeitsbereich *global*, d. h. auf die Variable kann von jeder Stelle des Programms aus zugegriffen werden. Dies kann bei größeren Programmen zu unerwünschten Seiteneffekten führen, wenn man versehentlich an verschiedenen Stellen ein- und dieselbe Variable zu unterschiedlichen Zwecken benutzt.

Um diese Gefahr gering zu halten, gibt es in B++ *keine implizite Definition globaler Variablen durch Wertzuweisung* an einen beliebigen Bezeichner, wie das z. B. in JavaScript möglich ist. Eine globale Variable muss deshalb explizit definiert werden – entweder auf der globalen Programmebene (außerhalb aller Funktionen und Klassen) oder mit Hilfe der Verarbeitungsanweisung `#defvar` (vgl. 4.5.5).

In realen Programmen sind globale Variablen eher ein Ausnahmefall und werden fast ausschließlich zum Austausch von Daten zwischen einzelnen, voneinander mehr oder weniger unabhängigen Programmteilen verwendet. Um wiederverwendbare und gut wartbare Programme zu erstellen, sollten globale Variablen so weit wie möglich vermieden und – wenn sie schon unbedingt notwendig sind – gut dokumentiert werden.

Der *Gültigkeitsbereich lokaler Variablen* (und auch ihre Lebensdauer) erstreckt sich in B++ auf die *Funktion*, innerhalb der sie definiert werden. Die Definition erfolgt dabei entweder in Form einer Liste lokaler Variablenbezeichner am Ende des Funktionskopfes, durch ein Semikolon von der Parameterliste getrennt, oder durch Verwendung des Schlüsselwortes `var` bzw. eines Typbezeichners. Nachfolgend dazu ein Beispiel:

```
/* using function-level variables */
// calculate n!
var fac(n; i,f)
{
    if (n==0)
        return 1;
    for(f=n, i=n-1; i>1; --i)
        f *= i;
    return f;
}

void main()
{
    for (var i=1;i<=10;++i)
        print("fac(",i,")=" , fac(i) , "\n");
}
```

Beispiel 9 Verwenden von lokalen Variablen

Bei der Entwicklung neuer Programme ist die Variablendeklaration/-definition mit `var`²⁶ (oder einer expliziten Typangabe) zu bevorzugen. Sie ermöglicht die Definition lokaler Variablen an beliebiger Stelle im Implementierungsrumpf einer Funktion bei gleichzeitiger Initialisierung. Das verbessert einerseits die Lesbarkeit des Codes, andererseits kann sich dadurch aber auch die Effizienz des Programms verbessern, indem Initialisierungen nur dann ausgeführt werden, wenn dies auch nötig ist, d. h. der betreffende Programmzweig tatsächlich erreicht wird.

5.2 Datentypen

In B++ werden alle in einem Programm vorkommenden Werte in sogenannten *Value-Objekten* abgelegt. Ein solches Objekt besitzt im Wesentlichen eine Typinformation sowie den jeweils *zugeordneten Wert selbst* oder eine *Referenz* darauf. Dementsprechend kann man zwischen Werttypen und Referenztypen unterscheiden²⁷. Diese Unterscheidung ist für die Verwendung in zweierlei Hinsicht von Bedeutung:

- Bei der Zuweisung an Variablen werden Werttypen vollständig kopiert, bei Referenztypen dagegen nur der Verweis auf den Datenbereich. Deshalb betreffen hier Datenänderungen *alle* Variablen, die auf den jeweiligen Wert verweisen.

Ein Beispiel:

```
main()
{
    var s1 = "Hello";           // initialize s1 as string
    var s2 = s1;                // assign variable s1 to s2
    s2[1] = 'a';                // modify second character in s2
    print(s1, "\n");           // print value of s1
}
```

Beispiel 10 Zuweisung von Referenztypen

Hier wird „Hallo“ und nicht – wie man vielleicht vermuten könnte – „Hello“ ausgegeben. Der einfache Grund liegt darin, dass die Variablen `s1` und `s2` lediglich Verweise auf ein- und denselben Datenbereich beinhalten. Eine „echte“ Kopie der Zeichenkette in `s1` ließe sich beispielsweise mit Hilfe der Funktion `string` (vgl. Abschnitt 11.2) erzeugen.

- Die erwähnten Value-Objekte und mit ihnen die Daten der Werttypen werden auf dem Stack angelegt und mit demselben aufgeräumt. Anders verhält es sich mit den Referenztypen. Deren Datenbereich wird im Normalfall mit Hilfe einer Referenzzählung verwaltet, d. h., er wird freigegeben, wenn er von keiner Variablen mehr benutzt wird. Darüber hinaus gibt es auch die Möglichkeit einer expliziten Speicherverwaltung durch das Anwenderprogramm.

²⁶ Die Variablenliste im Funktionskopf war die einzige Möglichkeit der Deklaration lokaler Variablen in BOB bzw. BOB+ 1.x. In B++ existiert diese Form nur aus Gründen der Abwärtskompatibilität.

²⁷ Technisch gesehen sind die Typen, deren Wert nicht mehr als 8 Bytes benötigt, Wert-, alle anderen Referenztypen. Hier wird der eigentliche Datenbereich dynamisch erzeugt und im Value-Objekt nur ein Zeiger darauf abgelegt.

5.2.1 Werttypen

Werttypen sind in B++ die numerischen Typen sowie der Typ `null`.

Integrale Datentypen

Diese Typen repräsentieren ganzzahlige numerische Werte unterschiedlicher Genauigkeit. Tabelle 4 gibt hierzu eine Übersicht.

Typ	Breite (Bits)	Wertebereich	Bemerkung
<code>sbyte</code>	8	-128 ... 127	Byte mit Vorzeichen (wie <code>char</code> in C)
<code>byte</code>	8	0 ... 255	Byte ohne Vorzeichen (wie <code>unsigned char</code> in C)
<code>short</code>	16	$-2^{15} \dots 2^{15}-1$	WORD mit Vorzeichen
<code>ushort</code>	16	$0 \dots 2^{16}-1$	WORD ohne Vorzeichen (wie <code>unsigned short</code> in C)
<code>int</code>	32	$-2^{31} \dots 2^{31}-1$	DWORD mit Vorzeichen
<code>uint</code>	32	$0 \dots 2^{32}-1$	DWORD ohne Vorzeichen
<code>intptr</code>	32 oder 64	wie <code>int</code> oder <code>long</code> (plattformabhängig)	Repräsentation einer Adresse
<code>long</code>	64	$-2^{63} \dots 2^{63}-1$	wie <code>long long</code> in C
<code>ulong</code>	64	$0 \dots 2^{64}-1$	wie <code>unsigned long long</code> in C

Tabelle 4: Integrale Datentypen

Anders als in C/C++ werden vorzeichenbehaftete bzw. vorzeichenlose Typen nicht durch einen Modifizierer (`signed` bzw. `unsigned`) unterschieden, sondern die Unterscheidung gehört direkt zum Typbezeichner. Außerdem haben die integralen Typen in B++, mit Ausnahme von `intptr`, eine feste Breite, unabhängig von Compiler und Ausführungsplattform. Dies erleichtert die Übertragung von Programmen – auch in vorkompilierter Form – zwischen verschiedenen Plattformen.

Der Datentyp `intptr` dient der, gewöhnlich temporären, Aufnahme plattformspezifischer Adressen. Er ist deshalb auf 16- und 32-Bit-Plattformen 32 Bit bzw. auf 64-Bit-Plattformen 64 Bit breit. Intern wird `intptr` wie eine Ganzzahl mit Vorzeichen behandelt, so dass auch Adressrechnungen ausgeführt werden können.

Der Grundtyp der integralen Datentypen ist der Datentyp `int`. Dies bedeutet, dass ganzzahlige Literale als `int` interpretiert werden, sofern sie ihrem Wert nach auf den Typ `int` abgebildet werden können und nicht explizit gekennzeichnet sind (vgl. 4.3.3). Werte des Typs `int` werden auch zur Darstellung einzelner Zeichen und Boolescher Werte verwendet.

B++ kennt keinen „echten“ Booleschen Datentyp, stattdessen werden `int`-Werte verwendet, wobei der Wert Null (0) als `false` und alle anderen Werte als `true` angesehen werden. Allerdings kann man den reservierten Bezeichner `bool` zur Deklaration von Variablen verwenden, um die Intention besser zu verdeutlichen.

Auf Werte integraler Datentypen können folgende Operationen angewandt werden:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- Verwendung in String-Ausdrücken
- Bitoperationen
- numerische Vergleiche
- Verwendung als Funktionsparameter

Werte integraler Typen können in numerischen Ausdrücken und Vergleichen ohne explizite Typkonvertierung mit Werten der Typen `float` und `double` kombiniert werden. In diesem Fall erfolgt eine implizite Umwandlung des gesamten Ausdrucks nach `float` bzw. `double`. Gleiches gilt für die Zuweisung an eine statisch typisierte Variable der Typen `float` oder `double`.

Bei Verwendung von integralen Werten in String-Ausdrücken (Verkettung) erfolgt implizit eine Konvertierung in einen `int`-Wert, der als Zeichencode interpretiert wird.

Datentyp `float`

Der Datentyp `float` repräsentiert eine Gleitpunkt-Realzahl mit einfacher Genauigkeit (32 Bit). Der Wertebereich liegt zwischen $-3.4E38$ und $+3.4E38$ bei einer numerischen Genauigkeit von 6 Stellen. Kleinster darstellbarer Betrag ist $1.2E-37$.

Zulässige Operationen für den Typ `float` sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- numerische Vergleiche
- Verwendung als Funktionsparameter

Werte vom Typ `float` können in numerischen Ausdrücken und Vergleichen ohne explizite Typkonvertierung mit Werten integraler Typen und dem Typ `double` kombiniert werden. Hierbei erfolgt eine implizite Umwandlung des gesamten Ausdrucks in den „größeren“ Datentyp, d. h. bei Kombination mit integralen Typen ist der Wert vom Typ `float`, bei Kombination mit `double` vom Typ `double`. Bei einer Zuweisung an eine statisch typisierte Variable der integralen Typen oder des Typs `double` erfolgt eine implizite Konvertierung in den entsprechenden Typ.

Datentyp `double`

Der Datentyp `double` repräsentiert eine Gleitpunkt-Realzahl mit doppelter Genauigkeit (64 Bit). Der Wertebereich liegt zwischen $-1.7E308$ und $+1.7E308$ bei einer numerischen Genauigkeit von 15 Stellen. Kleinster darstellbarer Betrag ist $2.2E-308$.

Zulässige Operationen für den Typ `double` sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- numerische Vergleiche
- Verwendung als Funktionsparameter

Werte vom Typ `double` können in numerischen Ausdrücken und Vergleichen ohne explizite Typkonvertierung mit Werten integraler Typen und dem Typ `float` kombiniert werden. Hierbei erfolgt eine implizite Umwandlung des gesamten Ausdrucks in den „größeren“ Datentyp, d. h. den Typ

`double`. Bei einer Zuweisung an eine statisch typisierte Variable der integralen Typen oder des Typs `float` erfolgt eine implizite Konvertierung in den entsprechenden Typ.

Datentyp `null`

Der Datentyp `null` kennzeichnet einen nicht initialisierten Wert. Im Quelltext eines B++-Programms werden Daten dieses Typs durch das Schlüsselwort `null`²⁸ beschrieben, das man auch als Konstante und einzigen möglichen Wert des Typs `null` auffassen kann. Außerdem kann eine Reihe vordefinierter Funktionen `null` zurückgeben.

Zulässige Operationen für den Typ `null` sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter

Werte vom Typ `null` können in Vergleichen ohne explizite Typkonvertierung mit allen anderen Typen kombiniert werden. Dabei ist der Wert `null` kleiner als jeder andere Wert.

5.2.2 Referenztypen

Alle Datentypen in B++, die keine Werttypen sind, sind Referenztypen.

Datentyp `string`

Der Datentyp `string` repräsentiert eine Zeichenkette. Werte dieses Typs können mit Hilfe von Zeichenkettenliteralen (vgl. 4.3), durch die vordefinierte Funktion `newstring` (vgl. 11.1) oder die Konvertierungsfunktion `string` (siehe 11.2) explizit erzeugt werden. Darüber hinaus entstehen sie bei der Anwendung des Verkettungsoperators (+) sowie als Rückgabewert einiger weiterer vordefinierter Funktionen.

Zulässige Operationen für den Typ `string` sind:

- Zuweisung an eine Variable
- Verwendung in `string`-Ausdrücken
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter
- Zugriff auf einzelne Zeichen mit Hilfe des Operators `[]`

Hinweise:

- Der direkte Zugriff auf Elemente eines (benannten) `string`-Literal mit dem Operator `[]` ist nicht zulässig.
- Bei der Zuweisung eines `string`-Literal an eine Variable wird eine Kopie der Zeichenkette angelegt. Auf sie kann uneingeschränkt zugegriffen werden.

²⁸ Es ist nicht möglich, statisch typisierte Variablen des Typs `null` zu definieren. Dies wäre nicht sinnvoll, da eine solche Variable nur einen einzigen Wert, nämlich `null`, annehmen könnte und damit überflüssig wäre.

Datentyp `FILE`

Der Datentyp `FILE` ist ein Verweis auf eine geöffnete Datei. Instanzen dieses Typs werden mit der vordefinierten Funktion `fopen` erzeugt und mit `fclose` wieder freigegeben.

Zulässige Operationen für den Datentyp `FILE` sind:

- Zuweisung an eine Variable
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter (insbesondere bei vordefinierten I/O-Funktionen (siehe Abschnitt 11.3))

Datentyp `vector`

Der Datentyp `vector` repräsentiert ein Datenfeld fester, d. h. bei der Erzeugung festgelegter, oder dynamischer Größe, dessen Elemente beliebige, auch typverschiedene, Werte sein können. Instanzen dieses Typs werden mit den vordefinierten Funktionen `newvector` oder `Vec` bzw. `T` erzeugt.

Alternativ kann ein Wert des Typs `vector` auch als Literal erzeugt (initialisiert) werden. Hierzu wird folgende Syntax verwendet:

```
vectorliteral ::= '[' [ literal { ',' literal } ] ']' .
```

Es wird also eine in eckige Klammern eingeschlossene kommaseparierte Liste von Werten notiert, wobei die Werte selbst beliebige Literale (Vektor-Literale eingeschlossen) sind.

Beispiel 11 illustriert dies:

```
var intvec = [1,2,3,4,5];
var mixedVec = [1, "Hello", 3.14, null];
var matrix2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

Beispiel 11 Initialisierung von Vektoren mit Literalen

Zulässige Operationen für den Datentyp `vector` sind:

- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Zugriff auf einzelne Elemente mit Hilfe des Operators `[]`

Hinweise:

- Der direkte Zugriff auf Elemente eines (benannten) `vector`-Literals mit dem Operator `[]` ist nicht zulässig.
- Bei der Zuweisung eines `vector`-Literals an eine Variable wird eine Kopie des gesamten Datenfeldes angelegt. Auf sie kann uneingeschränkt zugegriffen werden.

Datentyp `dictionary`

Der Datentyp `dictionary` repräsentiert ein assoziatives Array, also eine Menge von Schlüssel-Wert-Paaren, wobei die Schlüssel jeweils Zeichenketten und die Werte beliebigen Typs sind.

Durch die Verwendung von Werten des Typs `dictionary` lassen sich beliebig komplexe hierarchische Datenstrukturen aufbauen. Dabei sind die Schlüssel innerhalb einer Hierarchieebene immer eindeutig.

Instanzen des Typs `dictionary` werden mit der vordefinierten Funktion `newdict(sorted=0)` erzeugt. Dabei kann mit dem Argument `sorted` festgelegt werden, ob die Elemente entsprechend der alphanumerischen Abfolge der Schlüssel physisch sortiert (Wert $\neq 0$) oder in der Reihenfolge des Hinzufügens (Wert = 0) belassen werden sollen.

Alternativ kann ein Wert des Typs `dictionary` auch als Literal erzeugt (initialisiert) werden. Hierzu wird folgende Syntax verwendet:

```
dictliteral ::= '{' '[' key ':' literal {',' key ':' literal} ']' '}'
```

Es wird also eine in geschwungene Klammern eingeschlossene kommaseparierte Liste von Schlüssel-Wert-Paaren notiert, wobei Schlüssel und Werte jeweils durch einen Doppelpunkt getrennt sind. Für die Schlüssel können Zeichenketten (in Anführungszeichen) oder gültige Bezeichner (siehe Abschnitt 4.2) angegeben werden. Die Werte selbst sind beliebige Literale, Vektor- und Dictionary-Literale eingeschlossen.

Beispiel:

```
var person = {  
    firstname : "Hugo",  
    lastname  : "Meyer",  
    "years"   : 25  
};
```

Beispiel 12 Erzeugen einer Dictionary-Variablen aus einem Literal

Zulässige Operationen für den Datentyp `dictionary` sind:

- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Zugriff auf einzelne Elemente / Anlegen neuer Elemente mit Hilfe der Operatoren `.` und `[]`

Hinweise:

- Der direkte Zugriff auf Elemente eines (benannten) `dictionary`-Literals mit dem Operator `[]` oder dem Punktoperator ist nicht zulässig.
- Bei der Zuweisung eines `dictionary`-Literals an eine Variable wird eine Kopie des gesamten Datenfeldes angelegt. Auf sie kann uneingeschränkt zugegriffen werden.

Datentyp `buffer`

Der Datentyp `buffer` stellt einen Puffer für unstrukturierte Binärdaten bereit. Variablen dieses Typs dienen als Lese- bzw. Schreibpuffer für binäre Ein-/Ausgabefunktionen (vgl. Abschnitt 11.3) sowie zur Übergabe von Datenbereichen an Systemfunktionen.

Instanzen vom Typ `buffer` werden mit Hilfe der Funktion `newbuffer(size, fillByte=0)` explizit erzeugt (siehe Abschnitt 11.1). Alternativ kann der Funktion `newbuffer` auch ein Wert vom Typ `string` oder `charbuffer` übergeben werden. In diesem Fall wird der Puffer mit einer Kopie des Inhalts des übergebenen Arguments erzeugt.

Darüber hinaus können aus `int`-, `float`- und `double`-Werten mit Hilfe der Konvertierungsfunktion `buffer(arg)` (vgl. Abschnitt 11.2) Puffervariablen erzeugt werden, beispielsweise zur Übergabe als Referenzparameter an Systemfunktionen.

Zulässige Operationen für den Datentyp `buffer` sind:

- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Der Zugriff auf einzelne Elemente (Bytes) erfolgt mit Hilfe des Operators `[]`, wobei die Elemente als vorzeichenlose Integer-Werte behandelt werden.

Datentyp `charbuffer`

Der Datentyp `charbuffer` stellt einen Zeichenpuffer bereit. Die einzelnen Elemente sind, abhängig davon, ob mit ASCII- oder Unicode gearbeitet wird, 8 bzw. 16 Bits breit. Variablen dieses Typs dienen als Puffer für Zeichenkettenmanipulationen sowie zur Übergabe von Datenbereichen an Systemfunktionen.

Instanzen vom Typ `charbuffer` werden mit Hilfe der Funktion `newcbuffer(size, fillChar=0)` explizit erzeugt (siehe Abschnitt 11.1). Alternativ kann der Funktion `newcbuffer` auch ein Wert vom Typ `string`, `buffer` oder `charbuffer` übergeben werden. In diesem Fall wird der Puffer mit einer Kopie des Inhalts des übergebenen Arguments erzeugt.

Darüber hinaus lassen sich Instanzen dieses Typs mit Hilfe der Konstruktorfunktion `CBuf` als Verkettung von Einzelwerten der Typen `string`, `buffer` und `int` erzeugen.

Zulässige Operationen für den Datentyp `charbuffer` sind:

- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Der Zugriff auf einzelne Elemente (Zeichen) erfolgt mit Hilfe des Operators `[]`, wobei die Elemente als Integer-Werte behandelt werden.

Datentyp `function`

Der Datentyp `function` repräsentiert eine innerhalb des Programms aufrufbare Funktion. Intern werden vordefinierte und benutzerdefinierte (d. h. selbst in B++ geschriebene) Funktionen nochmals unterschieden²⁹. Aus Benutzersicht ist diese Unterscheidung nicht notwendig. Eine explizite Instanzenbildung von Funktionen ist nicht möglich. Von einer Funktion existiert stets genau eine Instanz, die für vordefinierte Funktionen bei der Initialisierung von B++ und für benutzerdefinierte Funktionen während der Übersetzung der Funktionsdeklaration (siehe Abschnitt 9) erzeugt wird³⁰.

Zulässige Operationen für den Datentyp `function` sind:

- Zuweisung an eine Variable (vgl. Abschnitt 9.5)
- Aufruf
- Verwendung als Funktionsparameter

²⁹ Deshalb liefert die Funktion `gettypename` (siehe 11.2) auch den Typnamen `CODE` für vordefinierte und `BYTECODE` für benutzerdefinierte Funktionen.

³⁰ Diese Aussage scheint auf den ersten Blick nicht ganz richtig: Wird ein Funktionsliteral, das in seiner Definition mindestens eine statische Variable enthält, an eine Variable zugewiesen, so wird eine neue Instanz der Funktion erzeugt. Auf diese Weise können mehrere Funktionen identischer Implementierung entstehen, die streng genommen jedoch unterschiedliche Funktionen sind, da ihre statischen Variablen unabhängig voneinander sind.

Datentyp `class`

Der Datentyp `class` repräsentiert eine innerhalb des Programms verfügbare Klasse (vgl. 10.1). Klassen sind Vorlagen für konkrete Objekte, deshalb ist eine explizite Instanzenbildung von Klassen nicht möglich. Von einer Klasse existiert stets genau eine Instanz, die während der Übersetzung der Klassendefinition erzeugt wird.

Zulässige Operationen für den Datentyp `class` sind:

- Zuweisung an eine Variable
- Zugriff auf innerhalb der Klasse definierte statische Variablen
- Aufruf in der Klasse definierter statischer Funktionen
- Verwendung als Funktionsparameter in den RTTI-Funktionen (siehe Abschnitt 11.2)
- Verwendung als Vorlage für die Bildung von Objektinstanzen mit dem Operator `new`

Datentyp `object`

Der Datentyp `object` repräsentiert konkrete Instanzen einer Klasse im Sinne von Objekten, deren Struktur entsprechend einer durch eine Klasse gegebenen Vorlage gebildet wird (vgl. 10.3). Objekte werden mittels des Operators `new` aus Vorlagen (Typ `class`) erzeugt.

Zulässige Operationen für den Datentyp `object` sind:

- Aufruf von Member-Funktionen
- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Verwendung in Ausdrücken, sofern entsprechende Operatoren definiert wurden

5.3 Statisch typisierte Variablen

Statisch typisierte Variablen werden durch die explizite Angabe eines Datentyps statt des Schlüsselworts `var` deklariert. Hierfür kann einer der vordefinierten Typbezeichner (vgl. Tabelle 2, Seite 20) oder der Name einer zuvor definierten Klasse verwendet werden.

An so definierte Variablen können nur Werte zugewiesen werden, die vom deklarierten Typ sind. Sofern möglich, erfolgt ggf. eine implizite Typumwandlung. Dabei gelten folgende allgemeine Konvertierungsregeln:

- Alle numerischen Typen sind ineinander konvertierbar. Sofern der Wertebereich des Zieltyps kleiner als der des Quelltyps ist, können dabei u. U. Datenverluste auftreten. Bei der Konvertierung zwischen ganzzahligen und reellwertigen Werten können darüber hinaus Verluste in der Genauigkeit der Zahldarstellung entstehen.
- Der Wert `null` ist zuweisungskompatibel zu allen Referenztypen.
- Einer Variablen vom Typ `FILE` kann der ganzzahlige Wert 0 zugewiesen werden (als Kennzeichen für eine nicht initialisierte Dateivariablen). Generell zuweisungskompatibel sind Werte des Typs `intptr`.
- Die Typen `string` und `charbuffer` sind wechselseitig implizit ineinander konvertierbar; bei der Konvertierung wird eine Kopie des Wertes angelegt.
- An Variablen des Typs `object` können beliebige Objekte (Instanzen von Klassen) zugewiesen werden.

- An Variablen mit definiertem Klassentyp können Instanzen (Objekte) der jeweiligen Klasse oder davon abgeleiteter Klassen zugewiesen werden.
- Klassen können zusätzlich Konvertierungsoperatoren definieren, die die implizite Umwandlung ihrer Instanzen in verschiedene Datentypen erlauben – siehe Abschnitt 10.10.9.

Darüber hinaus gibt es zur Umwandlung von Werten in die vordefinierten Datentypen Standardfunktionen zur expliziten Typkonvertierung. Sie sind in Abschnitt 11.2 beschrieben.

Einen Sonderfall stellen Variablen dar, die mit dem Schlüsselwort `auto` deklariert werden. Solche Variablen haben ebenfalls einen festen (statischen) Datentyp, der jedoch – wie in C++11³¹ – bei ihrer ersten Initialisierung auf den Typ des zugewiesenen Werts festgelegt wird. Deshalb müssen sie unmittelbar bei der Definition initialisiert werden.

Hinweis:

Die Überprüfung der Kompatibilität bei Zuweisungen erfolgt zur Laufzeit, sofern der Typ des zugewiesenen Wertes nicht schon zur Übersetzungszeit feststeht, was bei Literalen der Fall ist. Demzufolge führen Verletzungen der Typkompatibilität zu Laufzeitfehlern, die als Ausnahmen des Typs `string` signalisiert werden und vom Programm behandelt werden können – siehe Abschnitt 8.4).

5.4 Initialisierung von Variablen

Die Initialisierung von globalen Variablen, Member-Variablen in einer Klassendefinition (vgl. Abschnitt 10.1) sowie statischen Variablen in Funktionen (9.4) und Vorgabewerten für Funktionsparameter (vgl. 9.3) erfolgt zur Übersetzungszeit. Somit muss der Initialwert ein Literal (oder das Ergebnis eines literalen Ausdrucks) sein.

Lokale (temporäre) Variablen innerhalb einer Funktion werden zur Laufzeit initialisiert. Deshalb kann hier das Ergebnis eines beliebigen Ausdrucks zur Initialisierung verwendet werden.

Bei der Initialisierung statisch typisierter Variablen erfolgt während der Initialisierung eine Prüfung der Typkompatibilität des zugewiesenen Wertes.

Die Initialisierung von Variablen zur Übersetzungszeit erfolgt mit Hilfe des Zuweisungsoperators (=) in der Form

```
[type] varname = initexpr
```

Für statisch typisierte Variablen, die zur Laufzeit initialisiert werden, kann – analog zu C++ – außerdem (alternativ) die 'Konstruktorschreibweise', also die Form

```
type varname(initexpr)
```

verwendet werden.

Werden Variablen der eingebauten Standardtypen auf diese Weise initialisiert, erfolgt dabei ein impliziter Aufruf der Konvertierungsfunktion in den jeweiligen Zieltyp (vgl. Abschnitt 11.2). Bei der Initialisierung von Objekten (Variablen von einem konkreten Klassentyp) wird eine Instanz der entsprechenden Klasse erzeugt und deren Konstruktor aufgerufen.

Hinweise:

Eine Variable des Typs `FILE` kann ebenfalls in 'Konstruktorschreibweise' initialisiert werden. Dies ist identisch mit einer Initialisierung durch Zuweisung mit einem Aufruf der Funktion `fopen` (vgl. 11.3). Eine Definition

```
FILE fp(filename,mode)
```

entspricht also genau der Form

```
FILE fp = fopen(filename,mode)
```

³¹Das Verhalten ist nicht ganz identisch mit dem von C++-11. In B++ wird der Typ von lokalen (temporären) `auto`-Variablen in Funktionen erst zur Laufzeit bestimmt.

Bei beiden Varianten muss (bzw. sollte) die so geöffnete Datei explizit mit `fclose` wieder geschlossen werden.

Eine weitere Besonderheit ist bei der Definition statisch typisierter Variablen mit dem Pseudotyp `bool` zu beachten: Hier wird im Grunde einfach eine `int`-Variable definiert. Benutzt man allerdings zur Initialisierung die Konstruktorschreibweise, wird implizit die Typkonvertierungsfunktion `bool(x)` aufgerufen, die stets den Wert 1 (`true`) oder 0 (`false`) zurückgibt – siehe folgendes Beispiel:

```
void main()
{
    bool b1 = 17;
    bool b2(17);
    print(b1, " ", b2, "\n"); // prints 17 1
}
```

Beispiel 13 Initialisierung von `bool`-Variablen

Implizite Initialisierung

Wird eine Variable bei ihrer Definition nicht explizit initialisiert, so erhält sie implizit einen Standardwert. Dabei gelten folgende allgemeine Regeln:

- Eine dynamisch typisierte Variable (Typ `var`) erhält den Anfangswert `null`.
- Variablen der numerischen Typen werden mit dem Zahlenwert `Null` initialisiert.
- Variablen der Referenztypen haben generell den Initialwert `null`, sofern es sich nicht um lokale (temporäre) Variablen in Funktionen handelt.
- Temporäre Variablen der Containertypen (`buffer`, `charbuffer`, `vector`, `dictionary`) werden mit einem leeren Container initialisiert.
- Temporäre Variablen des Typs `string` werden als leere Zeichenkette initialisiert.
- Für temporäre streng typisierte Objektvariablen wird eine Objektinstanz erzeugt, wobei der jeweilige Konstruktor ohne Argumente aufgerufen wird (siehe Abschnitt 10.3). Dies setzt voraus, dass die jeweilige Klasse einen Konstruktor bereitstellt, der ohne Parameter aufgerufen werden kann.

5.5 Implizite und explizite Typkonvertierungen

Immer dann, wenn in einem Ausdruck Daten unterschiedlicher Typen verwendet werden, sind geeignete Datenkonvertierungen (bzw. Typkonvertierungen) erforderlich. Die in B++ für solche Konvertierungen geltenden Regeln werden nachfolgend zusammengestellt.

Innerhalb rein numerischer Ausdrücke erfolgt eine implizite Umwandlung der beteiligten Operanden in den Typ mit dem größten positiven Wertebereich, mindestens jedoch in den Typ `int`. Die hierbei geltenden Regeln entsprechen im Wesentlichen denen von C/C++:

- Werte der Typen `sbyte`, `byte`, `short` und `ushort` werden generell nach `int` konvertiert.
- Für die Größe der numerischen Typen gilt
`int < uint < long < ulong < float < double`.
- Der Typ `intptr` ist in seiner Größe plattformabhängig. Auf 64-Bit-Plattformen entspricht er dem Typ `long`, sonst dem Typ `int`.

Für nicht numerische Werte wird in Ausdrücken keine implizite Typumwandlung vorgenommen. Hier ist es Aufgabe des Programmierers, eine explizite Typumwandlung vorzunehmen und/oder geeignete Operatormethoden für Objekte bereitzustellen (siehe Abschnitt 10.10.9).

Die implizite Typumwandlung in numerischen Ausdrücken findet streng genommen nur für Operatoren mit zwei Operanden (Infix-Formen) statt.

Explizite Typumwandlungen erfolgen in der Form

typename(*expr*),

wobei *typename* der Zieltyp und *expr* ein Ausdruck ist, dessen Ergebnis in den Zieltyp umgewandelt werden soll.

Für die explizite Konvertierung in die eingebauten Datentypen stehen vordefinierte Funktionen zur Verfügung, die in Abschnitt 11.2 beschrieben sind.

Für benutzerdefinierte Objekttypen (Klassen) können Operatoren zur Typkonvertierung als Elementfunktionen definiert werden. Darüber hinaus besteht die Möglichkeit, explizite Konvertierungen in Objekttypen als globale Funktionen zu definieren – vgl. Abschnitt 10.10.9.

Implizite Typumwandlungen werden auch bei der Zuweisung von Werten an Variablen bzw. an Argumente von Funktionsaufrufen vorgenommen, sofern das Zuweisungsziel (die Variable oder das Argument) statisch typisiert ist. In diesem Fall werden die Funktionen zur expliziten Typumwandlung bzw. die entsprechenden Operatoren von Objekten automatisch aufgerufen.

5.6 Vordefinierte Variablen

In B++ existieren – wie in C – drei vordefinierte globale Variablen des `FILE`-Typs, die auf die Standardein- und -ausgabedateien des Systems verweisen:

- **`stdin`**: Standard-Eingabe
- **`stdout`**: Standard-Ausgabe
- **`stderr`**: Standard-Fehlerausgabe

Diese Standarddateien können *nicht* mit `fopen` geöffnet bzw. mit `fclose` geschlossen werden (vgl. Abschnitt 10.10).

Darüber hinaus gibt es folgende vordefinierte Variablen:

- **`__OSTYPE__`**: Bezeichnung des Betriebssystems (`string`), auf dem die B++-Laufzeitumgebung ausgeführt wird; mögliche Werte sind derzeit „WIN32“ für die Windows-Desktop-Version, „WINCE“ für Windows-CE, „DOS“ für die MS-DOS-Systeme, „LINUX“ für Linux-Systeme bzw. „UNIX“ für andere UNIX-Systeme.
- **`__UNICODE__`**: Buildvariante der Laufzeitumgebung (`int`); die Variable hat in der Unicode-Variante den Wert 1, sonst 0.

Mit Hilfe dieser beiden Variablen kann ein B++-Programm zur Laufzeit sein Verhalten an Besonderheiten der Umgebung, in der es ausgeführt wird, anpassen – im Unterschied zu den vordefinierten Literalen (vgl. 4.3.8), die zur Übersetzungszeit ausgewertet werden. Dies ist deshalb von Bedeutung, weil der vom Compiler erzeugte Bytecode unabhängig von der Plattform ist, auf der er später ausgeführt wird – plattformspezifisch ist lediglich die Laufzeitumgebung. Allerdings kann es sein, dass auf unterschiedlichen Zielplattformen verschiedene systemspezifische Funktionen zur Verfügung stehen bzw. bestimmte Systemeigenschaften berücksichtigt werden müssen.

```

if (__OSTYPE__ == "WINCE")
    Window(0,240,320,
           WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN,WS_EX_OVERLAPPEDWINDOW|
           WS_EX_CONTROLPARENT|WS_MAXIMIZE,title,_mainMenu);
else
    Window(0,640,480,
           WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN,WS_EX_OVERLAPPEDWINDOW|
           WS_EX_CONTROLPARENT,title,_mainMenu);

```

Beispiel 14 Verwendung der globalen Variablen `__OSTYPE__`

Beispiel 14 zeigt einen Ausschnitt aus der Initialisierung des Hauptfensters einer mit B++ implementierten Windows-Anwendung. Hier wird in Abhängigkeit davon, ob die Anwendung unter Windows-CE ausgeführt wird oder nicht, das Hauptfenster mit unterschiedlicher Größe und unterschiedlichen Stilen initialisiert.

6 Operatoren

Das Konzept der Operatoren in B++ entspricht weitgehend dem von C bzw. C++. Dieses Kapitel beschränkt sich daher auf eine kurze Referenz der in B++ verfügbaren Operatoren und vorhandener Unterschiede zu C++. Für Klassen (bzw. Objekte) können einige der Operatoren neu- bzw. undefiniert werden – siehe hierzu Abschnitt 10.10.

6.1 Arithmetische Operatoren

Die arithmetischen Operatoren realisieren in Verbindung mit numerischen Operanden die vier Grundrechenarten. Der Additionsoperator ist darüber hinaus auch auf Zeichenketten (Datentyp `string`) anwendbar. In diesem Fall dient er als *Verkettungsoperator*.

B++ kennt folgende arithmetischen Operatoren:

Operator	Bedeutung	Beispiel
<code>+</code> (unär)	positives Vorzeichen	<code>+a</code>
<code>-</code> (unär)	negatives Vorzeichen	<code>-a</code>
<code>*</code>	Multiplikation	<code>a * b</code>
<code>/</code>	Division	<code>a / b</code>
<code>%</code>	Divisionsrest (Modulo)	<code>a % b</code>
<code>+</code>	Addition	<code>a + b</code>
<code>-</code>	Subtraktion	<code>a - b</code>

Tabelle 5 Arithmetische Operatoren in B++

Hinweise:

- Das Verhalten der Operatoren hängt von den Datentypen der Operanden ab. Haben beide Operanden integrale Datentypen so wird eine ganzzahlige Operation durchgeführt, hat mindestens ein Operand den Typ `float` oder `double`, eine Gleitkomma-Operation.
- Haben die Operanden unterschiedliche Datentypen, so erfolgt vor Ausführung der Operation eine Umwandlung in den Typ mit dem größeren Wertebereich.
- Das Ergebnis von Gleitkomma-Operationen ist vom Typ `float`, wenn keiner der Operanden vom Typ `double` ist, sonst vom Typ `double`. Das Resultat ganzzahliger Operationen ist mindestens von Typ `int`.
- Ist ein Operand des Additionsoperators (`+`) vom Typ `string`, so wirkt der Operator als Verkettungsoperator. Der andere Operand muss dann entweder auch vom Typ `string` oder von einem integralen Typ sein. Im letztgenannten Fall wird das niederwertigste Byte (bzw. Wort in UNICODE-Builds) des numerischen Operanden als Code eines einzelnen Zeichens interpretiert.
- Die Vorzeichenoperatoren verhalten sich wie die in C/C++. Insbesondere heißt dies, dass sie einen integralen Operanden mit einem geringeren Wertebereich in einen Wert vom Typ `int` umwandeln. Das unäre Minus kehrt außerdem das Vorzeichen numerischer Werte um.
- Alle arithmetischen Operatoren können für Objekttypen (Klassen) als Elementfunktionen definiert werden – siehe Abschnitt 10.10.8.

6.2 Vergleichsoperatoren

Die Vergleichsoperatoren – auch als relationale Operatoren bezeichnet – dienen dem Größenvergleich ihrer (stets zwei) Operanden.

B++ kennt folgende Vergleichsoperatoren:

Operator	Bedeutung	Beispiel
<code>==</code>	gleich	<code>a == b</code>
<code>!=</code>	ungleich	<code>a != b</code>
<code>></code>	größer als	<code>a > b</code>
<code>>=</code>	größer oder gleich	<code>a >= b</code>
<code><</code>	kleiner als	<code>a < b</code>
<code><=</code>	kleiner oder gleich	<code>a <= b</code>

Tabelle 6 Vergleichsoperatoren in B++

Hinweise:

- Kommen als Operanden numerische Werte unterschiedlicher Typen vor, so erfolgt zunächst implizit eine Konvertierung in denjenigen Typ mit dem größten Wertebereich. Dies kann wegen bei der Typumwandlung auftretender numerischer Ungenauigkeiten zu Problemen, insbesondere bei den Operatoren `==` und `!=` führen.
- Ist einer der Operanden numerisch, so muss auch der andere Operand numerisch sein³².
- Zeichenketten (`string`) können mit Zeichenketten oder mit `null` verglichen werden.
- Werte aller anderen Referenztypen können mit Werten desselben Typs (einschließlich Weak Referenzen des entsprechenden Typs) oder mit `null` verglichen werden. Im Einzelnen gilt dabei:
 - Für die Typen `buffer` und `charbuffer` und `vector` erfolgt ein elementweiser Vergleich.
 - Für die übrigen Referenztypen werden die Adressen der Datenbereiche verglichen³³, sofern es sich nicht um Objekttypen (Klassen) handelt, die die Methoden `compare` bzw. `equals` implementieren und dadurch ein verändertes Verhalten der Vergleichsoperatoren definieren (vgl. Abschnitt 10.10.7)
- Werte des Typs `null` sind kleiner als alle anderen Werte.

³² Ein Vergleich eines numerischen Wertes mit einem Wert vom Objekttyp ist ebenfalls möglich, sofern das betreffende Objekt eine Konvertierung in den entsprechenden numerischen Typ oder eine geeignete Implementierung der Methoden `equals` bzw. `compare` bereitstellt.

³³ Deshalb ist hier i. a. nur die Verwendung der Operatoren `==` und `!=` sinnvoll.

6.3 Logische Operatoren

Die logischen Operatoren verknüpfen Wahrheitswerte miteinander. B++ besitzt keinen eigenen Datentyp für Wahrheitswerte, stattdessen gelten folgende Regeln:

- Werte vom Typ `null` haben den Wahrheitswert `false`.
- Numerische Werte haben den Wahrheitswert `false`, wenn ihr Betrag 0 ist, sonst `true`.
- Alle anderen Werte haben den Wahrheitswert `true`.³⁴

Folgende logischen Operatoren sind in B++ definiert:

Operator	Bedeutung	Beispiel
!	NOT	!a
&&	AND	a && b
	OR	a b

Tabelle 7 Logische Operatoren in B++

Hinweise:

- Der Operator `!` liefert als Resultat 1 für `true` und 0 für `false`.
- Ausdrücke mit den binären Operatoren `&&` bzw. `||` werden von links nach rechts ausgewertet. Die Auswertung bricht ab, sobald das Ergebnis des Ausdrucks feststeht. Der Rückgabewert ist dann derjenige Operand, der zum Abbruch der Auswertung führte. Deshalb dürfen die Ergebnisse der Operatoren *nicht* implizit als *Integer*-Werte angesehen werden. Mit Hilfe der Konvertierungsfunktion `bool` können sie explizit in die Wahrheitswerte `true` bzw. `false` konvertiert werden.
- Der NOT-Operator kann für Objekttypen (Klassen) als Elementfunktion definiert werden – siehe Abschnitt 10.10.6.

6.4 Bitoperatoren

Die Bitoperatoren sind nur auf Operanden integraler Typen (siehe Abschnitt 5.2.1) direkt anwendbar. Sie realisieren Operationen auf den einzelnen Bits der Operanden. Darüber hinaus können sie für Objekttypen (Klassen) definiert werden – siehe Abschnitt 10.10.8.

Folgende Bitoperatoren sind in B++ definiert:

Operator	Bedeutung	Beispiel
~	bitweises NOT	~a
&	bitweises AND	a & b
	bitweises OR	a b
^	bitweises XOR	a ^ b
<<	bitweise Linksverschiebung	a << 3
>>	bitweise Rechtsverschiebung	a >>3

Tabelle 8 Bitoperatoren in B++

³⁴ Achtung: Es gibt einen Unterschied zwischen dem Wahrheitswert `true` und dem vordefinierten Literal `true`. Das Literal ist einfach ein Alias für den numerischen Wert 1.

Der Wert des rechten Operanden der Verschiebeoperatoren darf nicht negativ sein und sollte die Anzahl der Bits des linken Operanden nicht überschreiten.

Das Verhalten des Operators `>>` hängt - wie in C/C++ - vom Typ des linken Operanden ab. Ist der linke Operand vorzeichenbehaftet, bleibt das Vorzeichen erhalten (bei negativen Werten wird das höchstwertige Bit auf 1 gesetzt). Bei vorzeichenlosem linken Operanden wird das höchstwertige Bit stets auf 0 gesetzt.

Haben die Operanden der Operatoren `&`, `|` bzw. `^` unterschiedliche Typen, wird vor der Ausführung der Operation eine Konvertierung in den betragsmäßig größeren Typ vorgenommen – auch hier gelten die Regeln von C/C++.

6.5 Zuweisungsoperatoren

Die Zuweisungsoperatoren dienen der Zuweisung eines Wertes an eine Variable. Wie in C/C++ wird die Zuweisung mit dem Operator `=` realisiert, der mit den zweiwertigen arithmetischen bzw. Bitoperatoren kombiniert werden kann. Nachfolgende Tabelle zeigt die Möglichkeiten:

Operator	Bedeutung	Beispiel
<code>=</code>	einfache Zuweisung	<code>a = b</code>
<code>+=</code>	Zuweisung mit Addition	<code>a += b</code>
<code>-=</code>	Zuweisung mit Subtraktion	<code>a -= b</code>
<code>*=</code>	Zuweisung mit Multiplikation	<code>a *= b</code>
<code>/=</code>	Zuweisung mit Division	<code>a /= b</code>
<code>%=</code>	Zuweisung mit Divisionsrest	<code>a %= b</code>
<code> =</code>	Zuweisung mit bitweisem OR	<code>a = b</code>
<code>&=</code>	Zuweisung mit bitweisem AND	<code>a &= b</code>
<code>^=</code>	Zuweisung mit bitweisem XOR	<code>a ^= b</code>
<code><<=</code>	Zuweisung mit Linksverschiebung	<code>a <<= 2</code>
<code>>>=</code>	Zuweisung mit Rechtsverschiebung	<code>a >>= 2</code>

Tabelle 9 Zuweisungsoperatoren in B++

Hinweise:

- Der einfache Zuweisungsoperator ist mit allen Datentypen verwendbar. Dabei ist zu beachten, dass bei der Zuweisung von Werten der Referenztypen keine Kopie des Wertes, sondern lediglich ein Verweis darauf zugewiesen wird.
Bei Verwendung der statischen Typisierung ist eine Zuweisung nur dann möglich, wenn der zuzuweisende Wert denselben Datentyp wie das Ziel der Zuweisung hat oder implizit in diesen Typ konvertiert werden kann – siehe auch 5.3.
- Die zusammengesetzten Zuweisungsoperatoren sind Kombinationen aus dem einfachen Zuweisungsoperator und einem arithmetischen oder Bitoperator (vgl. 6.1), wobei das Zuweisungsziel als linker Operand der jeweiligen Operation dient.
So ist etwa der Ausdruck `a += b` identisch mit `a = a + b`.
- Für die Operanden der zusammengesetzten Zuweisungsoperatoren gelten die gleichen Regeln wie für die der entsprechenden einfachen Operatoren.

6.6 Inkrement- und Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren sind unär. Sie erhöhen (Operator `++`) bzw. erniedrigen (Operator `--`) ihren Operanden jeweils um den Wert 1. Der Operand muss dabei eine mit einem integralen Typ oder einem Objekttyp initialisierte Variable sein³⁵.

Beide Operatoren können in Präfix- oder Postfix-Notation verwendet werden. Ein Unterschied zwischen diesen Notationen besteht bei der Verwendung innerhalb von Ausdrücken. Bei der Präfix-Notation ist der Wert des Teilausdrucks gleich dem Wert des Operanden nach dem Inkrement/Dekrement während bei der Postfix-Notation der Wert des Operanden vor dem Inkrement/Dekrement zurückgegeben wird.

Nachfolgende Tabelle soll die Varianten veranschaulichen:

Operator	Bedeutung	Beispiel	Entsprechung
<code>++x</code>	Inkrement (Präfix)	<code>b = ++a</code>	<code>a = a+1, b = a</code>
<code>x++</code>	Inkrement (Postfix)	<code>b = a++</code>	<code>b = a, a = a+1</code>
<code>--x</code>	Dekrement (Präfix)	<code>b = --a * 7</code>	<code>a = a-1, b = a * 7</code>
<code>x--</code>	Dekrement (Postfix)	<code>b = a-- * 7</code>	<code>b = a * 7, a = a-1</code>

Tabelle 10 Inkrement und Dekrement

Hinweise:

- Die Inkrement- bzw. Dekrement-Operatoren erzeugen kürzeren und schnelleren Bytecode als ihre „ausgeschriebenen“ Entsprechungen (siehe Tabelle 10). Sie sind deshalb besonders für die Modifikation von Zählvariablen in Schleifen (vgl. Abschnitt 8) geeignet³⁶.
- In zusammengesetzten Ausdrücken sollten diese Operatoren mit Vorsicht und nur dann verwendet werden, wenn die etwas höhere Effizienz wirklich notwendig ist, da neben der schlechteren Lesbarkeit u. U. unerwünschte Seiteneffekte auftreten können. So wird beispielsweise in einem Ausdruck `c = a || (--b == 1)` die Variable `b` niemals dekrementiert, solange `a true` ergibt.
- Die Operatoren können auch für Objekttypen (Klassen) definiert werden – siehe Abschnitt 10.10.4.

³⁵ Implementierungsbedingt können die Operatoren nicht direkt auf den Rückgabewert einer Funktion angewendet werden, auch wenn es sich dabei um ein Objekt handelt, das geeignete Operatorfunktionen bereitstellt.

³⁶ Der erzeugte Bytecode der Präfix-Notation ist kürzer und damit schneller als der der Postfix-Notation. Deshalb sollte die Präfix-Notation bevorzugt werden, wenn das spezielle Verhalten der Postfix-Variante nicht erforderlich ist.

6.7 Sonstige Operatoren

Indexoperator

Der Indexoperator wird durch eckige Klammern `[]` ausgedrückt. Er dient dem indexbasierten Zugriff auf einzelne Elemente eines Containers (`dictionary`, `vector`, `buffer`, `charbuffer`) oder einer Zeichenkette.

Beispiel:

```
/* operator [] example */
main()
{
    var vec = T(1,2,3,4);          // create a vector
    vec[1] = 7;                    // assign 7 to second element of vec
    print(vec[1], "\n");           // prints 7 to console
}
```

Beispiel 15 Verwenden des Index-Operators

Für die Verwendung mit einem Dictionary muss der Operand in der Klammer vom Typ `string` sein (Schlüssel), in allen anderen Fällen von einem integralen Typ (Index).

Für Objekttypen kann der Operator ebenfalls definiert werden (siehe 10.10.2), hier hängt der Typ des Index-Operanden von der jeweiligen Implementierung ab.

Punktoperator (Property-Operator)

Der Punkt wird in vielen Programmiersprachen zur Referenzierung von Elementen strukturierter Datentypen (Strukturen, Objekte) verwendet. In B++ ist er ein Operator, der für den Typ `dictionary` fest vordefiniert ist und dort in seiner Semantik der des Indexoperators entspricht. Für Objekte dient er dem Zugriff auf nicht statische Datenelemente. Darüber hinaus kann der Punktoperator für Objekte bzw. deren Klassen umdefiniert werden, z. B. um zusätzliche Pseudo-Eigenschaften (*Properties*) zu definieren (siehe Abschnitt 10.10.3).

Der Operator hat die allgemeine Form

variable . identifier

wobei `variable` entweder vom Typ `dictionary` oder ein Objekt sein muss.

Bei Variablen des Typs `dictionary` ist `identifier` der Bezeichner (Schlüssel) eines Eintrags.

Beispiel:

```
// operator . example
main()
{
    auto dict = { a: 1, b: 2 }; // define a dictionary
    dict.c = dict.a + dict.b;   // add new value c to dictionary
    print(dict.c);              // print value
}
```

Beispiel 16 Verwenden des Punktoperators

Klammern

Runde Klammern zählen ebenfalls zu den Operatoren. Sie werden in zwei Zusammenhängen verwendet:

- als Begrenzung der Argumentliste von Funktionen bei Deklaration und Aufruf
`myFunc(a, b, c)`

- zur Steuerung der Auswertungsreihenfolge von Ausdrücken – „Klammern rechnen wir zuerst aus!“

$x = (a + b) * c$

Klassen können den Operator als Elementfunktion definieren (vgl. 10.10.1). Instanzen dieser Klassen lassen sich dann wie Funktionen aufrufen.

Bedingungsoperator

Der Bedingungsoperator hat die allgemeine Form

$condexpr ? expr1 : expr2.$

Dabei wird zunächst der Ausdruck *condexpr* (eine Bedingung) ausgewertet. Sein Resultat wird als Wahrheitswert interpretiert. Ist das Ergebnis *true*, so wird anschließend *expr1*, sonst *expr2* ausgewertet. Das jeweilige Resultat ist das Ergebnis des gesamten Ausdrucks.

Beispiel:

```
/* get minimum */
var min(var a, var b)
{
    return a < b ? a : b;
}
```

Beispiel 17 Verwenden des Bedingungsoperators

Hinweis:

Der Typ des Rückgabewerts der Funktion in obigem Beispiel ist nicht unbedingt bekannt. Ist beispielsweise *a* vom Typ *int* und *b* vom Typ *float*, so wird der Bedingungsausdruck (*a*<*b*) temporär nach *float* konvertiert. Abhängig vom Ergebnis ist der Rückgabewert der Funktion vom Typ *int* oder *float*.

Komma-Operator

Der Komma-Operator gestattet die Verkettung von mehreren Ausdrücken zu einer Ausdrucksliste. Anders ausgedrückt: Mit Hilfe dieses Operators lassen sich mehrere Ausdrücke dort notieren, wo syntaktisch nur einer vorgesehen ist. Dabei ist das Ergebnis des Gesamtausdrucks (also der Liste) das des letzten Teilausdrucks.

Hinweis:

Der *Komma-Operator* ist nicht zu verwechseln mit dem *Trennzeichen Komma*, das in Parameter- oder Variablenlisten verwendet wird.

Operatoren `new`, `delete`, `::` und `->`

Diese Operatoren haben nur im Zusammenhang mit Klassen und Objekten bzw. bei der expliziten Speicherverwaltung (Operator `delete`) Bedeutung. Sie werden in den Abschnitten 10.3 bis 10.5 sowie in Kapitel 13 ausführlicher erläutert und sind in nachfolgender Tabelle nur der Vollständigkeit halber aufgeführt.

Operator	Bedeutung	Beispiel
<code>new</code>	Objektinstanz aus Klasse erzeugen	<code>a = new A()</code>
<code>delete</code>	Löschen einer Objektinstanz	<code>delete a</code>
<code>::</code>	Definition von Methoden, Aufruf von Klassenmethoden	<code>A::A() { ... }</code> <code>i = A::getMaxId()</code>
<code>-></code>	Aufruf von Instanzmethoden	<code>name = a->getName()</code>

Tabelle 11 Objektbezogene Operatoren in B++

Unäre Referenzoperatoren `&` und `*`

Die beiden Operatoren sind auf Referenztypen anwendbar und erlauben die Konvertierung eines Wertes in eine *Weak-Referenz* (Operator `&`) bzw. die Umwandlung einer Weak-Referenz in einen Wert (Operator `*`).

Eine Weak-Referenz ist ein Verweis auf ein Datenobjekt von einem Referenztyp, das explizit von der Referenzzählung und damit von der automatischen Speicherverwaltung ausgenommen wird. Ihr Zweck besteht hauptsächlich darin, Speicherlecks zu verhindern, die aufgrund zyklischer Referenzen in Datenstrukturen entstehen können. Darüber hinaus lässt sich mit ihnen eine explizite Speicherverwaltung realisieren. Eine ausführlichere Beschreibung der Speicherverwaltung erfolgt in Kapitel 13.

Hinweise:

- Weak-Referenzen können nicht an statisch typisierte Variablen zugewiesen werden.
- Der unäre Dereferenzierungsoperator `*` kann außerdem explizit für Objekte (bzw. deren Klassen) definiert werden (siehe Abschnitt 10.10.5). Die entsprechende Operatorfunktion wird dann aufgerufen, wenn der Operator mit einem „echten“ Objekt, also nicht mit einer Weak-Referenz, verwendet wird.

6.8 Hierarchie der Operatoren

Werden mehrere Operatoren innerhalb eines Ausdrucks verwendet, so ergibt sich die Frage nach der Reihenfolge ihrer Auswertung. Um eine solche Reihenfolge festlegen zu können, wird eine Rangfolge (Hierarchie) der Operatoren definiert. B++ hält sich dabei weitgehend an die Regeln von C/C++. In Tabelle 12 ist die Hierarchie der Operatoren in B++ aufgeführt. Dabei werden Operatoren höherer Priorität vor jenen niederer Priorität ausgewertet. Bei Operatoren der gleichen Prioritätsstufe erfolgt die Auswertung von links nach rechts entsprechend der Notation.

Priorität	Kategorie	Operatoren
0	Komma	,
1	Zuweisung	= += -= *= /= %= =, &=, ~=, ^=, <<= >>=
2	Entscheidung	? :
3	OR (logisch)	
4	AND (logisch)	&&
5	OR (bitweise)	
6	XOR (bitweise)	^
7	AND (bitweise)	&
8	Gleichheit	== !=
9	Relation	< <= >= >
10	Shift	<< >>
11	Addition	+ -
12	Multiplikation	* / %
13	Präfix	+ - ! ~ & * ++ -- new delete
14	Postfix	() [] . ++ -- ->

Tabelle 12 Operatorklassifizierung und -hierarchie in B++

7 Schlüsselwörter und reservierte Bezeichner

Schlüsselwörter sind im syntaktischen Sinne Terminalsymbole. Das bedeutet, dass sie bereits bei der syntaktischen Analyse identifiziert und speziellen internen Symbolen zugeordnet werden. Innerhalb eines Programms ist den Schlüsselwörtern eine feste (zentrale) Bedeutung zugeordnet. Deshalb können keine benutzerdefinierten Bezeichner erzeugt werden, die den Schlüsselwörtern entsprechen.

Die meisten Schlüsselwörter in B++ sind C bzw. C++ entlehnt und besitzen eine analoge Bedeutung. Nachfolgende Tabelle gibt eine Übersicht.

Schlüsselwort	Kategorie	Bedeutung
null	Typkonstante	Bezeichner für den Datentyp <i>null</i> und dessen einziger Wert
void	Typkonstante	Kennzeichnung von Funktionen ohne Rückgabewert (an Stelle eines Rückgabetyps)
do	Steuerung	Konstruktion von Wiederholungsanweisungen
while	Steuerung	Konstruktion von Wiederholungsanweisungen
for	Steuerung	Konstruktion von Wiederholungsanweisungen
break	Steuerung	Abbruch einer Wiederholungs- oder Selektionsanweisung
continue	Steuerung	vorzeitiger Sprung zum Anfang einer Wiederholungsanweisung
if	Steuerung	Konstruktion bedingter Anweisungen
else	Steuerung	Konstruktion bedingter Anweisungen
switch	Steuerung	Anfangskennzeichen einer Selektionsanweisung (Fallunterscheidung)
case	Steuerung	Kennzeichnung eines Auswahlwertes in einer Selektionsanweisung
default	Steuerung	Kennzeichen des Standardzweiges einer Selektionsanweisung
return	Steuerung	(vorzeitige) Rückkehr aus einer Funktion
try	Ausnahmen	Einleitung eines geschützten Blocks
catch	Ausnahmen	Beginn der Ausnahmebehandlung
throw	Ausnahmen	Auslösen einer Ausnahme
var	Deklaration	Deklaration einer Variablen dynamischen (beliebigen) Typs
auto	Deklaration	Deklaration einer statisch typisierten Variablen mit Typfestlegung durch Initialisierung
function	Deklaration	Deklaration eines Funktionsliterals
class	Deklaration	Beginn einer Klassendeklaration
static	Deklaration	Deklaration eines Klassen-Members oder einer statischen lokalen Variable
public	Deklaration	Einleitung eines Abschnitts mit öffentlichen Elementen in einer Klassendeklaration

protected	Deklaration	Einleitung eines Abschnitts mit geschützten Elementen in einer Klassendeklaration
private	Deklaration	Einleitung eines Abschnitts mit privaten Elementen in einer Klassendeklaration
new	Speicherverwaltung	Erzeugen einer Objektinstanz
delete	Speicherverwaltung	Zerstören einer Objektinstanz oder eines Datums von einem anderen Referenztyp (Weak-Referenz)
operator	Deklaration	Deklaration einer Operatorfunktion
TRON	Debugging	Einschalten Trace-Modus
TROFF	Debugging	Ausschalten Trace-Modus
TRSTEP	Debugging	Einschalten Einzelschrittmodus

Tabelle 13 Schlüsselwörter in B++

Die Schlüsselwörter **TRON**, **TROFF** und **TRSTEP**, die im Quelltext als separate Anweisungen notiert werden, unterstützen ein elementares Debugging auf der Ebene des vom Compiler generierten Bytecodes.

Im Trace-Modus, der mit **TRON** ein- und mit **TROFF** ausgeschaltet wird, erfolgt eine textuelle Ausgabe des Programmablaufs auf Bytecode-Ebene, z. B. über eine Konsole bei Verwendung der Kommandozeilenversion von B++ oder ein spezielles Ausgabefenster einer GUI (vgl. Abbildung 2).

TRSTEP schaltet den Trace-Modus ebenfalls ein, bewirkt aber zusätzlich einen Halt des Programms nach der Ausführung jedes Einzelbefehls. Der Einzelschrittbetrieb wird beim Erreichen einer **TROFF**-Anweisung oder durch eine spezielle Tastatureingabe (ESC in der Kommandozeilenversion) wieder verlassen.

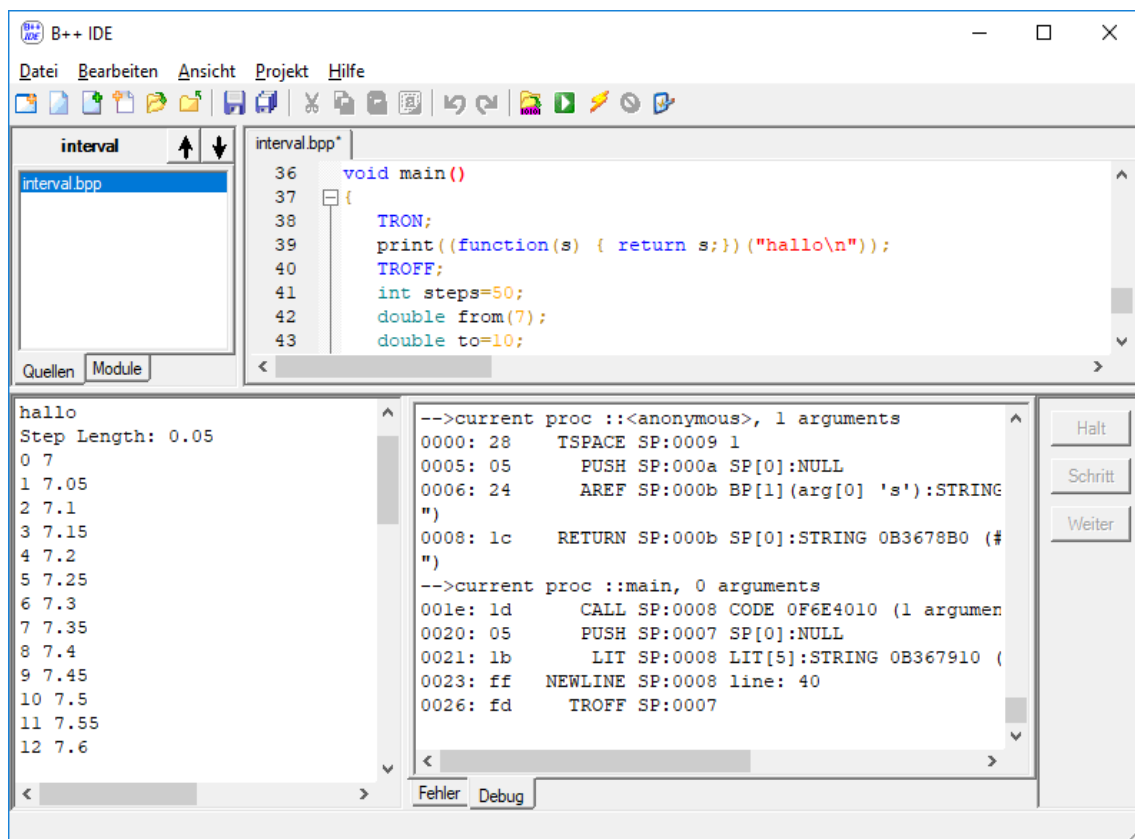


Abbildung 2 Beispiel einer Trace-Ausgabe in der B++-IDE

Die Trace-Ausgabe stellt die Programmadressen, die ausgeführten Bytecodes, die Position des Stackzeigers sowie Typen und Werte der jeweils betroffenen Operanden dar.

Im Zusammenhang mit der objektorientierten Programmierung kennt B++ noch folgende **reservierte Bezeichner** (mitunter auch "*magic names*" genannt):

- **this** Verweis auf das aktuelle Objekt in Member-Funktionen
- **self** Verweis auf die aktuelle Funktion (Methode)
- **dtor** Alias-Bezeichner für den Destruktor einer Klasse (siehe 10.3)
- **defined** Pseudo-Operator zum Test, ob ein benanntes Literal definiert ist (ab B++ 1.2)
- **OP_CALL, OP_VREF, OP_VSET, OP_PREF, OP_PSET, OP_NEG, OP_PLUS, OP_INC, OP_PIC, OP_DEC, OP_PDEC, OP_ADD, OP_ADD_R, OP_SUB, OP_SUB_R, OP_MUL, OP_MUL_R, OP_DIV, OP_DIV_R, OP_REM, OP_REM_R, OP_BOR, OP_BOR_R, OP_BAND, OP_BAND_R, OP_XOR, OP_XOR_R, OP_SHL, OP_SHL_R, OP_SHR, OP_SHR_R, OP_NOT, OP_BNOT, OP_UNREF**
spezielle Funktionsnamen zur Definition von Operatoren (siehe 10.10)

Die reservierten Bezeichner sind *keine Schlüsselwörter* im oben beschriebenen Sinne. Sie besitzen jedoch implizit eine besondere Bedeutung im Programm und sollten deshalb nicht als allgemeine Bezeichner verwendet werden. Gleiches gilt für die Namen der vordefinierten Datentypen (vgl. Abschnitt 5.2) und die Bezeichner der vordefinierten benannten Literale (vgl. 4.3.8) sowie der vordefinierten Variablen (siehe 5.6).

8 Steuerstrukturen

Für jedes Programm, das aus mehr als einer bloßen Aneinanderreihung von Befehlen bestehen soll, werden Möglichkeiten zur Steuerung des Programmablaufs benötigt. Grundsätzlich unterscheidet man hier zwischen *Verzweigungen des Programmablaufs auf Grund von Bedingungen* und *Wiederholungen von Programmabschnitten*.

Viele Programmiersprachen erlauben zudem noch unbedingte Sprunganweisungen. Solche Anweisungen sind in B++ nicht vorgesehen³⁷ und werden deshalb nicht weiter behandelt.

8.1 Bedingungen (if-else)

Zur Steuerung der Programmabarbeitung in Abhängigkeit von einer Bedingung dient die **if-else-Anweisung**, die in Syntax und Semantik dem gleichnamigen Konstrukt in C/C++ entspricht:

```
if ( bedingung ) anweisung1 [ else anweisung2 ] .
```

Ergibt der Ausdruck *bedingung* den Wahrheitswert `true`, so wird *anweisung1*, andernfalls *anweisung2* ausgeführt. Die auszuführenden Anweisungen können einfache Anweisungen oder Anweisungsblöcke sein. Insbesondere kann hierfür die *if-else-Anweisung* selbst verwendet werden, wodurch die Formulierung geschachtelter Bedingungen bzw. Fallunterscheidungen möglich ist.

Beispiel:

```
main()
{
    FILE fp = fopen("testfile.txt", "rt");
    if (fp)                                // same as fp != null
    {
        // do anything
        fclose(fp);
    }
    else
        print("opening testfile.txt failed\n");
}
```

Beispiel 18 Ausführungssteuerung mit if-else

8.2 Fallunterscheidungen (switch-case)

B++ unterstützt Fallunterscheidungen mit Hilfe der **switch-case-Anweisung**, die sich in Syntax und Semantik an das entsprechende Konstrukt in C/C++ anlehnt und genau dem in JavaScript entspricht.

Anders als in C++ kann hinter `case` nicht nur eine Konstante, sondern ein beliebiger Ausdruck stehen. Der Ausdruck hinter `switch` kann ebenfalls beliebigen Typs sein, sofern ein Vergleich mit den Werten hinter `case` definiert ist.

Wie in C/C++ gibt es ein „fall through“, d. h., wird ein `case`-Zweig nicht mit `break` verlassen, werden die Anweisungen des folgenden Zweiges mit ausgeführt.

Der `default`-Zweig (Standardaktion) muss *hinter* den `case`-Zweigen notiert werden.

³⁷ Sie sind auch in keinem B++-Programm zwingend erforderlich.

Beispiel:

```
switch (myVariable)
{
    case 1:
        // do something
        break;
    case 2:
        // do any other
        // fall through
    default:
        // do something else
}
```

Beispiel 19 Verwenden einer Fallunterscheidung

8.3 Wiederholungen (Schleifen)

Für die Formulierung von Wiederholungen von Codeabschnitten kennt B++ die Anweisungen `while`, `do-while` und `for`. Syntax und Semantik dieser Anweisungen entsprechen dabei den gleichnamigen Konstrukten in C/C++.

8.3.1 Die while-Anweisung

Die **while-Anweisung** hat die allgemeine Form

```
while ( bedingung ) anweisung
```

Dabei wird *anweisung* so lange (wiederholt) ausgeführt, wie *bedingung* den Wert `true` ergibt.

Beispiel:

```
/* type contents of file to console */
void main()
{
    FILE fp = fopen("testfile.txt", "rt");
    if (fp) // same as fp != null
    {
        while(!feof(fp))
        {
            string s = fgets(fp);
            print(s);
        }
        fclose(fp);
    }
    else
        print("opening testfile.txt failed\n");
}
```

Beispiel 20 Verwenden der while-Schleife

8.3.2 Die do-while-Anweisung

Die **do-while-Anweisung** hat die allgemeine Form

```
do anweisung while ( bedingung )
```

Sie ist der `while`-Anweisung sehr ähnlich. Auch hier wird *anweisung* so lange ausgeführt, wie *bedingung* den Wert `true` ergibt. Der wesentliche Unterschied besteht darin, dass die Bedingung erst am Ende geprüft und damit *anweisung* stets mindestens einmal ausgeführt wird.

Beispiel:

```
/* prompt user for input until empty line was entered */
void main()
{
    string s;
    do
    {
        print("enter line: ");
        s = gets();
        // do anything
    }
    while (strlen(s) > 0);
}
```

Beispiel 21 Verwenden der do-while-Schleife

8.3.3 Die for-Anweisung

Die **for-Anweisung** ist die mächtigste Form der Formulierung einer Wiederholung. Wie in C/C++ (und im Gegensatz zu Sprachen wie BASIC oder PASCAL) dient sie nicht zwingend der Abarbeitung einer festgelegten Anzahl von Schleifendurchläufen, sondern ist eher eine Verallgemeinerung bzw. Erweiterung der *while*-Anweisung. Die for-Anweisung hat die allgemeine Form

```
for ( init ; bedingung ; reinit ) anweisung
```

Vor dem Beginn der Abarbeitung einer *for*-Schleife wird zunächst einmalig der Initialisierungsausdruck *init*³⁸ ausgeführt. Vor jedem Schleifendurchlauf wird der Ausdruck *bedingung* ausgewertet. Ergibt er *true*, so wird *anweisung* ausgeführt, andernfalls ist die Schleife beendet. Der Ausdruck *reinit* wird am Ende jedes Schleifendurchlaufs (d. h. nach Ausführung von *anweisung*) ausgeführt.

Beispiel:

```
/* list all arguments of commandline to console */
void main()
{
    auto argvec = getargs();           // get argument vector
    int n = vecsize(argvec);           // get number of elements
    for(int i=0; i<n; ++i)              // print each element to console
        print(argvec[i], "\n");
}
```

Beispiel 22 Verwenden der for-Schleife

Hinweis:

Die Ausdrücke im Kopf einer *for*-Anweisung können auch leer sein (weggelassen werden). Für *init* bzw. *reinit* wird dann keine Aktion ausgeführt. Wird *bedingung* weggelassen, so entsteht eine Endlosschleife, die nur mit Hilfe von *break* (siehe 8.3.5) oder durch eine Ausnahme (siehe Abschnitt 8.4) explizit abgebrochen werden kann. Der generierte Bytecode enthält dann keine Bedingungsprüfung, wodurch eine Endlosschleife der Form *for(;;)* ... schneller abgearbeitet wird als eine mit Hilfe *while(true)* ... konstruierte.

³⁸ Die Ergebnisse der Ausdrücke *init* und *reinit* sind dabei ohne Bedeutung. Semantisch haben sie die Funktion von Anweisungen.

8.3.4 Geschachtelte Schleifen

Schleifen können ineinander geschachtelt werden. Ein typischer Fall für die Anwendung geschachtelter Schleifen ist das Füllen eines mehrdimensionalen Feldes, wie in nachfolgendem Beispiel gezeigt:

```
/* create and fill a 2-dim array */
void main()
{
    int rows = 8;
    int cols = 8;
    /* create a table */
    vector array = newvector(rows);
    for (int i=0; i<rows; ++i) array[i] = newvector(cols);
    /* initialize table */
    for (i=0; i<rows; ++i)
        for (int j=0; j<cols; ++j)
            array[i][j] = rows * i + j;
    // do anything
    // ...
}
```

Beispiel 23 Bearbeiten mehrdimensionaler Felder mit geschachtelten for-Schleifen

Eine explizite Begrenzung der maximalen Schachtelungstiefe gibt es in B++ nicht.

8.3.5 Die Anweisungen break und continue

Die Anweisungen `break` und `continue` können innerhalb von Schleifen als zusätzliche Hilfsmittel zur Ablaufsteuerung eingesetzt werden. Mit `break` kann eine Schleife vorzeitig, d. h. unabhängig von der eigentlichen Abbruchbedingung, abgebrochen werden. Mit `continue` wird der Rest des aktuellen Anweisungsblocks einer Schleife übersprungen und – sofern die Abbruchbedingung nicht erfüllt ist – direkt zum nächsten Schleifendurchlauf übergegangen.

Beispiel:

```
/* write all printable characters of testfile.txt to console */
int main()
{
    FILE fp = fopen("testfile.txt", "rt");
    if (!fp)
    {
        print("error opening file\n");
        return 1;
    }
    while(true)                // loops forever
    {
        if (feof(fp))
            break;              // abort loop if end of file reached
        var c = getc(fp);
        if ((c<' ' && (c != '\n'))
            continue;           // ignore control characters other than EOL
        putc(c, stdout);        // write character to console
    }
    fclose(fp);
    return 0;
}
```

Beispiel 24 Schleifensteuerung mit break und continue

8.4 Fehlerbehandlung (try – catch – throw)

Das Konzept der Fehlerbehandlung orientiert sich an dem von C++. Dementsprechend wird mit dem Schlüsselwort `try` ein „geschützter Block“ eingeleitet, dem ein mit `catch` eingeleiteter Block zur Behandlung von Ausnahmen folgt.

Ein try-catch-Block hat die allgemeine Form

```
try anweisungsblock catch ( exceptionid ) anweisungsblock
```

Mit

```
throw exception
```

kann innerhalb eines ausführbaren Programmabschnitts (einer Funktion) eine Ausnahme (*exception*) ausgelöst werden. Innerhalb eines `catch`-Blocks kann `throw` ohne nachfolgende Angabe einer Ausnahme verwendet werden, um die bereits abgefangene Ausnahme erneut auszulösen. B++ gestattet – wie C++ – die Verwendung beliebiger Datentypen für das Auslösen von Ausnahmen.

Beispiel:

```
try
{
    var myVar = anyFunc();
    if (myVar) < 0)
        throw "myVar must not be negative";
    // do something else
}
catch (e)
{
    print(e, "caught\n");
    throw; // rethrow exception
}
```

Beispiel 25 Auslösen und Behandeln von Ausnahmen

Der für die zu behandelnde Ausnahme im `catch`-Block verwendete Bezeichner kann beliebig gewählt werden und ist, anders als andere lokale Variablen, nur innerhalb dieses Blocks (nicht innerhalb der gesamten umschließenden Funktion) sichtbar. Er überdeckt zudem im Falle eines Namenskonflikts eine gleichnamige lokale Variable.

Weil B++ keine statische Typisierung von Ausnahmen kennt, ist die Definition mehrerer aufeinanderfolgender `catch`-Blöcke nicht sinnvoll und syntaktisch auch nicht zugelassen.

9 Funktionen

Wie bereits in Kapitel 3 angedeutet, sind Funktionen die zentralen Strukturierungselemente eines jeden B++-Programms. Grundsätzlich ist dabei zwischen zwei Typen von Funktionen, den *vordefinierten* und den *benutzerdefinierten* zu unterscheiden. Vordefinierte Funktionen (siehe Kapitel 11) sind fester Bestandteil des B++-Laufzeitsystems. Sie sind in jedem Programm verfügbar und können vom Benutzer, d. h. dem Programmierer, normalerweise³⁹ nicht geändert werden.

Das eigentliche Verhalten eines Programms wird mit Hilfe der benutzerdefinierten Funktionen festgelegt. Der Einsprungpunkt `main` – siehe Kapitel 3 – ist selbst eine solche Funktion.

9.1 Funktionsdefinition

Eine benutzerdefinierte Funktion hat die allgemeine Form

```
[returntype] funcname ( parameterlist [; localvarlist] ) body
```

Dabei ist *funcname* der Funktionsname, der ein gültiger Bezeichner (siehe 4.2) sein muss. *parameterlist* ist eine kommaseparierte Liste benannter Funktionsparameter und *localvarlist* definiert die Namen der innerhalb der Funktion gültigen weiteren lokalen Variablen (ebenfalls als kommaseparierte Liste). *body* schließlich ist ein Anweisungsblock, der den Funktionsrumpf enthält. Sowohl *parameterlist* als auch *localvarlist* sind optional. Fehlt *localvarlist*, so kann auch das als Trennzeichen zwischen den Listen verwendete Semikolon entfallen.

Vor dem Funktionsnamen kann optional ein Rückgabetytpe spezifiziert werden, für den eines der Schlüsselwörter `var`, `null` bzw. `void` oder ein konkreter Typname angegeben werden kann. Dabei verbietet `void` die Rückgabe eines Wertes, während bei Angabe von `var`, `null` oder einem Typnamen verlangt wird, dass eine `return`-Anweisung tatsächlich mit einem Rückgabewert verwendet wird. Vor den benannten Parametern ist optional das Schlüsselwort `var` oder die Angabe eines Typbezeichners zulässig. Wird hier ein Typbezeichner angegeben, so ist das entsprechende Argument statisch typisiert.

Die Definition lokaler Variablen kann nicht nur in der Liste im Funktionskopf, sondern auch an beliebiger Stelle im Funktionsrumpf mit Hilfe des Schlüsselwortes `var` oder unter Angabe eines Typbezeichners bzw. des Schlüsselwortes `auto` (zur Definition einer statisch typisierten Variablen) erfolgen. Dies ermöglicht (bzw. erzwingt im Falle von `auto`) auch die unmittelbare Initialisierung der jeweiligen Variablen bei ihrer Definition. Auch im Funktionsrumpf definierte Variablen sind innerhalb der gesamten Funktion sichtbar, nicht nur im sie umschließenden Anweisungsblock. Die Variablendefinition muss im Quelltext vor der Verwendung einer lokalen Variablen erfolgen.

Benannte Parameter werden innerhalb des Funktionsrumpfes wie lokale Variablen behandelt. Globale Bezeichner sind innerhalb einer Funktion sichtbar, sofern sie nicht durch gleichnamige lokale Bezeichner überdeckt werden. Im Falle der Überdeckung können globale Variablen mit Hilfe des vorangestellten Operators `::` referenziert werden.

Im Unterschied zu den meisten anderen Skriptsprachen erlaubt B++ *keine implizite Definition globaler Variablen* durch einfache Wertzuweisung an einen beliebigen Bezeichner. Diese Festlegung vermeidet die Gefahr des unbeabsichtigten Erzeugens von globalen Variablen auf Grund von Tippfehlern. Ist es tatsächlich beabsichtigt, innerhalb eines Funktionsrumpfes eine globale Variable zu erzeugen, so muss dies explizit mit Hilfe der Verarbeitungsanweisung `#defvar` (vgl. 4.5.5) erfolgen.

³⁹ Prinzipiell ist es möglich, vordefinierte Funktionen durch benutzerdefinierte zu ersetzen und unter Nutzung von Funktionszeigern (siehe Abschnitt 9.5) sogar zu erweitern.

Anders als in C/C++ geben Funktionen in B++ streng genommen *immer* einen Funktionswert zurück. Wird eine Funktion mit `return`, gefolgt von einem Argument, verlassen, so ist der Wert dieses Arguments der Funktionswert. Ansonsten ist der Wert undefiniert – genau genommen der Wert, der gerade zuoberst auf dem Stack liegt. Wird der Rückgabetyt einer Funktion explizit angegeben, so erfolgt nach dem Funktionsaufruf eine Prüfung des Typs des Funktionswertes⁴⁰. Ist der Rückgabewert ein Literal, wird die Typprüfung bereits durch den Compiler vorgenommen.

Beispiele für Funktionsdefinitionen:

```
// parameterless function without return value
void sayHello()
{
    print("Text from function body\n");
}

// named parameter, local variables and return value
var factorial(n; i, result)
{
    for(result=1, i=n; i>1; --i)
        result *= i;
    return result;
}

// same as above, but using static typed variables declared in body
int factorial(int n)
{
    int result = 1;
    for(var i=n; i>1; --i)
        result *= i;
    return result;
}

/* no named parameters but local variable and return value */
var queryName(;name)
{
    name = "";
    while (strlen(name) == 0)
    {
        print("Tell me your name: ");
        name = fgets(stdin);
    }
    return name;
}

/* same as above but using static typed variable definition in body */
string queryName2()
{
    string name = "";
    while (strlen(name) == 0)
    {
        print("Tell me your name: ");
        name = fgets(stdin);
    }
    return name;
}
```

Beispiel 26 Varianten von Funktionsdefinitionen

⁴⁰Der Typ des Rückgabewertes muss im Allgemeinen **genau** dem deklarierten Typ entsprechen, es werden keinerlei implizite Konvertierungen vorgenommen. Die einzige Ausnahme bilden hier Objekttypen, die als untereinander kompatibel angesehen werden.

9.2 Aufruf von Funktionen

Der Aufruf einer Funktion erfolgt durch Nennung des Funktionsnamens, gefolgt von einer geklammerten kommaseparierten Liste der Parameter. Syntaktisch kann ein Funktionsaufruf als einzelne Anweisung oder an Stelle eines Wertes innerhalb eines Ausdrucks stehen.

Beispiel:

```
main()
{
    sayHello();           // simple statement - return value is ignored
    var f = factorial(17); // assign return value to variable
    var g = sqrt(2.0) * 0.5; // use function in expression
    print(queryName());   // use function as parameter of another one
}
```

Beispiel 27 Varianten des Funktionsaufrufs

Hinweise:

- Beim Aufruf einer Funktion müssen *mindestens* so viele Parameter angegeben werden, wie in der Funktionsdeklaration festgelegt, sofern keine Vorgabewerte für Parameter definiert wurden (vgl. 9.3). Werden mehr Parameter angegeben, so werden den benannten Parametern die *ersten* (d. h. am weitesten links stehenden) Parameterwerte zugewiesen.
- Die Überprüfung der Anzahl der übergebenen Parameter erfolgt zur Laufzeit. Werden beim Aufruf einer Funktion weniger Parameter übergeben als laut Deklaration erforderlich, führt dies also nicht zu einem Compiler-, sondern zu einem Laufzeitfehler.

Wie C/C++ unterstützt B++ den rekursiven Aufruf von Funktionen.

Beispiel:

```
var factorial(var n)    // recursive version of factorial function
{
    if (n > 1)
        return n * factorial(n-1);
    return 1;
}
```

Beispiel 28 Rekursiver Aufruf von Funktionen

Die mögliche Rekursionstiefe wird dabei lediglich durch die Größe des Stacks begrenzt. Weil B++ einen dynamischen Stack benutzt, ist dies – einen ausreichend großen Arbeitsspeicher vorausgesetzt – ohne praktische Bedeutung. Bei sehr großen Rekursionstiefen ergibt sich eine Beschränkung eher aus der Größe des für die Ausführungsumgebung verfügbaren Stacks, da die B++-VM Funktionsaufrufe selbst rekursiv verarbeitet (in MS-DOS-Umgebungen hat der Stack eine Größe von maximal 64 KB, unter Win32 standardmäßig 1 MB).

9.3 Variable Parameterlisten und Vorgabeparameter

Wie im vorherigen Abschnitt erwähnt, ist es möglich, einer Funktion beim Aufruf mehr Parameter zu übergeben als in der Deklaration angegeben. B++ erlaubt unter Ausnutzung dieser Eigenschaft die Konstruktion von Funktionen mit einer beliebigen Anzahl von Parametern. Für den Zugriff auf die übergebenen Parameter wird dann an Stelle von Parameternamen die vordefinierte Funktion `arg` verwendet. Die Bestimmung der Parameteranzahl erfolgt mit `argcnt` (siehe 11.9). Ab Version 1.2 von B++ kann optional die ‚Ellipse‘ (...) als letztes Element der formalen Parameterliste einer Funktion verwendet werden, um die Parameterliste explizit als variabel zu kennzeichnen und so eine Notation analog zu C/C++ zu erreichen.

```

var sum(...)
{
    var result = 0;
    var n = argcnt();
    for (var i=0; i<n; ++i)
        result += arg(i);
    return result;
}

void main()
{
    print(sum(0,1,2,3,4,5,6,7,8,9), "\n");
}

```

Beispiel 29 Verwenden variabler Parameterlisten

Darüber hinaus können für benannte Parameter in einer Funktionsdeklaration Vorgabewerte angegeben werden. Solche Werte werden implizit eingesetzt, wenn die betreffende Funktion mit einer geringeren als der deklarierten Parameteranzahl aufgerufen wird. Als Vorgabewerte sind beliebige Literalwerte (auch literale Ausdrücke) zulässig.

Auch dazu ein Beispiel:

```

// builds string serialization of a vector
string serializeVector(vector vec, string delimiter = ",")
{
    string result = "";
    int n = size(vec);
    for (int i=0; i<n; ++i)
    {
        if (i>0) result += delimiter;
        result += vec[i];
    }
    return result;
}

void main()
{
    // print comma separated list of command line arguments
    print(serializeVector(getargs()), "\n");
    // print command line arguments, each argument at its own line
    print(serializeVector(getargs(), "\n"), "\n");
}

```

Beispiel 30 Optionale Funktionsargumente mit Vorgabeparametern

Hier werden die beim Aufruf angegebenen Kommandozeilen-Parameter zunächst als kommaseparierte Liste (unter Nutzung des Vorgabewertes für das Argument *delimiter*) und anschließend zeilenweise ausgegeben.

9.4 Statische Variablen in Funktionen

Innerhalb einer Funktion können mit Hilfe des Schlüsselworts `static` *statische* Variablen definiert werden, deren Sichtbarkeit auf die jeweilige Funktion beschränkt ist, die jedoch – im Unterschied zu 'normalen' lokalen Variablen – eine feste Adresse besitzen (und nicht beim Aufruf temporär auf dem Stack angelegt werden).

Die Definition statischer (lokaler) Variablen erfolgt in der Form

```
static [type] name [= value];
```

Dem Variablenbezeichner (*name*) kann eine explizite Typangabe (*type*) vorangestellt werden. Optional kann die Variable mit einem Anfangswert (*value*) initialisiert werden. Die Initialisierung

erfolgt während der Übersetzung (nicht zur Laufzeit), weshalb *value* ein Literal, vgl. Abschnitt 4.3, sein muss.

Statische Variablen behalten ihren Wert zwischen den Aufrufen einer Funktion und erlauben so die Verwendung von Statusinformationen innerhalb einer Funktion. Typische Anwendungsfälle dafür sind Zähl- und Generatorfunktionen – siehe Beispiel 31.

```
// simple ID generator function
uint createID()
{
    static uint _id = 0;
    return ++_id;
}
```

Beispiel 31 Einfache Generatorfunktion mit statischem Zähler

9.5 Funktionszeiger

In Abschnitt 5.2.2 wurde der Datentyp `function` erklärt, der einem Wert vom Funktionstyp zugeordnet ist. Tatsächlich lässt sich eine Funktion in gewissen Grenzen wie ein „normaler“ Wert behandeln, d. h. insbesondere an eine Variable zuweisen, anderen Funktionen als Parameter übergeben oder selbst als Rückgabewert einer Funktion verwenden. Da Werte vom Funktionstyp nichts anderes als einen Zeiger auf den ausführbaren Code der jeweiligen Funktion speichern, sind sie mit den Funktionszeigern in C/C++ vergleichbar und können in analoger Weise verwendet werden.

Beispiel:

```
void myFunc()
{
    print("myFunc called\n");
}

main(;f)
{
    f = myFunc;           // assign myFunc to variable f
    f();                  // call function
}
```

Beispiel 32 Definition und Verwendung eines Funktionszeigers

9.6 Überladen von Funktionen

B++ identifiziert eine Funktion ausschließlich anhand ihres Namens und nicht (wie z. B. C++) anhand der vollständigen Signatur. Damit ist ein Überladen, d. h. die Definition mehrerer Funktionen gleichen Namens, im eigentlichen Sinne nicht möglich. Die Neudefinition einer bereits vorhandenen Funktion führt zwar nicht zu einem Fehler, ersetzt die vorher definierte gleichnamige Funktion aber vollständig.

Unter Ausnutzung der in den vorangegangenen Abschnitten beschriebenen variablen Parameterlisten und Funktionszeiger ist in B++ jedoch ein „Pseudo-Überladen“ möglich. Beispiel 33 Zeigt das Prinzip:

```

var myFunc1 = null;
myFunc(i) {
    print("myFunc(", i, ") called\n");
}
myFunc2() {
    if (argcnt() != 2)
        myFunc1(arg(0));
    else print("myFunc(", arg(0), ",", arg(1), ") called\n");
}
main() {
    myFunc1 = myFunc;
    myFunc = myFunc2;
    myFunc(1, 2);
    myFunc(1);
}

```

Beispiel 33 Pseudo-Überladen von Funktionen

Ziel ist es hier, zwei Funktionen zu definieren, die beide über den Namen `myFunc` aufrufbar sind, wobei die eine einen Parameter und die andere zwei Parameter verarbeitet.

Zuerst wird unter dem Namen `myFunc` die Variante mit einem Parameter definiert. Die Variante für zwei Parameter erhält zunächst einen anderen Namen (`myFunc2`) und wird so implementiert, dass sie eine variable Anzahl von Parametern verarbeiten kann. Ist die Anzahl der übergebenen Parameter nicht zwei, so ruft sie eine Funktion `myFunc1` auf, andernfalls wird der eigene Code ausgeführt.

Die Funktion `main` weist nun der (globalen) Variablen `myFunc1` den Code von `myFunc` und `myFunc` den Code von `myFunc2` zu.

9.7 Anonyme Funktionen und Funktionslitterale

Eine anonyme Funktion ist eine Funktion ohne zugeordneten Namen. Die Definition einer anonymen Funktion erfolgt stets mit Hilfe eines Funktionsliterals der Form:

```
function [ [classid] ] [returntype]( parameterlist ; localvarlist ) body
```

Sie unterscheidet sich also von einer „normalen“ Funktionsdefinition (vgl. 9.1) zunächst dadurch, dass sie durch das Schlüsselwort `function` eingeleitet wird.

Hinter `function` kann optional ein Bezug zu einer Klasse (`classid`), eingeschlossen in eckige Klammern, angegeben werden, wobei für `classid` der Bezeichner einer Klasse steht. Auf diese Weise lassen sich auch *anonyme Methoden*, also mit einer Klasse verbundene Funktionen (vgl. Abschnitt 10.7), definieren. Anschließend wird – ebenfalls optional – der Typ des Rückgabewertes angegeben, danach folgen die Spezifikation der Parameter und ggf. der lokalen Variablen sowie der Implementierungsblock.

Anonyme Funktionen unterscheiden sich in ihrem Verhalten nicht von regulären Funktionen, eröffnen jedoch einige zusätzliche Möglichkeiten:

- Sie können lokalen Variablen innerhalb einer Funktion (oder Member-Variablen eines Objekts) zugewiesen werden und somit nur lokal sichtbar sein.
- Sie können wie Literale gebraucht und somit überall verwendet werden, wo ein Literalwert stehen kann – insbesondere auch als Bestandteil literaler Ausdrücke sowie als Wert von Vorgabeparametern (siehe 9.3) oder innerhalb der Literalkonstrukturen der Typen `vector` bzw. `dictionary`.
- Sie können an der Stelle eines Wertes, z. B. innerhalb eines Funktionsaufrufs oder eines Ausdrucks inline definiert und unmittelbar ausgeführt werden.

Beispiele:

```
// define replacement for WIN HIWORD/LOWORD macros using named literals
#define HIWORD function(x) { return (x >> 16) & 0xFFFF;}
#define LOWORD function(x) { return x & 0xFFFF;}

// use anonymous compare function for argument of qsort function
var myVec = [4,2,1,7,0,8,9];
qsort(myVec, function (a,b) { return a-b; });

// call anonymous function in an expression
var str = function() {
    var result = "";
    for (var c = getc(); c >= ' '; c = getc()) result += c;
    return result;
}();
print(str);
```

Beispiel 34 Anonyme Funktionen

Hinweis:

Von einem (kompilierten) Funktionsliteral, das – wie in obigem Beispiel – als anonyme Funktion in einem Ausdruck oder als Argument direkt aufgerufen wird, wird bei jedem Aufruf eine temporäre Instanz erzeugt, falls die betreffende Funktion mindestens eine lokale statische Variable definiert, was mit einem bestimmten Zeitaufwand verbunden ist.

Bei sehr häufigen Aufrufen, z. B. in einer Schleife, ist es dann sinnvoll, das Literal zunächst an eine lokale Variable zuzuweisen und diese im betreffenden Ausdruck zu verwenden. Die obige Sortierung ließe sich auch so schreiben:

```
// use anonymous compare function for argument of qsort function
var myVec = [4,2,1,7,0,8,9];
var cmpfunc = function (a,b) { return a-b; };
qsort(myVec, cmpfunc);
```

Beispiel 35 Anonyme Funktion als Vergleichsfunktion beim Sortieren eines Arrays

In diesem Fall ergeben sich daraus allerdings keine Laufzeitunterschiede, da die hier verwendete Vergleichsfunktion keine statischen Variablen definiert.

Rekursive Aufrufe anonymer Funktionen

Weil eine anonyme Funktion keinen Namen besitzt, steht die Frage, wie rekursive Aufrufe realisiert werden können. B++ verwendet dazu den reservierten Bezeichner `self`, der innerhalb einer Anweisung (bzw. eines Ausdrucks) verwendet werden kann, um die umschließende Funktion selbst zu referenzieren. Er wird dabei wie ein gewöhnlicher Funktionsname gebraucht.

Beispiel:

```
// recursive compute Fibonacci number
var fib = function int(int arg) {
    return arg < 2 ? arg : self(arg-1)+self(arg-2);
};
print(fib(12)); // prints 144
```

Beispiel 36 Verwenden von `self` zur Rekursion in anonymen Funktionen

Die Verwendung von `self` für rekursive Aufrufe ist nicht zwingend auf anonyme Funktionen beschränkt, sondern auch in benannten Funktionen und Member-Funktionen (Methoden) von Objekten – vgl. 10.4 – möglich.

10 Objektorientierte Programmierung

Das Konzept der Objektorientierung in B++ ist trotz syntaktischer Ähnlichkeit einfacher als das von C++. Dennoch werden die grundlegenden Paradigmen objektorientierter Programmierung – Datenkapselung, Vererbung und Polymorphie – berücksichtigt.

Darüber hinaus bestehen zusätzliche Möglichkeiten, wie die Konstruktion partieller Klassen, nachträgliche Redefinition von Member-Funktionen sowie die Definition von Operatoren für Objekte.

In den folgenden Abschnitten werden die Prinzipien und Möglichkeiten beschrieben.

10.1 Aufbau einer Klasse

Zur Erläuterung des Aufbaus einer Klasse in B++ soll Beispiel 37 dienen.

Es stellt den Anfang einer fiktiven Klassenbibliothek mit der Wurzel `MyBaseClass` und einer davon abgeleiteten Klasse `MyDerivedClass` dar. Alle Objektinstanzen dieser Bibliothek sollen einen eindeutigen Identifikator (`_ID`) besitzen, dessen Vergabe die einzige Aufgabe der Basis-klasse ist.

```
// declaration of MyBaseClass
class MyBaseClass
{
    MyBaseClass();
    uint getID();
    uint _ID;
    static uint createID();
    static uint _maxID = 0;    // ID counter, initialized to zero
}

// implementation of MyBaseClass
MyBaseClass::MyBaseClass() { _ID = createID(); }
uint MyBaseClass::getID() { return _ID; }
uint MyBaseClass::createID() { return ++_maxID; }

// declaration of MyDerivedClass
class MyDerivedClass : MyBaseClass
{
    MyDerivedClass(string name = "", var value = null);
    string getName();
    void setName(string name);
    var getValue();
    void setValue(var value);
    // member variables
    string _name;
    var _value;
}

// implementation of MyDerivedClass
MyDerivedClass::MyDerivedClass(var name = "", var value = null)
{
    MyBaseClass();
    _name = name;
    _value = value;
}
var MyDerivedClass::getName() { return _name; }
void MyDerivedClass::setName(var name) { _name = name; }
var MyDerivedClass::getValue() { return _value; }
void MyDerivedClass::setValue(var value) { _value = value; }
```

Beispiel 37 Definition von Klassen

Die Klassendeklarationen, wie auch die Implementierungen, ähneln auf den ersten Blick dem von C++ Gewohnten. Allerdings gibt es eine Reihe von wichtigen Unterschieden:

Sichtbarkeit von Elementen

Ohne explizite Angabe von Zugriffsspezifizierern sind in B++ Member- und Klassenfunktionen sowie Klassenvariablen (`static`) implizit öffentlich (`public`).

Member-Variablen sind implizit geschützt (`protected`), also nur in der deklarierenden Klasse und ihren Ableitungen sichtbar.⁴¹

Für eine explizite Festlegung des Zugriffs stehen die Schlüsselwörter `public`, `protected` und `private` zur Verfügung, für deren Verwendung die gleichen Regeln wie in C++ gelten.

Anders als in C++ gibt es keine `friend`-Deklarationen, also keine Möglichkeit, die Zugriffsregeln für einzelne ‚fremde‘ Klassen oder Funktionen aufzuheben.

Keine Mehrfachvererbung

Anders als in C++ hat eine B++-Klasse höchstens eine Basisklasse. Auch ein Interface-Konzept, wie etwa von Java oder C# bekannt, gibt es nicht.

Die Vererbung selbst ist immer öffentlich, d. h. alle öffentlichen Elemente der Basisklasse werden auch öffentliche Elemente der abgeleiteten Klassen.

Deklaration von Variablen und Methoden

In B++-Klassen ist die Nennung von Member-Funktionen in der Klassendeklaration optional. Etwa hätte man `MyDerivedClass` in obigem Beispiel auch so deklarieren können⁴²:

```
MyDerivedClass : MyBaseClass
{
    string _name;
    var _value;
}
```

Beispiel 38 Minimale Klassendeklaration

Diese Eigenschaft erlaubt es, Klassen bei Bedarf nachträglich um Methoden zu ergänzen (siehe hierzu auch 10.9).

Member-Variablen und alle statischen Elemente *müssen* deklariert werden.

Steht der `static`-Modifizierer vor einer kommaseparierten Variablenliste, so gilt er für alle ihre Elemente.

Hinweis:

Als Konsequenz aus der optionalen Deklaration von Methoden und der Möglichkeit, Methoden später neu zu definieren, ergibt sich, dass bei der späteren Implementierung angegebene Argumentlisten (und auch Standardargumente) die ggf. in der Deklaration aufgeführten vollständig ersetzen. Anders als z. B. in C++ müssen deshalb auch Standard-Argumentwerte bei der Implementierung im Methodenkopf angegeben werden – siehe Beispiel 37.

Eingeschränkte Initialisierung von Klassenvariablen

In B++ können Member- und Klassenvariablen (statische Member) bei ihrer Deklaration initialisiert werden, wobei die Initialisierung bereits zur Übersetzungs- und nicht erst zur Laufzeit erfolgt. Deshalb sind zur Initialisierung nur Literale (und keine Funktionsaufrufe) verwendbar. Die Initialisierung muss stets innerhalb der Klassendeklaration notiert werden (siehe `_maxID` in `MyBaseClass`).

⁴¹Dies entspricht den Regeln von BOB bzw. BOB+, die keine Zugriffsspezifizierer kennen.

⁴²Normalerweise ist von dieser Kurzform eher abzuraten, da sie die Lesbarkeit des Quelltextes sicher nicht verbessert, sie erlaubt jedoch die Erweiterung vorhandener Klassen um zusätzliche Methoden selbst dann, wenn die zu erweiternde Klasse nicht im Quelltext (also lediglich in einem vorübersetzten Modul) vorliegt. Nicht in der Klassendeklaration aufgeführte Elementfunktionen sind immer öffentlich (`public`).

Objekte werden immer dynamisch erzeugt

Alle Instanzen von Klassen (Objekte) haben den internen Typ `object` und sind damit Referenztypen (siehe 5.2.2). Der einzige Weg sie zu erzeugen führt über den Operator `new`:

```
var myObj = new MyDerivedClass("anumber",1000);
```

Dies gilt technisch gesehen auch für statisch typisierte Objektvariablen, die bei der Definition initialisiert werden. Die Notation

```
MyDerivedClass myObj("anumber",1000);
```

ist nicht mehr als eine verkürzte Schreibweise für

```
MyDerivedClass myObj = new MyDerivedClass("anumber",1000);
```

Deshalb ist es nicht möglich, wie in C++ direkt im Anschluss an eine Klassendeklaration Variablen dieses Typs anzulegen, was auch erklärt, weshalb hinter der schließenden Klammer der Klassendeklaration kein Semikolon steht⁴³.

In diesem Zusammenhang ist auch zu erwähnen, dass eine Zuweisung an eine statisch typisierte Objektvariable zur Initialisierung – anders als in C++ - kein Objekt konstruiert.

```
MyDerivedClass myObj = "anumber";
```

würde lediglich zu einem Fehler führen, weil hier versucht wird, an eine Objektvariable des Typs `MyDerivedClass` eine Zeichenkette zuzuweisen.

10.2 Inline-Definition von Klassen

B++ erlaubt es auch, Klassen ganz oder teilweise „*inline*“ zu definieren, d. h. die Implementierung der Methoden einer Klasse direkt innerhalb der Klassendefinition zu notieren. Beispiel 39 zeigt eine entsprechende Abwandlung von Beispiel 37, wobei hier zusätzlich auch Zugriffsspezifizierer verwendet werden.

⁴³ Genau genommen müsste es hier „... kein Semikolon stehen muss“ heißen. Die Notation eines Semikolons ist an dieser Stelle – vor allem zur Unterstützung von Dokumentationstools – zulässig, allerdings überflüssig.

```
// definition of MyBaseClass
class MyBaseClass
{
private:
    uint _ID;
    static uint _maxID = 0;    // ID counter, initialized to zero
public:
    static uint createID() { return ++_maxID; }
    MyBaseClass() { _ID = createID(); }
    uint getID() { return _ID; }
}

// definition of MyDerivedClass
class MyDerivedClass : MyBaseClass
{
private:
    string _name;
    var _value;
public:
    MyDerivedClass(string name = "", var value = null)
    {
        MyBaseClass();
        _name = name;
        _value = value;
    }
    string getName() { return _name; }
    void setName(string name) { _name = name; }
    var getValue() { return _value; }
    void setValue(var value) { _value = value; }
}
```

Beispiel 39 Inline-Definition von Klassen

Zu beachten ist dabei, dass alle Elemente einer Klasse, die innerhalb einer Inline-Implementierung verwendet werden, *vor ihrer Verwendung deklariert* sein, d. h. in der Klassendefinition vor der jeweiligen Methode notiert werden müssen.

Die Inline-Implementierung führt insgesamt zu einer verkürzten und mitunter übersichtlicheren Notation. Die Trennung von Deklaration und Definition (Implementierung) ist dagegen eher zu bevorzugen, wenn eine Klassenschnittstelle als öffentlicher Bestandteil einer Bibliothek bereitgestellt werden soll.

10.3 Konstruktoren und Destruktoren

Ein *Konstruktor* ist eine spezielle Member-Funktion eines Objekts, deren Aufgabe hauptsächlich darin besteht, bei der Objekterzeugung die Member-Variablen mit geeigneten Anfangswerten zu initialisieren.

In B++ hat ein Konstruktor – wie in C++ – denselben Namen wie die jeweilige Klasse. Da ein Überladen von Funktionen im eigentlichen Sinne nicht möglich ist (siehe 9.6), kann eine Klasse nur einen Konstruktor besitzen. Dieser Konstruktor muss einen Verweis auf das erzeugte Objekt als Funktionswert zurückgeben. In der Urversion von BOB [2] bedeutete dies, dass ein Konstruktor stets mit der Anweisung

```
return this;
```

verlassen werden musste. Der Compiler von B++ fügt den entsprechenden Bytecode automatisch an das Ende jeder Konstruktorfunktion an, so dass die Anweisung hier nur dann notiert werden muss, wenn der Konstruktor vorzeitig (auf Grund einer Abbruchbedingung) verlassen wird.

Anders als z. B. in C++ ruft ein Konstruktor in B++ nicht automatisch auch den Konstruktor einer ggf. vorhandenen Basisklasse auf. Ein solcher Aufruf muss explizit kodiert werden. Ein Beispiel zur Verdeutlichung:

```
class Base
{
    Base();
}

Base::Base()
{
    print("ctor of Base called\n");
}

class Derived : Base
{
    Derived();
}

Derived::Derived
{
    Base();
    print("ctor of Derived called\n");
}

main()
{
    var obj = new Derived();
    // do anything
    obj = null; // releases object
}
```

Beispiel 40 Konstruktoraufruf in B++

Hinweise:

- Ein Konstruktor lässt sich in B++ bei Bedarf auch wie eine „normale“ Member-Funktion aufrufen.
- Von einer Klasse, die keinen eigenen Konstruktor besitzt, können keine Instanzen erzeugt werden. Diese Eigenschaft lässt sich zur Definition *abstrakter Klassen* nutzen⁴⁴.

Ein *Destruktor* ist gewissermaßen das Gegenstück zum Konstruktor. Er dient dazu, beim Zerstören eines Objekts ggf. von ihm belegte externe Ressourcen (zusätzlicher Speicher, offene Dateien etc.) wieder freizugeben.

B++ übernimmt im Wesentlichen das Konzept der Destrukturen aus C++. Konkret heißt dies:

- Ein Destruktor wird als parameterlose Member-Funktion deklariert, die den Klassennamen mit vorangestellter Tilde (~) trägt.
- Der Destruktor wird unmittelbar vor der Zerstörung des Objekts automatisch aufgerufen, d. h. innerhalb des Destruktors sind alle Elemente (einschließlich des *this*-Verweises) noch gültig.
- Am Ende der Destruktorfunktion werden rekursiv auch die Destrukturen aller Basisklassen aufgerufen, d. h. der Destruktorcode der abgeleiteten Klasse wird immer *vor* dem der Basisklasse ausgeführt. Klassen ohne Destruktor werden dabei innerhalb der Klassenhierarchie übersprungen.

⁴⁴Als Alternative dazu kann ein Konstruktor auch als `protected` deklariert werden um das Erzeugen von Objektinstanzen zu verhindern.

Zur Verdeutlichung wird obiges Beispiel erweitert:

```
class Base
{
    Base();
    ~Base();
    getClassName();
}

Base::Base() {
    print("ctor of Base called\n");
}

Base::~~Base() {
    print("dtor of Base called\n");
}

Base::getClassName() { return "Base"; }

class Derived : Base
{
    Derived();
    ~Derived();
    getClassName();
}

Derived::Derived {
    Base();
    print("ctor of Derived called\n");
}

Derived::~~Derived {
    print("dtor of Derived called\n");
}

Derived::getClassName() { return "Derived"; }

main() {
    var obj = &new Derived();
    // do anything
    obj = delete obj;
}
```

Beispiel 41 Destruktoraufrufe in B++

Anmerkungen:

- In Beispiel 41 wird der unäre Operator & bei der Zuweisung der `Derived`-Instanz an die Variable `obj` verwendet, um so eine explizite Speicherverwaltung zu erzwingen. Dies dient ausschließlich der Demonstration der Verwendung des *delete*-Operators. Im Normalfall würde man eine einfache Zuweisung verwenden, wodurch `obj` beim Verlassen der Funktion automatisch freigegeben würde.
- Der Operator *delete* gibt in B++ immer `null` zurück. Er kann deshalb – wie hier in der Funktion `main` – als rechter Ausdruck einer Zuweisung verwendet werden, um die freigegebene Variable als ungültig zu markieren.
- Implementierungsbedingt bildet B++ den Destruktor auf eine Member-Funktion mit dem reservierten Bezeichner *dtor* ab. Dieser Bezeichner kann alternativ zur C++-konformen Schreibweise auch für die Deklaration/Definition des Destruktors verwendet werden. Außerdem lässt sich der Destruktor über diesen Namen wie eine normale Member-Funktion aufrufen⁴⁵.

⁴⁵ Vorsicht: Auch hier werden die Destruktoren der Basisklassen mit aufgerufen!

10.4 Zugriff auf Member-Funktionen und -Variablen innerhalb einer Klasse

Enthält die Implementierung einer Member-Funktion Funktionsaufrufe oder Variablenzugriffe, so wird zunächst in der aktuellen Klasse und ggf. deren Basisklassen nach einem passenden Element gesucht. Verläuft die Suche ergebnislos, wird der jeweilige Funktions- oder Variablenbezeichner als globaler Bezeichner angenommen. Steht vor dem Bezeichner das Schlüsselwort *this* mit nachfolgendem Methoden-Aufrufoperator (*->*), so wird der Bezeichner nur im jeweiligen Objekt gesucht.

Für statische Elemente ist das Verhalten ähnlich, jedoch werden hier zur Spezifikation einer konkreten Klasse vor dem eigentlichen Bezeichner der Klassenname und der Bereichsauflösungsoperator (*::*) angegeben.

In diesem Zusammenhang entsteht das Problem, dass unter Umständen in der Implementierung einer Member-Funktion auf ein globales Symbol (Variable oder Funktion) zugegriffen werden muss, das namensgleich mit einem Element der aktuellen Klasse ist.

In B++ kann – wie in C++ – der Zugriff auf die globalen Symbole mit Hilfe des Bereichsauflösungsoperators erzwungen werden, wie Beispiel 42 zeigt.

```
// globally defined function
message() { print("global function message called\n") };

class MyClass
{
    MyClass();
    message();    // member-function message
    doAction();
}

MyClass() {}
MyClass::message() { print("function MyClass::message\n");}
MyClass::doAction(obj)
{
    message();           // call internal message method
    this->message();      // call internal message method,
                          // search in current class only
    ::message()          // call global function message
}
```

Beispiel 42 Auflösung globaler Bezeichner

10.5 Virtuelle Methoden

In B++ gibt es keine „frühe Bindung“ im eigentlichen Sinne (Abschnitt 10.6 beschreibt, wie entsprechendes Verhalten dennoch realisierbar ist). Das bedeutet, dass alle Methoden einer Klasse – auch die statischen⁴⁶ – *virtuell* sind. Betrachten wir dazu nochmals Beispiel 41 aus Abschnitt 10.3 und ändern dessen *main*-Funktion wie folgt:

```
main()
{
    var obj = new Base();
    print(obj->getClassName(), "\n");
    obj = new Derived();
    print(obj->getClassName(), "\n");
}
```

Beispiel 43 Virtueller Aufruf einer Methode

⁴⁶Dies trifft zumindest dann zu, wenn der Aufruf wie der einer Elementfunktion, also nicht mit vorangestelltem Klassennamen und dem Operator *::* erfolgt.

Hier werden ein- und derselben Variablen (`obj`) nacheinander Instanzen der Klassen `Base` bzw. `Derived` zugewiesen⁴⁷ und jeweils deren `getClassName()`-Methoden aufgerufen. Dabei wird beim ersten Aufruf die ursprüngliche Version aus `Base` und beim zweiten die *überschriebene* aus `Derived` aktiviert.

Häufig kommt es vor, dass eine abgeleitete Klasse eine Methode ihrer Basisklasse mit dem Ziel überschreibt, deren Funktionalität zu erweitern, sie aber nicht vollständig zu ersetzen. Dann ist es wünschenswert, den Inhalt der geerbten Methode einfach aufzurufen, anstatt ihn vollständig neu zu implementieren. B++ benutzt hierfür eine besondere syntaktische Konstruktion:

Um innerhalb der Implementierung einer überschreibenden Methode die geerbte Vorfahrsmethode aufzurufen, wird dem Funktionsnamen das Präfix `BC_` (für `BaseClass`) vorangestellt.

```
class MyBaseClass
{
    MyBaseClass();
    writeInfo();
}

MyBaseClass::MyBaseClass() {} // ctor does nothing
MyBaseClass::writeInfo() { print("MyBaseClass::writeInfo() called\n"); }

MyDerivedClass : MyBaseClass
{
    MyDerivedClass();
    writeInfo();
}

MyDerivedClass::MyDerivedClass() {} // ctor does nothing
MyDerivedClass::writeInfo()
{
    print("MyDerivedClass::writeInfo() called\n");
    this->BC_writeInfo(); // call inherited writeInfo method
}

var main()
{
    var obj = new MyDerivedClass();
    obj->writeInfo();
    return 0;
}
```

Beispiel 44 Expliziter Aufruf von Methoden einer Basisklasse

Hinweise:

- Der Aufruf der geerbten Vorfahrsmethode muss über den `this`-Zeiger erfolgen (siehe Beispiel 44). Andernfalls würde der Compiler einen „gewöhnlichen“ Funktionsaufruf generieren.
- Der Aufruf mit dem `BC_`-Präfix bezieht sich auf die Basisklasse derjenigen Klasse, die die aufrufende Methode enthält. Analog zum Präfix `BC_` kann das Präfix `DC_` (für *defining class*) verwendet werden. Hier bezieht sich der Aufruf auf die Klasse, in der die aufrufende Methode definiert wurde. Mit seiner Hilfe kann verhindert werden, dass eine sich selbst rekursiv aufrufende Methode, die in einer abgeleiteten Klasse überschrieben und von dort mit `BC_` aufgerufen wird, in der Rekursion wieder die Methode der abgeleiteten Klasse aufruft.

⁴⁷ Bei der Zuweisung der `Derived`-Instanz an `obj` wird die zuvor zugewiesene `Base`-Instanz automatisch freigegeben. Die Freigabe der `Derived`-Instanz erfolgt beim Verlassen der `main`-Funktion.

Beispiel 45 demonstriert den Methodenaufruf innerhalb der Implementierung von Methoden mit und ohne Verwendung von `this` sowie die Verwendung von `BC_` und `DC_`.

```
class ClA
{
}
ClA::ClA() {}
ClA::test() { print("ClA::test\n"); }
ClA::callTest() {
    print("\n",__func__," called from class ",getclassname(this),"\n");
    print("this->DC_test : "); this->DC_test();
    print("this->test(); : "); this->test();
}

class ClB : ClA
{
}
ClB::ClB() {}
ClB::test() { print("ClB::test\n"); }
ClB::callTest() {
    print("\n",__func__," called from class ",getclassname(this),"\n");
    print("test() : "); test();
    try {
        print("this->BC_test : "); this->BC_test();
    }
    catch (e) { print(e,"\n"); }
    print("this->DC_test : "); this->DC_test();
    print("this->test(); : "); this->test();
    print("this->BC_callTest(); : "); this->BC_callTest();
}

class ClC : ClB
{
}
ClC::ClC() {}
ClC::test() { print("ClC::test\n"); }

void main()
{
    var obj = new ClC();
    obj->callTest();
    obj = new ClB();
    obj->callTest();
    obj = new ClA();
    obj->callTest();
}
```

Beispiel 45 Methodenaufreife mit `this`, `BC_` und `DC_`

Der einfache Aufruf der `test`-Methode (jeweils in den `callTest`-Methoden implementiert) führt hier unabhängig davon, ob `this` vorangestellt wird, jeweils zum gleichen Ergebnis. Für die Präfix-Aufrufe ist ein expliziter Objektverweis (`this`) jedoch zwingend notwendig.

10.6 Erweiterte Syntax des Operators `->`

Der Operator `->` dient in B++ ausschließlich dem Aufruf von Instanzmethoden eines Objekts und hat die allgemeine Form

objref->methodname(argumentlist)

Neben dieser Standardform (vgl. Beispiele in Abschnitt 10.5) gibt es eine erweiterte Variante des Aufrufs:

```
objref->(expr) (argumentlist)
```

Hier wird statt des Methodennamens ein Ausdruck angegeben, der als Ergebnis entweder den Namen der aufzurufenden Methode als Zeichenkette oder einen Verweis (im Sinne eines Zeigers) auf den Code der aufzurufenden Methode liefert.

Mit einem Ausdruck, der eine Zeichenkette als Ergebnis liefert, können Namen aufzurufender Methoden zur Laufzeit von Programm bestimmt werden. Das Verhalten des Methodenaufrufs selbst entspricht genau dem der Standardform, insbesondere können auch die Präfixe `BC_` und `DC_` (siehe 10.5) verwendet werden.

Darüber hinaus erlaubt die erweiterte Form die „frühe Bindung“ von Methoden, also deren nicht-virtuellen Aufruf.

Dies erreicht man durch einen Aufruf über

```
this->(classname::methodname) ()
```

Dabei ist `classname` der Name der eigenen Klasse oder einer ihrer Basisklassen, während `methodname` die aufzurufende Methode kennzeichnet.

Damit wird die aufzurufende Methode nicht (wie bei virtuellen Aufrufen) zur Laufzeit bestimmt, sondern der Verweis auf die auszuführende Methode direkt in den Bytecode eingebunden. Dementsprechend müssen die betreffende Klasse und die aufzurufende Methode zum Zeitpunkt der Übersetzung des aufrufenden Codes mindestens deklariert sein.

10.7 Methodenzeiger und Methodenlitterale

Methoden eines Objekts (Elementfunktionen) unterscheiden sich von ‚gewöhnlichen‘ Funktionen nur durch ihren Bezug zu einer Klasse und die Möglichkeit des Zugriffs auf Member-Variablen von Objekten (Instanzen der Klasse). Dementsprechend lassen sich auch Verweise auf Methoden an Variablen zuweisen oder als Argumente von Funktionsaufrufen verwenden. Variablen, denen ein Verweis auf eine Methode zugewiesen wurde, lassen sich – analog zu den Funktionszeigern, vgl. Abschnitt 9.5 – als *Methodenzeiger* auffassen.

Nachfolgendes Beispiel zeigt die Zuweisung einer Elementfunktion an eine Variable sowie Varianten des Aufrufs.

```
class ClassA
{
public:
    ClassA() {}
    Test(i) { print (__func__, "(" , i, ") \n"); }
};

main()
{
    var obj = new ClassA();
    var a = ClassA::Test;
    obj->(a) (1);
    a(obj, 1);
}
```

Beispiel 46 Methodenaufruf über Methodenzeiger

Die Zuweisung an eine Variable erfolgt wie beim Lesezugriff auf eine statische Variable einer Klasse.

Der Aufruf erfolgt entweder über einen Objektverweis mit der erweiterten Form des Operators `->` (siehe auch 10.6) oder wie ein einfacher Funktionsaufruf, wobei hier ein Objekt als zusätzlicher (erster) Parameter übergeben werden muss.

Auch Methoden können als Literale definiert werden, wobei zur Definition die Syntax

`function ['classid'] [returntype] ('paramlist [;localvarlist]')` *body* verwendet wird (vgl. 9.7).

Dabei bezeichnet *classid* eine Klasse, die zum Zeitpunkt der Literaldefinition bereits vorhanden (deklariert) sein muss. Ferner müssen alle zur Klasse gehörenden Variablen, die innerhalb der literalen Methode verwendet werden, definiert sein – siehe Beispiel 47.

```
class A
{
private:
    string _txt;
public:
    A(string txt="hello\n") { _txt = txt; }
}
// define global variable g initialized with member function literal
var g = function [A] string() { return _txt; }

main()
{
    var a = new A();
    print(a->(g)());
}
```

Beispiel 47 Definition einer Methode als Literal

Hinweis:

Das hier gezeigte Beispiel demonstriert eine Möglichkeit der temporären Erweiterung existierender Klassen durch zusätzliche Methoden. Dabei ist zu beachten, dass dadurch der für die Klassenelemente definierte Zugriffsschutz umgangen wird. Deshalb sollte diese Möglichkeit mit Bedacht eingesetzt und ihr Einsatz gut dokumentiert werden.

10.8 Partielle Klassen

B++ unterstützt partielle Klassen in ähnlicher Form wie C#, ohne jedoch dafür ein spezielles Schlüsselwort zu verwenden. Dabei meint *Partielle Klasse* eine Klasse, deren Definition auf mehrere Blöcke, die ggf. auch in unterschiedlichen Quelldateien stehen können, verteilt ist.

Dies eröffnet mehrere Möglichkeiten:

- Verteilen umfangreicher Klassen auf mehrere Dateien
- Behandlung inhaltlich verschiedener Aspekte einer Klasse in getrennten Abschnitten
- Klasseninhalte lassen sich variabel gestalten, d. h. in Abhängigkeit vom jeweiligen Kontext kann eine konkrete Klasse aus unterschiedlichen Teilen zusammengesetzt werden.
- Nutzung zur Vorwärtsdeklaration von zur Kompilierzeit eines Moduls unbekannten Klassen
- Bestehende Klassen können „nachträglich“ erweitert werden.

Für das folgende Beispiel nehmen wir an, dass ein Programm eine Klasse `Class1` verwendet, von der es eine Instanz bildet und darauf eine Methode `doAction` aktiviert. Die Klasse `Class1` selbst wird in einem anderen Modul abschließend definiert und implementiert, welches beim Aufruf mit dem Hauptprogramm kombiniert wird.

Zunächst das Hauptprogramm:

```
/* doAction example - main module action.bpp */

class Class1      // class prototype
{
    doAction();
}

main() {
    var obj = new Class1();
    obj->doAction();
}
```

Class1 wird hier nur als leerer Prototyp definiert⁴⁸ – den fordert der Compiler. Eine erste Version der vollständigen Definition und Implementierung erfolgt in einem Modul actcl1_1:

```
/* first version of Class1 module - actcl1_1.bpp */

class Class1
{
    Class1();                // ctor
    doAction();
    var _callCnt;            // counter for calls of doAction
}
// implementation

Class1::Class1() { _callCnt = 0; }
Class1::doAction()
{
    print(++_callCnt, ". call of doAction in module actcl1_1\n");
}
```

Nun können Hauptprogramm und Modul jeweils in Bytecode übersetzt und anschließend das Hauptprogramm unter Nutzung des Bibliotheksmoduls gestartet werden:

```
bp2 -c action -o action.bpm
bp2 -c actcl1_1 -o actcl1_1.bpm
bp2 -r action actcl1_1
```

Ohne Änderung am Hauptprogramm kann man jetzt die Klasse Class1 in einem anderen Modul erneut implementieren und alternativ zur obigen Variante verwenden:

```
/* second version of Class1 module - actcl1_2.bpp */

class Class1
{
    Class1();                // ctor
    doAction();
}
// implementation

Class1::Class1() {}          // ctor does nothing
Class1::doAction()
{
    print("doAction in module actcl1_2 activated\n");
}
```

Beispiel 48 Verwenden Partieller Klassen

Übersetzung und Aufruf erfolgen analog:

```
bp2 -c actcl1_2 -o actcl1_2.bpm
bp2 -r action actcl1_2
```

⁴⁸ Die Deklaration von doAction hätte man auch weglassen können.

10.9 Ersetzen von Member-Funktionen (Redefinition)

So, wie man bestehende „normale“ Funktionen durch einfache Neudefinition ersetzen kann (siehe 9.6), ist dies auch für Funktionen möglich, die Bestandteil einer Klasse sind. Die neu definierte Variante ersetzt die alte vollständig. Diese Eigenschaft lässt sich zur nachträglichen Änderung des Verhaltens einer Klasse nutzen.

Zur Veranschaulichung dient wieder das Beispiel aus dem vorangegangenen Abschnitt. Jetzt wird jedoch die Klasse `Class1` sofort implementiert.

```
/* doAction example - main module action.bpp */

class Class1
{
public:
    Class1();
    doAction();
}

Class1::Class1() {}
Class1::doAction() { print("Class1::doAction called\n"); }

main()
{
    var obj = new Class1();
    obj->doAction();
}
```

Das Beispiel sollte direkt lauffähig sein.

Nun wird ein Patch-Modul geschrieben, das das Verhalten der `doAction`-Methode aus `Class1` ändert:

```
/* module patch.bpp replaces doAction method of Class1 */
Class1::doAction()
{
    print("patched version of CtClass::doAction called\n");
}
```

Beispiel 49 Redefinition einer Member-Funktion

Zu beachten ist hier, dass das Patch-Modul nicht separat übersetzbar ist, da es keine Deklaration von `Class1` enthält, folgender Aufruf ist aber möglich:

```
bp2 action patch
```

Hier wird der Patch nach dem ursprünglichen Programm übersetzt, so dass `Class1` bereits vorhanden und der Compiler zufrieden ist.

Hinweise:

- Durch die Redefinition einer Methode wird deren Sichtbarkeitsstufe (`public`, `protected`, `private`) nicht verändert.
- Für die Redefinition gelten dieselben Regeln wie für eine „normale“ Implementierung einer Methode außerhalb der Klassendefinition. Insbesondere betrifft dies auch die Notwendigkeit der erneuten Spezifikation von Vorgabeparametern – vgl. Abschnitt 10.1.

10.10 Definieren von Operatoren

In B++ ist es möglich, einige Operatoren für Objekte (oder genauer: deren Klassen) umzudefinieren bzw. überhaupt zu definieren.

Ein Operator wird indirekt definiert, indem eine Klasse eine Methode (Elementfunktion) bereitstellt, deren Name einem der reservierten Bezeichner der Form `OP_XXX` (siehe Kapitel 7) entspricht. Alternativ kann zur Definition auch eine an C++ angelehnte Syntax mit dem Schlüsselwort `operator` und nachgestelltem Operatorsymbol verwendet werden; sie wird wegen ihrer Gebräuchlichkeit in den Beispielen der folgenden Abschnitte bevorzugt.

Hinweise:

- Auch bei der `operator`-Notation erzeugt der Compiler Methoden der Form `OP_XXX`. Deshalb sind explizite Operatoraufrufe nur über diese Methodennamen möglich.
- Wie „normale“ Methoden geben auch Operatorfunktionen einen Wert zurück. Deshalb kann eine Operatordefinition optional mit einem Rückgabebetyp eingeleitet sein.
- Anders als in C++ können in B++ keine Operatoren als globale Funktionen definiert werden, sondern ausschließlich als (nicht statische) Elementfunktionen von Klassen.

Nachfolgend werden die Regeln zum Definieren einzelner Operatoren bzw. Operatorgruppen erklärt.

10.10.1 Definieren des Funktionsoperators

Wenn eine Klasse den Funktionsoperator `()` definiert, werden ihre Instanzen zu Funktionsobjekten (mitunter auch als *Funktoren* bezeichnet). Solche Objekte können innerhalb des Programms wie Funktionen gebraucht werden.

Die Definition des Operators erfolgt in der Form

```
[returntype] OP_CALL(argumentlist) bzw.
```

```
[returntype] operator() (argumentlist),
```

wobei *argumentlist* eine kommaseparierte (und möglicherweise auch leere) Liste von Argumentdefinitionen ist, für die alle Regeln einer Argumentliste „normaler“ Funktionen gelten. Insbesondere können auch Argumentlisten variabler Länge oder Vorgabeparameter verwendet werden – vgl. Abschnitt 9.3.

Beispiel 50 illustriert die Vorgehensweise anhand einer Klasse, deren Instanzen wie Funktionen gebraucht werden, die eine beliebige Anzahl von Zeichenketten in eine Datei schreiben.

```

/* operator () example */
class StringWriter
{
public:
    StringWriter(string file);
    ~StringWriter();
    uint operator() ();
private:
    FILE _fp;
}

StringWriter::StringWriter(string file)
{
    _fp = fopen(file,"wt"); // open textfile for write
}

StringWriter::~~StringWriter()
{
    fclose(_fp);
}

uint StringWriter::operator() ()
{
    uint n = argcnt();
    for (uint i=1;i<n; ++i)
        fputs(arg(i),_fp);
    return n;
}

main()
{
    StringWriter writer("txtfile.txt");
    writer("Hello","World");
}

```

Beispiel 50 Definition des Funktionsoperators für eine Klasse

10.10.2 Definieren des Indexoperators

Der Indexoperator `[]` dient normalerweise dem indizierten Zugriff auf einzelne Elemente einer Zeichenkette oder eines Containers (Typen `vector`, `buffer` oder `charbuffer`) bzw. dem Zugriff auf Einträge eines assoziativen Containers (`dictionary`) über einen Schlüssel. Wird der Operator für Objekte definiert, so sind hierfür zwei Member-Funktionen – eine für den Lesezugriff und eine für den Schreibzugriff – erforderlich⁴⁹. Im Gegensatz zu den Standardvarianten muss das Index-Argument nicht zwingend eine Ganzzahl bzw. eine Zeichenkette sein.

Die Definition des Operators erfolgt in der Form

```

[returntype] OP_VREF(index) oder
[returntype] operator[] (index)

```

für den Lesezugriff bzw.

```

[returntype] OP_VSET(index, value) oder
[returntype] operator[]=(index, value) oder
[returntype] operator[var] (index, value)50

```

⁴⁹ Wird nur eine der Funktionen implementiert, funktioniert der Operator nur in einer Richtung.

⁵⁰ Das Schlüsselwort `var` dient nur der syntaktischen Unterscheidung bei der Operatordefinition. In Anlehnung an die Definition von Postfix-Inkrement-Operatoren in C++ kann statt `var` auch `int` verwendet werden.

für den Schreibzugriff.

Beispiel 51 zeigt ein Fragment einer Klasse `Point`, die einen n-dimensionalen Punkt repräsentiert. Die Koordinaten des Punktes werden mit Hilfe des Indexoperators gelesen und geschrieben.

```
class Point()
{
    Point(uint dim);
    ~Point();
    var operator[]=(uint index, var value);
    var operator[] (uint index);
    vector _coords;           // vector of coordinates
}

Point::Point(uint dim)
{
    _coords = newvector(dim);
    for (var i=0;i<dim;++i) _coords[i] = 0;
}

var Point::operator[]=(uint index, var value) {
    _coords[index] = value;
    return value;
}

var Point::operator[] (uint index) { return _coords[index]; }

void main()
{
    var point = new Point(2);
    point[0] = 12;           // set x-coordinate
    point[1] = 4.7;          // set y-coordinate;
    print("x=",point[0], " y=",point[1],"\n");
}
```

Beispiel 51 Implementierung des Indexoperators

Hinweis:

Die Implementierung der Variante für den Schreibzugriff (`OP_VSET`) sollte – wie in obigem Beispiel – den zugewiesenen Wert (`value`) als Rückgabewert weitergeben, um so die Semantik des Zuweisungsoperators zu erhalten.

10.10.3 Definieren des Punktoperators

Wie schon in Abschnitt 6.7 erwähnt, ist der Punkt in B++ ein „gewöhnlicher“ Operator für den Zugriff auf Elemente assoziativer Container (Datentyp `dictionary`) und auf öffentliche Datenelemente von Objekten, der für Objekte bzw. deren Klassen explizit definiert, d. h. also in seiner Funktionalität erweitert werden kann. Dies ermöglicht es z. B., Zugriffe auf Elemente eines Objektes im Sinne von Properties (wie sie etwa in C# verwendet werden) oder benutzerdefinierte assoziative Container zu implementieren, die ähnlich dem Typ `dictionary` verwendet werden können.

In seiner Semantik ähnelt der Punktoperator dem Indexoperator sehr stark, deshalb wird er auch in ähnlicher Weise definiert.

Die Definition erfolgt in der Form

```
[returntype] OP_PREF(key) oder
[returntype] operator.(key)
```


für den Lesezugriff bzw.

```
[returntype] OP_PSET(key, value) oder  
[returntype] operator.=(key, value)
```

für den Schreibzugriff.

Dabei ist muss *key* stets vom Typ *string* sein und einen gültigen Bezeichner (vgl. 4.2) repräsentieren.

Beispiel 52 Verwendet wieder eine Klasse für einen Punkt (diesmal einen zweidimensionalen) zur Darstellung einer möglichen Implementierung⁵¹.

```
class Point2d  
{  
private:  
    double _x;  
    double _y;  
public:  
    Point2d(double x=0, double y=0) { _x=x; _y=y; }  
    double get_x() { return _x; }  
    void set_x(double val) { _x = val; }  
    double get_y() { return _y; }  
    void set_y(double val) { _y = val; }  
    double operator . (string key){ return this->("get_" + key)(); }  
    double operator .=(string key, double val) {  
        this->("set_" + key)(val);  
        return val;  
    }  
};  
  
void main() {  
    Point2d pt(15,25);  
    pt.x *= 3;  
    pt.y *= 2;  
    print("pt: x=", pt.x, " y=", pt.y, "\n");  
}
```

Beispiel 52 Implementierung des Punktoperators

Hinweis:

Ein benutzerdefinierter Punktoperator hebt das Standardverhalten des Operators für die jeweilige Klasse nicht auf. Das heißt, die benutzerdefinierte Operatorfunktion wird nur aufgerufen, wenn keine im jeweiligen Kontext sichtbare Member-Variable mit dem durch *key* bezeichneten Namen existiert.

10.10.4 Definieren der Inkrement- und Dekrement-Operatoren

Wie in Abschnitt 6.6 erklärt, können die Inkrement- und Dekrement-Operatoren in Präfix- und Postfix-Notation verwendet werden. Entsprechend lassen sich auch beide Varianten für Klassen definieren.

Die Definition der Operatoren erfolgt in der Form

```
OP_INC() bzw. OP_DEC() oder  
operator++() bzw. operator--()
```

für die Präfix-Notation bzw.

⁵¹Darüber hinaus ist die Klasse *Point2d* selbst *inline* implementiert und es wird durchgängig eine statische Typisierung verwendet.

```

OP_PINC() bzw. OP_PDEC() oder
operator++(var) bzw. operator--(var) oder
operator++(int) bzw. operator--(int)

```

für die Postfix-Notation.

Analog zur Notation in C++ wird bei der operator-Variante ein dummy-Argument zur Unterscheidung von Präfix- und Postfix-Notation verwendet.

Beispiel 53 demonstriert die Definition der Operatoren anhand des Auschnitts einer Klasse für Rationale Zahlen.

```

class Rational
{
private:
    int _nom;    // nominator
    uint _denom; // denominator
public:
    Rational(int n=0, uint d=1) { _nom=n; _denom=d; }
    Rational operator++() { _nom += _denom; return this; }
    Rational operator++(int) {
        return new Rational(_nom+_denom,_denom);
    }
    Rational operator--() { _nom -= _denom; return this; }
    Rational operator--(int) {
        Rational r(_nom,_denom);
        return --r;
    }
    operator string() { return newstring("%d/%u",_nom,_denom); }
}

main()
{
    Rational r(1,2);
    print(r,"\n");           // prints 1/2
    print(r++, " -> ", r,"\n"); // prints 1/2 -> 3/2
    print(++r,"\n");         // prints 5/2
}

```

Beispiel 53 Definition der Inkrement- und Dekrement-Operatoren

Die Implementierungen für die Präfix-Notation modifizieren das jeweilige Objekt und liefern das modifizierte Objekt als Ergebnis zurück.

Bei den Implementierungen für die Postfix-Notation wird eine Kopie des ursprünglichen Objekts erzeugt, anschließend modifiziert und als Ergebnis zurückgegeben.

Hinweise:

- Die Implementierung der Postfix-Varianten unterscheidet sich vom aus C++ gewohnten Vorgehen! Dort gibt man eine Kopie des ursprünglichen Objekts zurück.
- Definiert eine Klasse nur die Präfix-Variante, so wird diese Implementierung implizit auch beim Gebrauch des Operators in der Postfix-Form benutzt, umgekehrt gilt dies nicht. Obwohl es der Compiler nicht verlangt, sollten generell Prefix- und Postfix-Varianten der Operatoren paarweise implementiert werden, um ein Verhalten zu erreichen, das dem der Standardtypen entspricht.

10.10.5 Definieren des Dereferenzierungsoperators

Der unäre Dereferenzierungsoperator `*` wird vor allem für die Umwandlung einer Weak Reference in einen „echten“ Wert verwendet (vgl. Abschnitt 6.7. und Kapitel 13). Darüber hinaus kann er für Objekte als Präfix-Operator wie folgt explizit definiert werden:

```
[returntype] OP_UNREF() bzw.  
[returntype] operator*().
```

Der Operator sollte verwendet werden, um ein eingebettetes Objekt oder sonstiges Datum zurückzuliefern, ohne das definierende Objekt selbst zu modifizieren. Ein typischer Anwendungsfall dafür wäre ein Iterator auf einem benutzerdefinierten Container, der das Objekt, auf das der Iterator zeigt, zurückgibt.

10.10.6 Definieren der weiteren unären Operatoren

In Tabelle 15 sind die weiteren unären Operatoren aufgeführt, die für B++-Klassen definiert werden können.

Operator	Funktion	Operator-Notation	Bedeutung
+	OP_PLUS()	operator+()	Positives Vorzeichen
-	OP_NEG()	operator-()	Negatives Vorzeichen (Zweierkomplement)
!	OP_NOT()	operator!()	Logische Negation
~	OP_BNOT()	operator~()	Binäre Negation (Einerkomplement)

Tabelle 14 Weitere definierbare unäre Operatoren

Die Implementierungsregeln für diese Operatoren entsprechen weitgehend denen von C++. Die jeweilige Operatorfunktion sollte das aufrufende Objekt unverändert lassen und eine modifizierte Kopie dieses Objekts zurückgeben.

Eine Implementierung des negativen Vorzeichens für die Klasse `Rational` aus Beispiel 53 könnte etwa so aussehen:

```
Rational operator-() { return new Rational(-_nom, _denom); }
```

Die Operatorfunktionen unärer Operatoren haben stets eine leere Argumentliste. Optional kann ein Rückgabetypp explizit angegeben werden.

10.10.7 Indirekte Definition von Vergleichsoperatoren mit den Methoden `compare` bzw. `equals`

Anders als C++ gestattet B++ keine direkte Definition der Vergleichsoperatoren (siehe Abschnitt 6.2). Indirekt ist dies jedoch möglich, indem eine Klasse die Elementfunktionen

```
[bool] equals([typename] other)
```

und/oder

```
[int] compare([typename] other)
```

definiert, wobei Typspezifikationen für Argumente und Rückgabewerte optional sind.

Die Methode `equals` sollte `true` (bzw. 1) zurückgeben, wenn eine Gleichheit zwischen dem aufrufenden Objekt und `other` vorliegt, andernfalls `false` (bzw. 0).

Die Methode `compare` sollte einen negativen Wert liefern, falls das aufrufende Objekt *kleiner* als `other` ist, einen positiven Wert, falls es *größer* als `other` ist, und 0 im Falle der Gleichheit.

Beide Methoden sollten weder das aufrufende Objekt noch das Vergleichsobjekt `other` verändern.

Wird ein Objekt in einem Ausdruck als linker Operand eines Vergleichsoperators verwendet, so wird in diesem Objekt im Falle der Operatoren `==` und `!=` zunächst nach einer Methode `equals`, für alle anderen Vergleichsoperatoren oder falls `equals` nicht existiert, nach einer Methode `compare` für die Durchführung des Vergleichs gesucht.

Kommt ein Objekt als rechter Operand vor, wird entsprechend verfahren und das Vergleichsergebnis umgekehrt. Sind beide Operanden Objekte, so ‚gewinnt‘ der linke Operand, falls er geeignete Vergleichsmethoden implementiert. Stellt kein Operand eine eigene Vergleichsmethode bereit, gelten die in Abschnitt 6.2 beschriebenen Standard-Vergleichsregeln.

Beispiel 54 erweitert die Klasse `Rational` aus Beispiel 53 um eine `compare`-Methode.

```
class Rational
{
    int _nom; // nominator
    uint _denom; // denominator
    Rational(int n=0, uint d=1) { _nom=n; _denom=d; }
    // other functions ...
    operator string() { return newstring("%d/%u", _nom, _denom); }
    operator double() { return ::double(_nom) / _denom; }
    int compare(var other) {
        double d = double() - ::double(other);
        return d > 0 ? 1 : d < 0 ? -1 : 0;
    }
}

main()
{
    Rational r(1,2);
    print (r<0,r>0,r==0,r!=0,0<r,0>=r); // prints 010110
}
```

Beispiel 54 Implementierung einer `compare`-Methode

In dem Beispiel wird auch eine Typkonvertierung nach `double` verwendet (siehe 10.10.9). Zu beachten ist dabei der explizite Aufruf der globalen Konvertierungsfunktion `double` mit Hilfe des Bereichsauflösungsoperators (`::`), wodurch ein rekursiver Aufruf der Elementfunktion `double` verhindert wird.

10.10.8 Binäre Infix-Operatoren

Bei der Definition der binären Infix-Operatoren ist zu beachten, dass das Objekt (der Operand) auf der linken oder rechten Seite des Operators stehen kann. Weil Operatoren in B++ immer als Elementfunktionen definiert werden, gibt es für jeden Operator zwei Funktionen (siehe Tabelle 15), wobei die Variante für den rechtsseitigen Operanden das Suffix `_R` hat.

Bei der Operator-Notation steht zur Unterscheidung der beiden Formen zwischen dem Schlüsselwort `operator` und dem Operatorsymbol ein `L` (Objekt als linksseitiger Operand) bzw. ein `R` (Objekt als rechtsseitiger Operand), wobei das `L` auch entfallen kann – die linksseitige Variante wird als Standard angenommen.

Operator	Funktion		Operator-Notation
	links	rechts	
Arithmetische Operatoren			
+	OP_ADD(<i>arg</i>)	OP_ADD_R(<i>arg</i>)	operator [L R] + (<i>arg</i>)
-	OP_SUB(<i>arg</i>)	OP_SUB_R(<i>arg</i>)	operator [L R] - (<i>arg</i>)
*	OP_MUL(<i>arg</i>)	OP_MUL_R(<i>arg</i>)	operator [L R] * (<i>arg</i>)
/	OP_DIV(<i>arg</i>)	OP_DIV_R(<i>arg</i>)	operator [L R] / (<i>arg</i>)
%	OP_REM(<i>arg</i>)	OP_REM_R(<i>arg</i>)	operator [L R] % (<i>arg</i>)
Bitweise Operatoren			
	OP_BOR(<i>arg</i>)	OP_BOR_R(<i>arg</i>)	operator [L R] (<i>arg</i>)
&	OP_BAND(<i>arg</i>)	OP_BAND_R(<i>arg</i>)	operator [L R] & (<i>arg</i>)
^	OP_XOR(<i>arg</i>)	OP_XOR_R(<i>arg</i>)	operator [L R] ^ (<i>arg</i>)
Verschiebeoperatoren			
<<	OP_SHL(<i>arg</i>)	OP_SHL_R(<i>arg</i>)	operator [L R] << (<i>arg</i>)
>>	OP_SHR(<i>arg</i>)	OP_SHR_R(<i>arg</i>)	operator [L R] >> (<i>arg</i>)

Tabelle 15 Definierbare Infix-Operatoren in B++

Das Prinzip der Definition ist für alle diese Operatoren gleich und ist in Beispiel 55 für den Additionsoperator veranschaulicht. Hier wird die Klasse `Point` aus 10.10.2 modifiziert⁵² und der Operator `+` so implementiert, dass mit seiner Hilfe ein neuer Punkt erzeugt wird, dessen Koordinaten als Summe der Koordinaten der Operanden gebildet werden, falls der zweite Operand ebenfalls vom Typ `Point` ist. Andernfalls wird versucht, den zweiten Operanden in den Typ `double` zu konvertieren und seinen Wert zu allen Komponenten des ersten Operanden zu addieren, wobei das Ergebnis ebenfalls ein neues Objekt ist.

```
class Point
{
    vector _coords;                // vector of coordinates
    vector coords() { return _coords; } // get coordinates
    Point(uint dim);                // ctor
    Point operator+(var other);      // operator + (left)
    Point operator R +(var other);   // operator + (right)
}

Point::Point(uint dim)
{
    _coords = newvector(dim);
    for (uint i=0; i<dim; ++i) _coords[i] = 0.0;
}

Point Point::operator+(var other)
{
    uint dim = size(_coords);
    if (dynamic_cast(Point, other))
    {
        vector ocoords = other->coords();
        uint odim = size(ocoords);
        if (odim < dim) dim = odim;
        Point result(dim);
    }
}
```

⁵²Die Indexoperatoren werden hier der Kürze halber weggelassen.

```

    vector rcoords = result->coords();
    for (uint i=0;i<dim;++i) rcoords[i] = _coords[i] + ocoords[i];
    return result;
}
double d = other; // implicit conversion
Point res(dim);
vector rc = res->coords()
for (uint j=0;j<dim;++j)
    rc[j] = _coords[j] + d;
return res;
}

Point Point::operator R+(var other)
{
    return OP_ADD(other);
}

```

Beispiel 55 Definition des Additionsoperators

Hinweise:

- Um für benutzerdefinierte Klassen ein Verhalten analog zu den eingebauten Datentypen zu erreichen, sollten die Infix-Operatoren stets paarweise (also für den linken und rechten Operanden) definiert werden.
- Die Operatoren sollten niemals einen oder beide Operanden modifizieren, sondern stets ein neues Objekt zurückliefern. Der Typ des Resultats sollte dem des definierenden Objekts entsprechen.
- Ist das Verhalten eines Operators – wie bei der Addition – symmetrisch, so kann die Implementierung einer Variante die der anderen aufrufen (vgl. `operator R+` in Beispiel 55). Für den Aufruf muss die Funktionsform verwendet werden.
- Der Typ des als Argument übergebenen zweiten Operanden ist häufig (wie auch im Beispiel) nicht statisch festgelegt, um eine Verwendung in gemischten Ausdrücken zu ermöglichen. In solch einem Fall ist die Implementierung der Operatorfunktion für die Durchführung geeigneter Typüberprüfungen bzw. -konvertierungen verantwortlich.
- Die Auswertung der Infix-Operatoren erfolgt von links nach rechts. Das bedeutet, dass zunächst der Typ des linken Operanden betrachtet wird. Handelt es sich um einen Objekttyp, so wird – falls vorhanden – dessen zugehörige linksseitige Operatorfunktion mit dem rechten Operanden als Argument aufgerufen. Ist der rechte Operand ein Objekt, der linke jedoch nicht, so wird die rechtsseitige Operatorfunktion mit dem linken Operanden als Argument aufgerufen.

10.10.9 Operatoren zur Typumwandlung

Wie schon in Abschnitt 5.5 erwähnt, können für Objekte (Klassen) Elementfunktionen zur Umwandlung in andere Datentypen definiert werden.

Syntaktisch sind dafür zwei Varianten zulässig:

```
[type] type()
```

oder

```
operator type().
```

Die erste Form ist eine ‚normale‘ Methodendefinition, wobei der Methodenname (*type*) dem Rückgabetyt entspricht. Optional kann der Rückgabetyt zusätzlich angegeben werden⁵³. Die zweite Form entspricht der C++-Notation für Konvertierungsoperatoren.

Konvertierungsmethoden haben grundsätzlich keine Argumente.

In Beispiel 54 (Abschnitt 10.10.7) werden für die Klasse `Rational` Konvertierungsoperatoren in die Typen `double` und `string` definiert.

Hinweis:

Obwohl B++ keinen echten Boolean-Typ kennt, kann eine Klasse einen Konvertierungsoperator `bool` definieren. Seine Implementierung sollte stets ein Ergebnis vom Typ `int` mit dem Wert 0 für `false` bzw. 1 für `true` liefern.

Zur expliziten Umwandlung von Werten in Objekttypen können auch globale Funktionen verwendet werden, die denselben Namen tragen, wie die Klasse, in deren Typ die Konvertierung erfolgen soll, also die Form

```
[classtype] classtype([type]arg)
```

haben.

Eine solche Funktion sollte stets ein Ergebnis vom Typ der entsprechenden Klasse zurückgeben – oder `null`, wenn eine Konvertierung nicht möglich ist. Ansonsten gibt es keine strengen Implementierungsregeln für derartige Funktionen.

Hinweis:

Technisch ist eine globale Konvertierungsfunktion eine „ganz gewöhnliche“ Funktion, die beliebige Argumente übernehmen, beliebige Aktionen ausführen sowie beliebige Ergebnisse liefern kann, und deren einzige Besonderheit darin besteht, dass ihr Name dem einer Klasse entspricht. Denkbar wäre auch, derartige Funktionen zum Erzeugen von Objekten (an Stelle des Operators `new`), also im Sinne von Factory-Funktionen einzusetzen.

⁵³Er sollte auch angegeben werden – so verspricht die Methode, tatsächlich das zurück zu liefern, was ihr Name erwarten lässt.

11 Vordefinierte Funktionen

Nachfolgend werden die verfügbaren vordefinierten Funktionen im Sinne einer Referenz beschrieben. Dabei erfolgt eine grobe Gruppierung nach den Einsatzbereichen. Die Beschreibungen sind innerhalb der einzelnen Unterabschnitte alphabetisch geordnet.

Hinweise:

- Die vordefinierten Funktionen sind *nicht statisch typisiert*. Somit werden bei der Zuordnung von Werten zu den Argumenten im Allgemeinen *keine impliziten Typumwandlungen* vorgenommen⁵⁴. Sofern einzelne Funktionen oder Funktionsgruppen solche Konvertierungen vornehmen, ist dies bei den jeweiligen Beschreibungen erwähnt.
- Die Implementierungen der Funktionen überprüfen die Gültigkeit der übergebenen Argumente zur Laufzeit und können im Fehlerfall Ausnahmen von Typ `string` auslösen.
- Die Typen der Parameter und Rückgabewerte werden in den jeweiligen Signaturen mit angegeben. Sofern die betreffenden Werte ihrer Semantik nach Wahrheitswerte sind, werden der Pseudo-Typbezeichner `bool` sowie die Werte `true` bzw. `false` verwendet.

11.1 Standardfunktionen für Container-Datentypen

Die hier zusammengefassten Funktionen dienen der Behandlung der vordefinierten Containertypen `buffer`, `charbuffer`, `dictionary` und `vector`.

11.1.1 Container-Konstruktorfunktionen

Diese Funktionen dienen der Erzeugung von Werten der verschiedenen Containertypen.

Funktion `CBuf`

`charbuffer CBuf (...)`

Die Funktion erzeugt einen Zeichenpuffer aus den Werten der Argumentliste.

Parameter:

- `...`: Kommaseparierte Werteliste. Die Elemente der Werteliste können von integralen Typen (Wert wird als Zeichencode interpretiert) sowie vom Typ `buffer`, `charbuffer` oder `string` sein. Argumente anderer Typen werden ignoriert.

Rückgabewert:

Aus den Argumenten erzeugter Zeichenpuffer.

Beispiel:

```
string str = "Hello World!";
charbuffer myCBuf = CBuf(str, 0x20, "Let's ", 'b', 'u', 'i', 'l', "d a buffer!", '\n');
print(string(myCBuf));
```

Beispiel 56 Erzeugen eines Zeichenpuffers aus mehreren Bestandteilen mit `CBuf`

⁵⁴Eine Ausnahme bilden hier die integralen Typen, die stets zueinander zuweisungskompatibel sind, weshalb keine explizite Umwandlung vorgenommen werden muss.

Funktion newbuffer

```
buffer newbuffer(var assignment) oder  
buffer newbuffer(uint size, byte fillbyte=0)
```

Die erste Form erzeugt einen Puffer als Kopie eines anderen Puffers oder des Inhalts einer Zeichenkette.

Bei der zweiten Form wird ein Puffer mit einer festgelegten Größe angelegt und mit einem einheitlichen Wert aller Elemente initialisiert.

Parameter:

- `assignment`: Wert, aus dem der Puffer erzeugt werden soll. Der Wert muss vom Typ `string`, `buffer` oder `charbuffer` sein.
- `size`: Puffergröße in Bytes.
- `fillbyte`: Initialwert der einzelnen Bytes.

Rückgabewert:

Erzeugter Puffer.

Hinweise:

- Für das `fillbyte`-Argument kann ein Wert beliebigen integralen Typs übergeben werden, wobei allerdings nur das niederwertigste Byte übernommen wird.
- In Unicode-Builds wird bei der Übergabe von Zeichenketten als `assignment` (erste Form) nur das niederwertigste Byte jedes Zeichens übernommen.
- Ist das Argument `assignment` vom Typ `string`, so wird das abschließende Null-Byte mit in den Puffer übernommen.

Funktion newcbuffer

```
charbuffer newcbuffer(var assignment) oder  
charbuffer newcbuffer(uint size, int fillchar=0)
```

Die erste Form erzeugt einen Zeichenpuffer als Kopie eines anderen Puffers oder des Inhalts einer Zeichenkette.

Bei der zweiten Form wird ein Zeichenpuffer mit einer festgelegten Größe angelegt und mit einem einheitlichen Wert aller Elemente initialisiert.

Parameter:

- `assignment`: Wert, aus dem der Puffer erzeugt werden soll. Der Wert muss vom Typ `string`, `buffer` oder `charbuffer` sein.
- `size`: Anzahl der Zeichen im erzeugten Puffer.
- `fillchar`: Initialwert der einzelnen Zeichen.

Rückgabewert:

Erzeugter Zeichenpuffer.

Hinweise:

- Für das `fillchar`-Argument kann ein Wert beliebigen integralen Typs übergeben werden, wobei implizit eine Umwandlung in einen Zeichencode (16 Bit in Unicode-Builds, 8 Bit sonst) vorgenommen wird.
- Ist in Unicode-Builds das Argument `assignment` (erste Form) vom Typ `buffer`, so werden die einzelnen Bytes aus dem Puffer ohne Vorzeichen auf 16 Bit erweitert, das höherwertige Byte jedes Zeichens wird also auf Null gesetzt.
- Ist das Argument `assignment` vom Typ `string`, so wird das abschließende Null-Zeichen mit in den Puffer übernommen.

Funktion `newdict`

```
dictionary newdict(bool sorted = false)
```

Die Funktion erzeugt eine neue `dictionary`-Instanz.

Parameter:

- `sorted`: Das Flag bestimmt die Ordnung der Einträge in der `dictionary`-Instanz. Ist der Wert `true`, so sind die Einträge alphanumerisch entsprechend den Schlüsseln geordnet, andernfalls in der Reihenfolge ihres Einfügens.

Rückgabewert:

Neu erzeugte (leere) `dictionary`-Instanz.

Hinweise:

- Der Wert des `sorted`-Flags beeinflusst die Resultate der Funktionen `dictgetkeys` und `dictgetvalues`. Ist es gesetzt (`true`), so werden die Einträge entsprechend der Schlüssel-sortierung, andernfalls in der Reihenfolge des Einfügens zurück gegeben.
- In DOS- und Windows-CE-Builds erfolgt der Zugriff auf Einträge sortierter Dictionaries schneller als auf die unsortierter.
- Dictionaries, die mit dem Literalkonstruktor (`{ ... }`, vgl. 5.2.2) erzeugt werden, sind stets unsortiert.

Funktion `newvector`

```
vector newvector(uint capacity = 0, bool init = true)
```

Die Funktion erzeugt einen neuen Vektor mit reserviertem Speicherplatz für `capacity` Elementen. Optional kann der Vektor mit `null`-Werten initialisiert werden.

Parameter:

- `capacity`: Anzahl der Elemente, für die Speicherplatz reserviert wird.
- `init`: Flag für die Initialisierung, Hat es den Wert `true` (bzw. einen von 0 verschiedenen Wert), wird der Vektor initialisiert, d. h. seine Anfangsgröße entspricht der in `capacity` festgelegten Elementzahl und alle Elemente erhalten den Wert `null`.

Rückgabewert:

Neu erzeugte `vector`-Instanz.

Hinweise:

In B++ gibt es keinen Weg, die Kapazität eines Vektors nachträglich explizit zu ändern. Eine Vergrößerung ist implizit durch Einfügen von Elementen (Funktion `vininsert`) möglich, mit Hilfe der Funktion `shrink` kann die Kapazität auf die für die aktuelle Elementzahl erforderliche Größe reduziert werden.

Die Angabe einer Anfangsgröße und die Initialisierung können die Ausführungszeit deutlich verkürzen, da keine Notwendigkeit zur Größenänderung zur Laufzeit besteht.

Beispiel:

```
// initialized vector
vector vec1 = newvector(10);
for (var i=0;i<10;++i) vec1[i-1] = i*i;
// vector with initial capacity but without initialization
vector vec2 = newvector(10,false);
for (var j=0;j<10;++j) vininsert(vec2,-1,j*j);
// initially empty vector
vector vec3 = newvector();
for (var k=0;k<10;++j) vininsert(vec3,-1,k*k);
```

Beispiel 57 Erzeugen und Initialisieren von Vektoren mit `newvector`

Funktionen `T` und `Vec`

`vector T(...)` oder

`vector Vec(...)`

Beide Funktionen sind synonym verwendbar. Sie konstruieren einen neuen Vektor, dessen Elemente die übergebenen Argumente in der angegebenen Reihenfolge sind.

Parameter:

- ...: Kommaseparierte Liste beliebiger Werte

Rückgabewert:

Neu erzeugte `vector`-Instanz.

Beispiel:

```
/* T() example */
main(;vec,i)
{
    // make a vector using strsplit-function
    vec = strsplit("This is a text.", " ");
    for(i=0;i<size(vec);++i)
        print(vec[i], "\n");
    /* make new vector using T-Function
       Note that element at index 3 is the vector constructed above. */
    vec = T(1,2,size(vec),vec,"TEST");
    for(i=0;i<size(vec);++i)
        print(vec[i], "\n");
}
```

Beispiel 58 Erzeugen eines Vektors aus einer heterogenen Elementliste

11.1.2 Allgemeine Containerfunktionen

Die in diesem Abschnitt beschriebenen Funktionen sind auf mehrere Containerarten anwendbar.

Funktion `foreach`

```
uint foreach(var container, var callback, ...)
```

Die Funktion iteriert über einen Container und ruft für jedes Element einen Handler (Funktion oder Objekt), jeweils mit Schlüssel und Wert des Elements sowie ggf. zusätzlichen Parametern auf. .

Parameter:

- `container`: Container, der durchlaufen werden soll.
Zulässige Argumenttypen sind `buffer`, `charbuffer`, `vector`, `dictionary`, `string` oder ein Objekt, dessen Klasse eine Elementfunktion `foreach` der Form

```
uint foreach(var callback, ...)
```

definiert.

- `callback`: Callback für die Aufzählung der Elemente in `container`. Zulässig ist hier eine Funktion oder ein Funktor-Objekt (ein Objekt, das `operator()` definiert) in der Form

```
bool func(var key, var value, ...).
```

Liefert ein Aufruf von `callback` als Ergebnis `false`, wird die Iteration durch `container` abgebrochen.

- `...`: Wird `foreach` mit mehr als zwei Parametern aufgerufen, werden die zusätzlichen Argumente für jedes Element an den Aufruf von `callback` weitergegeben.

Rückgabewert:

Anzahl der in Elemente in `container`, für die `callback` aufgerufen wurde.

Beispiel:

```
void main()
{
    // fill sorted dictionary from an unsorted one
    auto dict = { t:'t', e:'e',m:'m',p:'p',o:'o'}; // unsorted
    dictionary dict2 = newdict(true); // sorted
    // use dict2 for additional element
    foreach(dict,
        function(i,v,d) { d[i] = v; return true; },
        dict2);
    // callback for print container items
    auto iter = function(i, v) { print("I: ",i," V: ",v,"\n"); return true;};
    foreach(dict,iter); // list dict
    foreach(dict2,iter); // list dict2
    // enumerate a string
    foreach("Hello World",iter);
}
```

Beispiel 59 Verwenden von `foreach` zur Manipulation und Ausgabe von Containern

In Beispiel 59 wird `foreach` zuerst zum Kopieren eines unsortierten Dictionarys (`dict`) in ein sortiertes (`dict2`) mit einer anonymen Funktion für `callback` verwendet, der `dict2` als zusätzliches Argument übergeben wird. Anschließend erfolgt die Ausgabe der beiden Container sowie einer Zeichenkette unter Verwendung einer anderen Funktion für `callback`.

Hinweise:

- Ist `container` vom Typ `dictionary`, ist das erste Argument für `callback` der Schlüssel des jeweiligen Eintrags, also eine Zeichenkette. Für alle anderen Standardcontainer wird als Schlüssel der numerische Index des jeweiligen Elements im Container übergeben.
- Das zweite Argument für `callback` ist stets der jeweilige Elementwert. Handelt es sich um einen Wert von einem Referenztyp, kann der Wert innerhalb von `callback` modifiziert werden.
- Wird innerhalb von `callback` die Elementzahl in `container` geändert, werden also Elemente gelöscht oder hinzugefügt, sollte der jeweilige Aufruf von `callback` als Ergebnis `false` liefern um die Iteration abzubrechen.
- Wenn `container` ein Objekt ist, wird die `foreach`-Methode des betreffenden Objekts mit aufgerufen, wobei `callback` sowie die zusätzlichen Argumente an die Methode weitergeleitet werden. Sie sollte demnach die Form

```
uint foreach(var callback, ...)
```

haben. Besitzt das Objekt keine Implementierung von `foreach`, wird ein Laufzeitfehler erzeugt.

Funktion `qsort`

```
void qsort(var container, var callback = null)
```

Die Funktion sortiert den Inhalt eines Containers mit dem Quicksort-Algorithmus. Das Verhalten entspricht weitgehend dem der gleichnamigen Funktion des C-Standardbibliothek.

Parameter:

- `container`: Container des Typs `vector`, `buffer` oder `charbuffer`, dessen Inhalt sortiert werden soll.
- `callback`: Vergleichsfunktion oder Funktor-Objekt der Form

```
int cmpfunc(var left, var right)
```

`callback` wird für den Vergleich zweier Elemente aus `container` aufgerufen. Ist `callback` nicht angegeben, wird ein Standardvergleich der Werte durchgeführt.

Beispiel:

```
class comparer {
    comparer() {}
    operator()(a,b) { return b-a; } // makes descending order
};

void main() {
    var vec = [3,1,5,7,2,0,9];
    var sz = size(vec);
    // sort ascending using anonymous function for callback
    qsort(vec,function(a,b) { return a-b; });
    for (var i=0;i<sz;++i) print(vec[i]); // prints 0123579
    print("\n");
    // sort descending using comparer object
    qsort(vec,new comparer());
    for (i=0;i<sz;++i) print(vec[i]); // prints 9753210
}
```

Beispiel 60 Sortieren eines Vektors mit `qsort`

Beispiel 60 illustriert die Verwendung von `qsort` zum Sortieren eines Vektors in aufsteigender und absteigender Folge mit Hilfe unterschiedlicher `callback`-Implementierungen.

Hinweis:

Der Standardvergleich von Elementen entspricht der Verwendung der in Abschnitt 6.2 beschriebenen Vergleichsoperatoren und realisiert eine aufsteigende Sortierung.

Funktion `shrink`

```
uint shrink(var container)
```

Die Funktion reduziert die Kapazität eines Containers (reservierter Speicher) auf die entsprechend der tatsächlich enthaltenen Elemente notwendige Größe.

Parameter:

- `container`: Container, dessen Kapazität reduziert werden soll.
Zulässige Argumenttypen sind `buffer`, `charbuffer` und `vector`.

Rückgabewert:

Anzahl der in Elemente, für die in `container` Speicherplatz reserviert ist.

Funktion size

```
uint size(var container)
```

Die Funktion bestimmt die Anzahl der in einem Container enthaltenen Elemente.

Parameter:

- `container`: Container, dessen Größe (Anzahl der Elemente) bestimmt werden soll. Zulässige Argumenttypen sind `buffer`, `charbuffer`, `vector`, `dictionary` oder `string`.

Rückgabewert:

Anzahl der in `container` enthaltenen Elemente.

11.1.3 Vektor-Funktionen

Die Funktionen dieses Abschnitts beziehen sich auf Datenobjekte des Typs `vector`.

Funktion find

```
int find(vector vec, var searchval, int pos = 0)
```

Die Funktion durchsucht `vec` nach dem Wert `searchval` und liefert die Position von dessen erstem Auftreten als Ergebnis.

Parameter:

- `vec`: zu durchsuchender Vektor
- `searchval`: Wert, nach dem in `vec` gesucht wird
- `pos`: Startposition für die Suche

Rückgabewert:

Position (Index) des ersten Auftretens von `searchval` in `vec` bzw. -1, falls der Wert nicht gefunden wurde.

Hinweis:

Die Funktion führt einen Low-Level-Vergleich der Werte durch. Das bedeutet, dass keine ggf. vorhandenen Vergleichsmethoden für Objekte (`equals` oder `compare` – vgl. 10.10.7) aufgerufen werden.

Funktion rfind

```
int rfind(vector vec, var searchval, int pos = -1)
```

Die Funktion durchsucht `vec` rückwärts nach dem Wert `searchval` und liefert die Position von dessen letztem Auftreten als Ergebnis.

Parameter:

- `vec`: zu durchsuchender Vektor
- `searchval`: Wert, nach dem in `vec` gesucht wird
- `pos`: Startposition für die Suche; ist `pos` negativ oder größer als der höchstmögliche Index in `vec`, beginnt die Suche am Ende.

Rückgabewert:

Position (Index) des letzten Auftretens von `searchval` in `vec` bzw. -1, falls der Wert nicht gefunden wurde.

Hinweis:

Die Funktion führt einen Low-Level-Vergleich der Werte durch. Das bedeutet, dass keine ggf. vorhandenen Vergleichsmethoden für Objekte (`equals` oder `compare` – vgl. 10.10.7) aufgerufen werden.

Funktion `vclear`

```
void vclear(vector vec)
```

Die Funktion löscht alle Elemente des Vektors `vec`.

Parameter:

- `vec`: Vektor

Funktion `vecsize`

```
int vecsize(vector vec)
```

Die Funktion liefert die Anzahl der Elemente von `vec`

Parameter:

- `vec`: Vektor

Rückgabewert:

Anzahl der Elemente von `vec`

Funktion `verase`

```
vector verase(vector vec, uint pos = 0, int cnt = -1)
```

Die Funktion löscht, beginnend bei Index `pos`, höchstens die durch `cnt` gegebene Anzahl von Elementen aus Vektor `vec`.

Parameter:

- `vec`: Vektor, aus dem Elemente entfernt werden sollen

- `pos`: Index des ersten zu entfernenden Elements
- `cnt`: Anzahl der zu entfernenden Elemente. Ist `cnt` negativ oder größer als die Anzahl der vorhandenen Elemente ab `pos`, so werden alle Elemente ab Index `pos` entfernt.

Rückgabewert:

Vektor `vec` nach der Modifikation.

Funktion `vinset`

```
vector vinset (vector vec, int pos, var insertion, uint cnt = 1)
```

Die Funktion fügt `cnt`-mal `insertion` an der Indexposition `pos` in den Vektor `vec` ein.

Parameter:

- `vec`: Vektor, in den neue Elemente eingefügt werden sollen.
- `pos`: Indexposition in `vec`, an der das Einfügen erfolgt. Ist `pos` negativ oder größer als der größte aktuelle Index in `vec`, so erfolgt das Einfügen am Ende von `vec`.
- `insertion`: einzufügender Wert
- `cnt`: Anzahl der Elemente, die in `vec` eingefügt werden sollen.

Rückgabewert:

Vektor `vec` nach der Modifikation.

Funktion `vmove`

```
vector vmove (vector vec, uint srcpos, uint dstpos)
```

Die Funktion verschiebt das Element mit dem Index `srcpos` innerhalb des Vektors `vec` an die Stelle `dstpos`.

Parameter:

- `vec`: Vektor, in den neue Elemente eingefügt werden sollen.
- `srcpos`: Indexposition des zu verschiebenden Elements in `vec`.
- `dstpos`: Indexposition in `vec`, an die das Element verschoben werden soll.

Rückgabewert:

Vektor `vec` nach der Modifikation.

Beispiel:

```
vector arr = [1,2,3,4,5,6];
vmove(arr,4,1);
// arr is now [1,5,2,3,4,6]
```

Hinweis:

Im Falle eines ungültigen Indexwertes in *srcpos* oder *dstpos* löst die Funktion eine Ausnahme vom Typ `string` aus.

Funktion `vslice`

```
vector vslice (vector vec, uint from = 0, int cnt = -1)
```

Die Funktion erzeugt einen neuen Vektor aus einer Teilfolge der in *vec* enthaltenen Elemente. Der Vektor *vec* selbst bleibt dabei unverändert.

Parameter:

- *vec*: Vektor, aus dem eine Teilfolge extrahiert werden soll.
- *from*: Indexposition des ersten Elements in *vec*, Bestandteil der Teilfolge ist.
- *cnt*: Anzahl der Elemente aus *vec*, die in die Teilfolge übernommen werden sollen. Ist *cnt* negativ oder größer als die Anzahl der vorhandenen Elemente ab *from*, so werden alle Elemente ab Index *from* in das Ergebnis aufgenommen.

Rückgabewert:

Neue `vector`-Instanz mit der aus *vec* gebildeten Teilfolge.

Funktion `vswap`

```
vector vswap (vector vec, uint pos1, uint pos2)
```

Die Funktion vertauscht die Elemente an den Indexpositionen *pos1* und *pos2* innerhalb des Vektors *vec*.

Parameter:

- *vec*: Vektor, auf den sich die Operation bezieht
- *pos1*: Indexposition des ersten Elements
- *pos2*: Indexposition des zweiten Elements

Rückgabewert:

Vektor *vec* nach der Modifikation.

Beispiel:

```
vector arr = [1,2,3,4,5,6];  
vswap(arr,4,1);  
// arr is now [1,5,3,4,2,6]
```

Hinweis:

Im Falle eines ungültigen Indexwertes für *pos1* oder *pos2* wird eine Ausnahme vom `string`-Typ ausgelöst.

11.1.4 Dictionary-Funktionen

Die Funktionen dieses Abschnitts beziehen sich auf Datenobjekte des Typs `dictionary`.

Funktion `dictclear`

```
null dictclear (dictionary dict)
```

Die Funktion löscht alle Elemente des assoziativen Containers *dict*.

Parameter:

- `dict`: Dictionary

Rückgabewert:

`null`

Funktion `dictcontains`

```
bool dictcontains (dictionary dict, string key)
```

Die Funktion prüft, ob ein Eintrag mit dem Schlüssel *key* im Dictionary *dict* enthalten ist.

Parameter:

- `dict`: Dictionary, in dem nach *key* gesucht werden soll.
- `key`: Schlüssel, nach dem in *dict* gesucht wird.

Rückgabewert:

`true` (1), falls ein Eintrag mit dem Schlüssel *key* in *dict* enthalten ist, andernfalls `false` (0).

Funktion `dicterase`

```
bool dicterase (dictionary dict, string key)
```

Die Funktion entfernt den Eintrag mit dem Schlüssel *key* aus dem Dictionary *dict*.

Parameter:

- `dict`: Dictionary, aus dem der Eintrag *key* entfernt werden soll.
- `key`: Schlüssel des Eintrags, der aus *dict* entfernt wird.

Rückgabewert:

`true` (1), falls ein Eintrag mit dem Schlüssel *key* aus *dict* entfernt wurde (darin enthalten war), andernfalls `false` (0).

Funktion dictgetkeys

```
vector dictgetkeys (dictionary dict)
```

Die Funktion liefert einen Vektor mit den Schlüsseln aller in *dict* enthaltenen Elemente.

Parameter:

- *dict*: Dictionary

Rückgabewert:

Vektor mit den Schlüsseln der in *dict* enthaltenen Elemente

Hinweise:

- Der Ergebnisvektor enthält Kopien der Schlüssel des Containers. Änderungen innerhalb dieses Vektors durch den aufrufenden Code haben deshalb keinen Einfluss auf das Dictionary.
- Der Ergebnisvektor selbst kann an Stelle eines Iterators verwendet werden, um die Einträge des Containers *dict* zu durchlaufen – siehe Beispiel 61.
- Die Reihenfolge der Elemente des Ergebnisvektors hängt davon ab, ob der Container sortiert ist oder nicht (vgl. Funktion *newdict* in Abschnitt 11.1.1). In einem sortierten Dictionary sind die Schlüssel alphabetisch sortiert, andernfalls sind sie in der Reihenfolge ihres Einfügens in den Container geordnet.

Beispiel:

```
dictionary myDict = newdict(); // create unsorted dictionary
// add any entries here
// ...
vector keyvec = dictgetkeys(myDict);
uint n = size(keyvec);
for (uint i=0; i<n;++i)
    print(keyvec[i], ":" , myDict[keyvec[i]], "\n");
```

Beispiel 61 Verwenden von dictgetkeys zum Iterieren durch ein Dictionary

Funktion dictgetvalues

```
vector dictgetvalues (dictionary dict)
```

Die Funktion liefert einen Vektor mit den Werten aller in *dict* enthaltenen Elemente.

Parameter:

- *dict*: Dictionary

Rückgabewert:

Vektor mit den Werten der in *dict* enthaltenen Elemente

Hinweis:

Die Reihenfolge der Werte im Ergebnisvektor entspricht der Schlüsselreihenfolge – siehe Funktion `dictgetkeys`.

Beispiel:

```
dictionary myDict = newdict(); // create unsorted dictionary
// add any entries here
// ...
vector keyvec = dictgetkeys(myDict)
vector valvec = dictgetvalues(myDict);
uint n = size(keyvec); // same as size(valvec)
for (uint i=0; i<n;++i)
    print(keyvec[i], ":", valvec[i], "\n");
```

Beispiel 62 Verwenden von `dictgetvalues`

11.2 Typkonvertierungen und RTTI

C++ kennt keinen `cast`-Operator. Deshalb werden explizite Typkonvertierungen (vgl. Abschnitt 5.5) mit Hilfe von Funktionen durchgeführt, wobei der Funktionsname jeweils dem Bezeichner des Zieltyps entspricht.

C++ stellt einen Satz von Standardfunktionen zur expliziten Konvertierung in Werte der eingebauten Datentypen zur Verfügung. Sofern diese Funktionen mit Objekten als Parameter aufgerufen werden, wird der Aufruf an deren zugehörigen (gleichnamigen) Typumwandlungs-Operator (vgl. 10.10.9) weitergeleitet.

Ferner stellt C++ einige Funktionen zur Ermittlung von Typinformationen zur Laufzeit (RTTI) bereit.

11.2.1 Funktionen zur Typkonvertierung

Funktion `bool`

```
int bool(var expr)
```

Die Funktion bildet aus einem Wert beliebigen Typs einen booleschen Wert.

Parameter:

- `expr`: Ausdruck beliebigen Ergebnistyps

Rückgabewert:

0 (`false`) für alle numerischen Werte vom Betrag Null, den Wert **null** und alle Referenztypen mit dem Datenwert NULL (also ohne gültigen Wert);
sonst 1 (`true`).

Hinweis:

Falls `expr` ein Objekt repräsentiert, dessen Klasse einen `bool()`-Operator definiert, so wird dieser Operator aufgerufen.

Funktionen zur Konvertierung in integrale Typen

```
sbyte sbyte(var expr)
byte byte(var expr)
short short(var expr)
ushort ushort(var expr)
int int(var expr)
uint uint(var expr)
long long(var expr)
ulong ulong(var expr)
intptr intptr(var expr)
```

Alle hier aufgeführten Funktionen konvertieren einen Wert in den entsprechenden integralen Typ. Das Verhalten dieser Funktionen ist bis auf den Rückgabetyt identisch, weshalb sie hier gemeinsam behandelt werden.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei kann `expr` von einem beliebigen numerischen Typ, vom Typ `buffer` oder `string` bzw. ein Objekt sein.

Rückgabewert:

In den Zieltyp konvertierter Wert.

Hinweise:

- Ein Wert vom Typ `string` muss die Repräsentation einer Ganzzahl in dezimaler, oktaler, hexadezimaler oder dualer Schreibweise enthalten. Hexadezimalzahlen werden durch die Präfixe `,0x'` oder `,0X'`, Dualzahlen durch `,0b'` oder `,0B'` gekennzeichnet. Oktalzahlen beginnen mit `,0'`, gefolgt von einer Ziffer zwischen 1 und 7. In allen anderen Fällen wird eine Dezimaldarstellung angenommen.
- Wenn `expr` vom Typ `buffer` ist, muss der Puffer einen ganzzahligen Wert von 8, 16, 32 oder 64 Bits enthalten (die Größe des Puffers muss also 1, 2, 4 oder 8 sein).
- Ist `expr` von einem Objekttyp, so muss dessen Klasse einen entsprechenden Konvertierungsoperator definieren.
- Falls die Konvertierung nicht durchgeführt werden kann (im Falle eines unzulässigen Typs von `expr`), wird eine Ausnahme von Typ `string` ausgelöst.

Funktionen zur Konvertierung in Gleitkommatypen

```
double double(var expr)
float float(var expr)
```

Die hier aufgeführten Funktionen konvertieren einen Wert in den Typ `double` bzw. `float`. Das Verhalten dieser Funktionen ist bis auf den Rückgabetyt identisch, weshalb sie hier gemeinsam behandelt werden.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei kann `expr` von einem beliebigen numerischen Typ, vom Typ `buffer` oder `string` bzw. ein Objekt sein.

Rückgabewert:

In den Zieltyp konvertierter Wert.

Hinweise:

- Ein Wert vom Typ `string` sollte die Repräsentation einer Zahl in der Form `[whitespaces] [sign] [digits] [.digits] [(e | E) [sign] digits]` enthalten. Intern wird zur Konvertierung die Funktion `strtod` der C-Bibliothek verwendet, so dass zusätzliche plattformspezifische Erweiterungen möglich sein können.
- Wenn `expr` vom Typ `buffer` ist, muss der Puffer einen 32-Bit-Wert (`float`) oder einen oder 64-Bit-Wert (`double`) enthalten (die Größe des Puffers muss also 4 oder 8 sein).
- Ist `expr` von einem Objekttyp, so muss dessen Klasse mindestens einen der Konvertierungsoperatoren `float` bzw. `double` definieren.
- Falls die Konvertierung nicht durchgeführt werden kann (im Falle eines unzulässigen Typs von `expr`), wird eine Ausnahme von Typ `string` ausgelöst.

Funktion `buffer`:

```
buffer buffer(var expr)
```

Die Funktion konvertiert den Argumentwert in einen Wert vom Typ `buffer`.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei muss `expr` vom Typ `buffer`, `string` oder `charbuffer`, ein numerischer Wert oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von `expr`.

Hinweise:

- Hat `expr` einen numerischen Typ, so enthält der als Ergebnis gelieferte `buffer` eine bitweise Kopie des Arguments und seine Länge entspricht der Größe der internen Repräsentation des Argumentwertes in Bytes.
- Ist `expr` vom Typ `charbuffer` oder `string`, werden alle darin enthaltenen Zeichen (einschließlich des abschließenden Null-Zeichens) in das Ergebnis kopiert. Die Größe des Ergebnisuffers in Bytes entspricht demnach der Anzahl der kopierten Zeichen, multipliziert mit der Anzahl der Bytes zur Darstellung eines einzelnen Zeichens (1 oder 2).
- Ist `expr` selbst vom Typ `buffer`, wird einfach der Argumentwert (keine Kopie davon!) zurückgegeben.
- Wird als Argument ein Objekt übergeben, so muss dessen Klasse einen `buffer`-Konvertierungsoperator definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion `charbuffer`:

```
charbuffer charbuffer(var expr)
```

Die Funktion konvertiert den Argumentwert in einen Wert vom Typ `charbuffer`.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei muss `expr` vom Typ `buffer`, `string` oder `charbuffer`, ein numerischer Wert oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von `expr`.

Hinweise:

- Hat `expr` einen numerischen Typ (außer `intptr`), so enthält der als Ergebnis gelieferte `charbuffer` eine Zeichenkettenrepräsentation des Wertes in Dezimaldarstellung. Für Gleitkommazahlen wird automatisch eine Darstellung mit oder ohne Exponent gewählt, so dass eine möglichst kurze Zeichenkette als Ergebnis entsteht.
- Ist `expr` vom Typ `intptr`, so ist das Ergebnis eine hexadezimale Adressdarstellung (entsprechend der C-Funktion `printf` mit dem Formatflag `%p`).
- Ist `expr` vom Typ `buffer`, werden alle darin enthaltenen Bytes als einzelne Zeichen interpretiert und in das Ergebnis kopiert. Außerdem wird an das Ergebnis stets ein Null-Zeichen angehängt.
- Ist `expr` vom Typ `string`, wird die entsprechende Zeichenkette in das Ergebnis kopiert.
- Ist `expr` selbst vom Typ `charbuffer`, wird einfach der Argumentwert (keine Kopie davon!) zurückgegeben.
- Wird als Argument ein Objekt übergeben, so muss dessen Klasse einen `charbuffer`-Konvertierungsoperator definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion `dictionary`:

```
dictionary dictionary(var expr)
```

Die Funktion konvertiert den Argumentwert in einen Wert vom Typ `dictionary`.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei muss `expr` vom Typ `vector` bzw. `dictionary` oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von `expr`.

Hinweise:

- Ist *expr* vom Typ `vector`, so enthält das Ergebnis die Elemente des Argumentvektors, während die Schlüssel aus den `string`-Repräsentationen der Indizes gebildet werden.
- Ist *expr* selbst vom Typ `dictionary`, wird einfach der Argumentwert (keine Kopie davon!) zurückgegeben.
- Wird als Argument ein Objekt übergeben, so muss dessen Klasse einen `dictionary`-Konvertierungsoperator definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion `string`:

```
string string(var expr)
```

Die Funktion konvertiert den Argumentwert in einen Wert vom Typ `string`.

Parameter:

- *expr*: Zu konvertierender Wert. Dabei muss *expr* vom Typ `string` oder `charbuffer`, ein numerischer Wert oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von *expr*.

Hinweise:

- Hat *expr* einen numerischen Typ (außer `intptr`), so enthält der als Ergebnis gelieferte `string` eine Zeichenkettenrepräsentation des Wertes in Dezimaldarstellung. Für Gleitkommazahlen wird automatisch eine Darstellung mit oder ohne Exponent gewählt, so dass eine möglichst kurze Zeichenkette als Ergebnis entsteht.
- Ist *expr* von Typ `intptr`, so ist das Ergebnis eine hexadezimale Adressdarstellung (entsprechend der C-Funktion `printf` mit dem Formatflag `%p`).
- Ist *expr* vom Typ `charbuffer`, werden alle darin enthaltenen Zeichen in das Ergebnis kopiert.
- Ist *expr* selbst vom Typ `string`, wird als Ergebnis eine Kopie (!) des Argumentwertes erzeugt.
- Wird als Argument ein Objekt übergeben, so muss dessen Klasse einen `string`-Konvertierungsoperator oder eine Methode `toString()` zur Konvertierung definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion `vector`:

```
vector vector(var expr)
```

Die Funktion konvertiert den Argumentwert in einen Wert vom Typ `vector`.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei muss `expr` vom Typ `vector`, `buffer`, `charbuffer` bzw. `dictionary` oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von `expr`.

Hinweise:

- Ist `expr` vom Typ `buffer`, so wird jedes im Argument enthaltene Byte auf ein Element des Ergebnisvektors vom Typ `int` abgebildet.
- Ist `expr` vom Typ `charbuffer`, so wird jedes im Argument enthaltene Zeichen auf ein Element des Ergebnisvektors vom Typ `int` abgebildet.
- Ist `expr` vom Typ `dictionary`, so wird der Ergebnisvektor aus den im Argument enthaltenen Werten gebildet (das Ergebnis entspricht dem des Aufrufs von `dictgetvalues`, vgl. 11.1.4).
- Ist `expr` selbst vom Typ `vector`, wird einfach der Argumentwert (keine Kopie davon!) zurückgegeben.
- Wird als Argument ein Objekt übergeben, so muss dessen Klasse einen `vector`-Konvertierungsoperator definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

11.2.2 RTTI-Funktionen

Funktion `dynamic_cast`:

```
object dynamic_cast(class dest, var expr)
```

```
object dynamic_cast(string classname, var expr)
```

Die Funktion verhält sich ähnlich dem `dynamic_cast`-Operator in C++. Zunächst wird der Ausdruck `expr` ausgewertet. Ist das Ergebnis ein Objekt der Klasse `dest` (bzw. der durch `classname` spezifizierten Klasse) oder einer ihrer Ableitungen, so wird dieses Ergebnis, in allen anderen Fällen `null` zurückgegeben.

Parameter:

- `dest`: Klasse, zu der die Zugehörigkeit von `expr` geprüft werden soll.
- `classname`: Name der Klasse, zu der die Zugehörigkeit von `expr` geprüft werden soll.
- `expr`: Ausdruck beliebigen Typs

Rückgabewert:

Ergebnis von `expr` bei erfolgreichem Test, sonst `null`

Hinweis:

Im Gegensatz zu C++ wird kein Laufzeitfehler erzeugt, wenn *expr* kein Objekt ist.

Beispiel:

```
/* dynamic_cast example */
class Base
{
    Base() {}
}

class Derived : Base
{
    Derived() {}
}

void main(;obj1,obj2,o,i)
{
    obj1 = new Base();
    obj2 = new Derived();
    o = dynamic_cast(Base, obj1);           // returns obj1
    o = dynamic_cast(Base, obj2);           // returns obj2
    o = dynamic_cast(Derived, obj1);         // returns null
    o = dynamic_cast(Derived, obj2);         // returns obj2
    o = dynamic_cast("Derived", obj2);       // returns obj2
    o = dynamic_cast(getclassname(obj1), obj2); // returns obj2
    i = dynamic_cast(Base, 17);              // returns null (not 0)
}
```

Beispiel 63 Verwenden von dynamic_cast

Funktion getclassname:

```
string getclassname(var expr)
```

Die Funktion wertet den Ausdruck *expr* aus und gibt den Klassennamen des Resultats zurück, falls *expr* ein Objekt oder eine Klasse ergibt.

Parameter:

- *expr*: Ausdruck des Typs `class` oder `object`

Rückgabewert:

Klassename von *expr* bei Erfolg, sonst `null`

Hinweis:

Das Ergebnis ist eine Referenz (keine Kopie!) des in der zu *expr* gehörigen Klasse enthaltenen Klassennamens. Deshalb darf der Aufrufer den Ergebnisstring keinesfalls modifizieren. Ist dies gewünscht, sollte unbedingt mit Hilfe der Funktion `string` (siehe 11.2.1) zunächst eine Kopie des Ergebnisses erzeugt werden.

Funktion getsymbol

```
var getsymbol(string name, bool isclass = false)
```

Die Funktion liefert – sofern es existiert – zur Laufzeit das globale Symbol mit dem Bezeichner `name`.

Parameter:

- `name`: Name (Bezeichner) des Symbols, nach dem gesucht werden soll.
- `isclass`: Das Flag gibt an, in welcher Symboltabelle gesucht wird. Ist sein Wert `true`, wird nach einer Klasse gesucht, andernfalls nach einem Symbol anderen Typs (Funktion, globale Variable).

Rückgabewert:

Bei Erfolg wird der Wert des gefundenen Symbols zurückgegeben (also z. B. eine Funktion oder Klasse), andernfalls ist das Ergebnis `null`.

Hinweise:

- Der eigentliche Zweck der Funktion besteht darin, das Schreiben von Programmen zu ermöglichen, die auf unterschiedlichen Systemplattformen lauffähig sein sollen und gleichzeitig (unterschiedliche) systemspezifische (vordefinierte) Funktionen verwenden. Die Bestimmung der spezifischen Symbole zur Laufzeit verhindert hier Compilerfehler, die sonst wegen nicht definierter Symbole auftreten würden.
- Spezifiziert `name` eine globale Variable eines Referenztyps, kann das aufrufende Programm den Inhalt dieser Variablen manipulieren. Dies ist allerdings eher ein Seiteneffekt und sollte in einem realen Programm nicht ausgenutzt werden.
- Ein Rückgabewert `null` ist insofern nicht eindeutig als er einerseits ein nicht vorhandenes Symbol, andererseits aber auch eine globale Variable, die momentan den Wert `null` besitzt, bedeuten kann. Für echte (nicht anonyme) Funktionen und Klassen spielt dies jedoch keine Rolle, da sie niemals den Wert `null` haben können.

Beispiel 64 illustriert die Verwendung von `getsymbol` in einer Funktion `getCwd`, die das aktuelle Arbeitsverzeichnis ermittelt. Weil B++ hierfür keine vordefinierte Funktion bereitstellt, werden systemspezifische Funktionen (unter MS-DOS ein Interruptaufruf und unter Windows die API-Funktion `GetCurrentDirectory`, deren Adresse mit `GetKernel32Func` bestimmt wird) verwendet. Im Beispiel wird zwischen den Betriebssystemen zunächst mit Hilfe der entsprechenden vordefinierten Literale (vgl. Abschnitt 4.3.8) unterschieden. Die Funktion `getsymbol` wird verwendet, um zur Laufzeit in Abhängigkeit von der Plattform, auf der das Programm *übersetzt* wird, die jeweiligen Systemfunktionen zu referenzieren. Wollte man das Programm so modifizieren, dass es als vorübersetzter Bytecode, also ohne Neuübersetzung, auf verschiedenen Plattformen ablauffähig ist, müsste zur Unterscheidung der Plattform die vordefinierte Variable `__OSTYPE__` (siehe Abschnitt 5.6) statt der Literale verwendet werden.

```
// Gets current working directory
string getCwd()
{
    string buf = null;
    if (MSDOS)
    {
        buf = newstring(256);
        function segread = getsymbol("segread");
        function setreg = getsymbol("setreg");
        function int86x = getsymbol("int86x");
        segread();
        setreg("ah", 0x47);
        setreg("dl", 0);
        intptr pbuf = addr(buf);
        setreg("ds", pbuf >> 16);
        setreg("si", pbuf & 0xFFFF);
        int86x(0x21);
        return buf;
    }
    else if (WIN32)
    {
        function GetFunc = getsymbol("GetKernel32Func");
        intptr pfunc = GetFunc("GetCurrentDirectory?");
        uint siz = CallLibFunc(pfunc, 0, 0);
        buf = newstring(siz);
        CallLibFunc(pfunc, siz, addr(buf));
        return buf;
    }
    throw "Invalid OS";
}
```

Beispiel 64 Anwendung von `getsymbol` für die Benutzung plattformspezifischer Funktionen

Funktion `gettype`:

```
int gettype (var expr)
```

Die Funktion wertet den Ausdruck *expr* aus und gibt dessen Typ zurück.

Parameter:

- *expr*: beliebiger Ausdruck

Rückgabewert:

Datentyp-ID von *expr*

Hinweise:

- Das Ergebnis kann mit `gettypename` in eine lesbare Darstellung umgewandelt werden.
- Die ermittelte ID ist zum Typvergleich mit anderen Ausdrücken verwendbar.

Funktion `gettypename`:

```
string gettypename(int typeid, bool prefixed=false)
```

Die Funktion ermittelt den Namen des durch *typeid* spezifizierten Typs.

Parameter:

- `typeid`: ID des Datentyps, dessen Name ermittelt werden soll.
- `prefixed`: Das Flag gibt an, ob dem Ergebnis das Präfix ‚BVT_‘ vorangestellt werden soll, um eine Zeichenkettendarstellung des internen symbolischen Namens des Typs zu erhalten.

Rückgabewert:

Typname von `typeid`. Tabelle 16 listet die möglichen Werte auf:

Type-ID	Typname (Ergebnis)	Datentyp
0	NULL	null
1	VOID	void
2	AUTO	auto
64	SBYTE	sbyte
65	BYTE	byte
68	SHORT	short
69	USHORT	ushort
72	INT	int
73	UINT	uint
74	INTPTR	intptr
76	LONG	long
77	ULONG	ulong
104	FLOAT	float
108	DOUBLE	double
128	FILE	FILE
129	CODE	‚native‘ Funktion
130	VAR	Variable, Dictionary-Eintrag (intern)
160	OBJECT	object
161	STRONGOBJECT	streng typisiertes Objekt
168	CLASS	class
176	STRING	string
177	BUFFER	buffer
178	CHARBUFFER	charbuffer
180	VECTOR	vector
181	BYTECODE	B++-Funktion
182	LOCALVARINFO	Interne Information zu lokalen Variablen
184	DICTIONARY	dictionary
224	OBJECTREF	freie Referenz auf object
232	CLASSREF	freie Referenz auf class
240	STRINGREF	freie Referenz auf string
241	BUFFERREF	freie Referenz auf buffer
242	CHARBUFFERREF	freie Referenz auf charbuffer
244	VECTORREF	freie Referenz auf vector
246	BYTECODEREF	freie Referenz auf B++-Funktion
248	DICTIONARYREF	freie Referenz auf dictionary

Tabelle 16 Namen der Datentypen in B++

Hinweise:

- Bei der expliziten Angabe `typeid`-Arguments sollten die symbolischen Namen gegenüber dem Zahlenwert bevorzugt werden (vgl. Tabelle 2, S. 16).
- Die ID eines Datentyps kann mit der Funktion `gettype` ermittelt werden.

11.3 Ein-/Ausgabe

Die Funktionen dieses Abschnitts dienen der dateibasierten Ein- und Ausgabe von Daten. Sie orientieren sich weitgehend an den I/O-Funktionen der Standard-C-Bibliothek.

Funktion `fclose`:

```
int fclose(FILE fp)
```

Die Funktion schließt eine zuvor mit `fopen` geöffnete Datei.

Parameter:

- `fp`: Handle der zu schließenden Datei

Rückgabewert:

Bei Erfolg 0, sonst Wert ungleich 0

Hinweis:

Die Funktion darf nicht mit den Standard-Streams `stdin`, `stdout` und `stderr` verwendet werden.

Funktion `feof`:

```
int feof(FILE fp)
```

Die Funktion testet, ob das Ende der angegebenen Datei erreicht ist.

Parameter:

- `fp`: Handle der zu überprüfenden Datei

Rückgabewert:

Wert ungleich 0, wenn Dateiende erreicht, sonst 0

Funktion `fgets`:

```
string fgets(FILE fp)
```

Die Funktion liest eine Zeichenkette bzw. Zeile aus der in `fp` angegebenen geöffneten Textdatei. Das Lesen endet, wenn ein Zeilenumbruch oder das Dateiende erreicht ist.

Parameter:

- `fp`: Handle einer im Textmodus zum Lesen geöffneten Datei

Rückgabewert:

Bei Erfolg wird die gelesene Zeile als Zeichenkette zurückgegeben. Bei Fehler oder erreichtem Dateiende ist das Ergebnis eine leere Zeichenkette.

Hinweise:

- Das Erreichen des Dateiendes sollte mit der Funktion `feof` geprüft werden.
- Ein ggf. aus der Datei gelesener Zeilenumbruch (`'\n'`) ist nicht Bestandteil des Ergebnisses.

Funktion `fgetws`

```
string fgetws(FILE fp)
```

Die Funktion liest eine Wide-Character-Zeichenkette (UTF-16) bzw. Zeile aus der in `fp` angegebenen geöffneten Textdatei. Das Lesen endet, wenn ein Zeilenumbruch oder das Dateiende erreicht ist.

Parameter:

- `fp`: Handle einer im Textmodus zum Lesen geöffneten Datei

Rückgabewert:

Bei Erfolg wird die gelesene Zeile als Zeichenkette zurückgegeben. Bei Fehler oder erreichtem Dateiende ist das Ergebnis eine leere Zeichenkette.

Hinweise:

- Die Funktion ist in der MS-DOS-Version von B++ nicht verfügbar.
- Das Erreichen des Dateiendes sollte mit der Funktion `feof` geprüft werden.
- Ein ggf. aus der Datei gelesener Zeilenumbruch (`'\n'`) ist nicht Bestandteil des Ergebnisses.

Funktion `fopen`:

```
FILE fopen(string filename, string mode)
```

Die Funktion versucht, eine Datei mit dem in `filename` angegebenen Namen im durch `mode` spezifizierten Modus zu öffnen. Sie entspricht in ihrem Verhalten der gleichnamigen C-Funktion.

Parameter:

- `filename`: Name der zu öffnenden Datei. Hier ist die Angabe eines einfachen Dateinamens oder eines Dateipfades (absolut oder relativ) möglich.
- `mode`: Der Parameter beschreibt den Öffnungsmodus der Datei. Dabei sind folgende Modi möglich:
- `r`: Öffnen einer existierenden Datei zum Lesen.

- `w`: Anlegen einer Datei und Öffnen zum Schreiben. Existiert die angegebene Datei bereits, so wird sie überschrieben.
- `a`: Öffnen einer Datei zum Schreiben ab dem aktuellen Dateiende (anhängen von Daten). Ist die Datei noch nicht vorhanden, so wird sie neu angelegt.
- `r+`: Öffnen einer vorhandenen Datei zum Lesen und Schreiben.
- `w+`: Anlegen einer Datei und Öffnen zum Lesen und Schreiben. Existiert die angegebene Datei bereits, so wird sie überschrieben.
- `t`: Öffnen der Datei im Textmodus (kombinierbar mit `a`, `r`, `r+`, `w` und `w+`).
- `b`: Öffnen der Datei im binären Modus (kombinierbar mit `a`, `r`, `r+`, `w` und `w+`).

Rückgabewert:

`FILE`-Handle bei Erfolg, sonst `null`

Hinweis:

Mit dieser Funktion geöffnete Dateien sollten stets mit `fclose` geschlossen werden.

Funktion `fputs`:

```
int fputs(string text, FILE fp)
```

Die Funktion schreibt die Zeichenkette in `text` in die durch `fp` angegebene geöffnete Textdatei.

Parameter:

- `text`: zu schreibende Zeichenkette
- `fp`: Handle der Datei

Rückgabewert:

Bei Erfolg wird der Zeichencode des zuletzt geschriebenen Zeichens, andernfalls der Code von `EOF` (-1) zurückgegeben.

Funktion `fputws`

```
int fputws(string text, FILE fp)
```

Die Funktion schreibt die Zeichenkette in `text` in die durch `fp` angegebene geöffnete Textdatei, wobei für die Ausgabe eine Wide-Character-Darstellung (UTF16) verwendet wird.

Parameter:

- `text`: zu schreibende Zeichenkette
- `fp`: Handle der Datei

Rückgabewert:

Bei Erfolg wird der Zeichencode des zuletzt geschriebenen Zeichens, andernfalls der Code von `EOF` (-1) zurückgegeben.

Hinweis:

Die Funktion ist in der MS-DOS-Version von B++ nicht verfügbar.

Funktion fread

```
uint fread(buffer readbuf, FILE fp)  
uint fread(buffer readbuf, uint cnt, FILE fp)  
uint fread(buffer readbuf, uint size, uint cnt, FILE fp)
```

Diese Funktionen lesen Daten aus einer geöffneten (Binär-)Datei ab der aktuellen Dateiposition und hängen sie an den Inhalt von *readbuf* an.

Die erste Variante liest dabei den gesamten Inhalt bis zum Erreichen des Dateiendes, die zweite höchstens *cnt* Bytes und die dritte höchstens *cnt* Elemente der Größe *size* (in Bytes).

Parameter:

- *readbuf*: Puffer, der als Ziel der Leseoperation verwendet wird
- *size*: Größe eines zu lesenden Elements in Bytes (für die beiden ersten Varianten implizit 1)
- *cnt*: Anzahl der höchstens zu lesenden Elemente
- *fp*: Handle einer geöffneten Binärdatei

Rückgabewert:

Anzahl der vollständig gelesenen Elemente.

Hinweise:

- Im Falle eines Lesefehlers wird keine Ausnahme generiert. Für den Test auf einen Lesefehler kann die Funktion `errno` verwendet werden, für die Prüfung auf Erreichen des Dateiendes die Funktion `feof`.
- Die Kapazität des Lesepuffers *readbuf* wird bei Bedarf automatisch vergrößert.

Funktion freadval:

```
var freadval(FILE fp)
```

Die Funktion liest einen (beliebigen) Wert aus einer Binärdatei und gibt ihn als Ergebnis zurück. Das Format der Datei muss dem von der Funktion **fwriteval** erzeugten entsprechen.

Parameter:

- *fp*: Handle einer geöffneten Binärdatei

Rückgabewert:

Bei Erfolg wird der gelesene Wert, andernfalls `null` zurückgegeben..

Funktion fwrite

```
uint fwrite(buffer buf, FILE fp)
uint fwrite(buffer buf, uint cnt, FILE fp)
uint fwrite(buffer buf, uint size, uint cnt, FILE fp)
```

Diese Funktionen schreiben Daten aus *buf* in eine geöffnete (Binär-)Datei ab der aktuellen Dateiposition.

Die erste Variante schreibt den gesamten Inhalt von *buf*, die zweite höchstens *cnt* Bytes und die dritte höchstens *cnt* Elemente der Größe *size* (in Bytes).

Parameter:

- *buf*: Puffer, der die zu schreibenden Daten enthält
- *size*: Größe eines zu schreibenden Elements in Bytes (für die beiden ersten Varianten implizit 1)
- *cnt*: Anzahl der höchstens zu schreibenden Elemente
- *fp*: Handle einer geöffneten Binärdatei

Rückgabewert:

Anzahl der vollständig geschriebenen Elemente.

Hinweis:

Im Falle eines Schreibfehlers wird keine Ausnahme generiert. Für den Test auf einen Schreibfehler kann die Funktion `errno` verwendet werden.

Funktion fwriteval:

```
int fwriteval(var value, FILE fp)
```

Die Funktion schreibt einen (beliebigen) Wert in eine geöffnete Binärdatei.

Parameter:

- *value*: beliebiger Wert
- *fp*: Handle einer geöffneten Binär-Datei

Rückgabewert:

Bei Erfolg 1, sonst 0

Hinweis:

Mit Hilfe der Funktionen `fwriteval` und `freadval` lassen sich komplexe Datenstrukturen – insbesondere auch Objekthierarchien – serialisieren und deserialisieren. Dabei werden bei der Deserialisierung Querverweise zwischen Objekten automatisch wiederhergestellt.

Beispiel 65 demonstriert die Verwendung der Funktionen `fwriteval` und `freadval` zur Serialisierung einer verketteten Liste. Es kann auch als Beispiel zur Verwendung statisch typisierter Objekte (siehe Abschnitt 5.3) und zur Implementierung des Punkt-Operators (vgl. 10.10.3) dienen.

```
/* Serialization example using fwriteval and freadval */
class ListEntry          // define a simple list class
{
    static uint _maxID=0;
    var _a, _b;
    uint _ID;
    ListEntry _next = null;
    ListEntry(var a=null, var b=null) {
        _a=a;
        _b=b;
        _ID = ++_maxID;
    }
    operator.(string key) {
        switch (key) {
            case "a": return _a;
            case "b": return _b;
            case "next": return _next;
        }
        throw "Invalid key " + key;
    }
    var operator.=(string key, var val) {
        switch (key) {
            case "a": _a=val;
            case "b": _b=val;
            default: throw "Invalid key "+key;
        }
        return val;
    }
    void append(ListEntry entry) {
        if (_next) _next->append(entry);
        else _next=entry;
    }
    void pr() { print("ID: ", _ID, " ", _a, " ", _b, "\n"); }
}

void main()
{
    ListEntry lst(-1,0);
    // append more items to lst
    for(int i=0; i<9; ++i) lst->append(new ListEntry(i,i+1));
    for (ListEntry l=lst; l; l = l.next) l->pr();    // print contents
    string fname = getenv("TEMP")+"/obj.dat";
    FILE fp=fopen(fname,"wb"); // open file for write
    fwriteval(lst,fp);         // write entire list to file
    fclose(fp);
    lst = null;                // free list
    print(lst == null, "\n");   // should print "1"
    fp=fopen(fname,"rb");      // open file for read
    lst = freadval(fp); // read contents and assign to lst
    fclose(fp);
    for (l = lst; l; l=l.next) l->pr(); // print contents
}
```

Beispiel 65 Verwenden von `fwriteval` und `freadval` zur Serialisierung von Daten

Funktion getc:

```
int getc()  
int getc(FILE* fp)
```

Die Funktion liest ein einzelnes Zeichen von der Standardeingabe oder aus der durch *fp* spezifizierten Datei.

Parameter:

- *fp*: Handle der Datei

Rückgabewert:

Zeichencode des gelesenen Zeichens. Bei Fehler oder Dateiende wird EOF zurückgegeben.

Hinweis:

Die Variante ohne Parameter setzt im Normalfall die Existenz einer Konsole voraus. Eine B++-Ausführungsumgebung kann allerdings ein eigenes Eingabeobjekt (Implementierung der Schnittstelle `IBppInput`, siehe Referenz) installieren, um eine andere Datenquelle als `stdin` zu verwenden.

Funktion gets:

```
string gets()
```

Die Funktion liest eine Zeichenkette von der Standardeingabe. Das Lesen endet bei Eingabe eines Zeilenwechsels. Der abschließende Zeilenumbruch (`'\n'`) ist nicht im Ergebnis enthalten.

Rückgabewert:

Bei Erfolg wird die gelesene Zeile als Zeichenkette andernfalls eine leere Zeichenkette.

Hinweis:

Die Funktion setzt im Normalfall die Existenz einer Konsole voraus. Eine B++-Ausführungsumgebung kann allerdings ein eigenes Eingabeobjekt (Implementierung der Schnittstelle `IBppInput`, siehe Referenz) installieren, um eine andere Datenquelle als `stdin` zu verwenden.

Funktion getwc:

```
int getwc(FILE* fp)
```

Die Funktion liest ein einzelnes Wide-Character-Zeichen (UTF-16) aus der durch *fp* spezifizierten Datei.

Parameter:

- *fp*: Handle der Datei

Rückgabewert:

Zeichencode des gelesenen Zeichens. Bei Fehler oder Dateiende wird EOF zurückgegeben.

Hinweis:

Die Funktion ist in der MS-DOS-Version von B++ nicht verfügbar.

Funktion print:

```
void print(...)
```

Die Funktion schreibt die Werte der übergebenen Argumente auf die Standardausgabe.

Parameter:

Kommaseparierte Liste von Argumenten beliebigen Typs. Die Argumente werden in der angegebenen Reihenfolge ausgegeben. Die Art der Ausgabe hängt dabei wie folgt vom Datentyp ab:

- Zeichenketten werden unverändert ausgegeben.
- Werte der integralen Typen werden als unformatierte Dezimazahlen ausgegeben.
- Werte der Typen `double` und `float` werden als unformatierte Double-Werte (wie mit dem Formatflag `%g` der C-Funktion `printf`) ausgegeben.
- Werte des Typs `intptr` werden als hexadezimale Adressen dargestellt.
- Bei allen anderen Typen werden Typname und Adresse ausgegeben.

Hinweis:

Die Funktion setzt im Normalfall die Existenz einer Konsole voraus. Eine B++-Ausführungsumgebung kann allerdings ein eigenes Ausgabeobjekt (Implementierung der Schnittstelle `IBppPrinter`, siehe Referenz) installieren, um ein anderes Ausgabeziel als `stdout` zu verwenden.

Funktion putc:

```
int putc(int c, FILE* fp)
```

Die Funktion schreibt ein einzelnes Single-Byte-Zeichen in die durch `fp` spezifizierte Datei.

Parameter:

- `c`: Zeichencode des zu schreibenden Zeichens
- `fp`: Handle der Datei

Rückgabewert:

Zeichencode des geschriebenen Zeichens. Bei einem Fehler wird `EOF` (-1) zurückgegeben.

Funktion `putwc`:

```
int putwc(int c, FILE* fp)
```

Die Funktion schreibt ein einzelnes Wide-Character-Zeichen (UTF-16) in die durch `fp` spezifizierte Datei.

Parameter:

- `c`: Zeichencode des zu schreibenden Zeichens
- `fp`: Handle der Datei

Rückgabewert:

Zeichencode des geschriebenen Zeichens. Bei einem Fehler wird `EOF` (-1) zurückgegeben.

Hinweis:

Die Funktion ist in der MS-DOS-Version von B++ nicht verfügbar.

11.4 Zeichenkettenfunktionen

Funktion `newstring`:

Die Funktion `newstring` existiert in drei verschiedenen Varianten, die sich in ihrem Verhalten grundlegend unterscheiden. Sie werden deshalb nachfolgend einzeln beschrieben.

```
string newstring(var assignment)
```

Es wird ein neuer String als Kopie eines anderen Strings oder eines Pufferinhalts erzeugt.

Parameter:

- `assignment`: Wert, aus dem die neue Zeichenkette erzeugt wird. Zulässig sind Argumente der Typen `string`, `buffer` oder `charbuffer`.

Rückgabewert:

neu erzeugter String

Hinweise:

- Im Unterschied zur Konvertierungsfunktion `string` (vgl. 11.2.1) führt `newstring` keine implizite Konvertierung von Objekten durch.

- Ist *assignment* vom Typ *buffer*, so wird jedes darin enthaltene Byte als Zeichencode interpretiert und in das Ergebnis aufgenommen.

```
string newstring(uint size, int fillchar=' ')
```

Es wird eine Zeichenkette aus *size* Zeichen erzeugt, die jeweils mit dem Wert von *fillchar* initialisiert werden.

Parameter:

- *size*: Anzahl der Zeichen (ohne abschließendes '\0'-Zeichen)
- *fillchar*: Zeichen, das zur Initialisierung verwendet wird

Rückgabewert:

neu erzeugter String

```
string newstring(string format, ...)
```

Die Funktion erzeugt einen neuen String aus einer Format-Spezifikation und einer nachfolgenden kommaseparierten Werteliste.

Parameter:

- *format*: Formatstring. Die Syntax entspricht der für die C-Funktion `printf`, mit der Einschränkung, dass die Variablenspezifikation '%n' und Präzisionsangabe '*' nicht unterstützt werden.
- *...*: Kommaseparierte Argumentliste zur Ersetzung der Variablen in *format*.

Rückgabewert:

neu erzeugter String

Hinweis:

Die in der Argumentliste verwendeten Argumente sollten im Normalfall numerische Werte (einschließlich des Typs `intptr`) oder Zeichenketten sein. Für Werte anderer Typen wird die Adresse des zugehörigen Datenbereichs (Ergebnis der Funktion `addr()`) übergeben.

Funktion strcmp:

```
int strcmp(string s1, string s2)
```

Die Funktion vergleicht die Zeichenketten *s1* und *s2* zeichenweise, wobei zwischen Groß- und Kleinschreibung unterschieden wird. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- *s1*: erster Operand

- `s2`: zweiter Operand

Rückgabewert:

`< 0`, falls `s1 < s2`
`0`, falls `s1 == s2`
`> 0`, falls `s1 > s2`

Funktion `strerase`

```
string strerase(string str, uint pos, int cnt = -1)
```

Die Funktion löscht ein oder mehrere Zeichen aus der Zeichenkette `str`.

Parameter:

- `str`: Zeichenkette auf die sich die Operation bezieht
- `pos`: Index des ersten zu löschenden Zeichens in `str`
- `cnt`: Anzahl der zu löschenden Zeichen

Rückgabewert:

Modifizierte Zeichenkette nach dem Löschen.

Hinweise:

- Ist `cnt` negativ oder größer als die Anzahl der hinter `pos` verbleibenden Zeichen, wird der gesamte Inhalt ab `pos` bis zum Ende von `str` gelöscht.
- Der Rückgabewert ist der modifizierte Wert von `str`, keine Kopie.

Funktion `strfind`

```
int strfind(string str, string search, uint start = 0)
int strfind(string str, int search, uint start = 0)
```

Die Funktion sucht, beginnend beim Index `start`, in `str` nach dem ersten Auftreten einer Teilzeichenkette oder eines einzelnen Zeichens und gibt den Index der Fundstelle zurück.

Parameter:

- `str`: Zeichenkette auf die sich die Operation bezieht
- `search`: Teilzeichenkette oder Zeichen, nach der/dem gesucht werden soll
- `start`: Indexposition in `str`, an der die Suche beginnt

Rückgabewert:

Indexposition von `search` in `str` bei Erfolg, sonst `-1`.

Funktion **stricmp**:

```
int stricmp(string s1, string s2)
```

Die Funktion vergleicht die Zeichenketten *s1* und *s2* zeichenweise, wobei zwischen Groß- und Kleinschreibung *nicht* unterschieden wird. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- *s1*: erster Operand
- *s2*: zweiter Operand

Rückgabewert:

```
< 0, falls s1 < s2  
0, falls s1 == s2  
> 0, falls s1 > s2
```

Hinweis:

Zum Vergleich werden die einzelnen Elemente der Argumente im Kleinbuchstaben umgewandelt. Wenn *s1* und *s2* Zeichen mit einem ASCII-Code > 127 (z. B. Umlaute) vorkommen, hängt das Verhalten von den Spracheinstellungen des ausführenden Betriebssystems ab.

Funktion **strinsert**

```
string strinsert (string str, uint pos, string insstr)  
string strinsert (string str, uint pos, int insc, uint cnt = 1)
```

Die Funktion fügt eine Zeichenkette oder ein einzelnes Zeichen an der Indexposition *pos* in die Zeichenkette *str* ein.

Parameter:

- *str*: Zeichenkette, auf die sich die Operation bezieht
- *pos*: Indexposition, an der die Zeichenkette oder das Zeichen eingefügt werden soll
- *insstr*: Zeichenkette, die in *str* eingefügt werden soll
- *insc*: Zeichen, das in *str* eingefügt werden soll
- *cnt*: Anzahl der Zeichen *insc*, die in *str* eingefügt werden sollen

Rückgabewert:

Modifizierte Zeichenkette nach dem Löschen.

Hinweise:

- Ist *pos* größer als der Index des letzten Zeichens von *str*, so erfolgt ein Anfügen an das Ende von *str*.
- Der Rückgabewert ist der modifizierte Wert von *str*, keine Kopie.

Funktion strlen:

```
uint strlen(string s)
```

Die Funktion liefert die Länge der Zeichenkette *s*. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- *s*: Zeichenkette

Rückgabewert:

Anzahl der Zeichen in *s*

Funktion strlwr:

```
string strlwr(string s)
```

Die Funktion konvertiert alle Großbuchstaben in *s* in Kleinbuchstaben.

Parameter:

- *s*: Zeichenkette

Rückgabewert:

Modifizierte Zeichenkette nach der Umwandlung

Hinweis:

Das Verhalten der Funktion hängt von den aktuellen Lokalisierungseinstellungen ab, die bei Bedarf mit Hilfe der Funktion `setlocale` eingestellt werden können.

Funktion strreplace

```
string strreplace(string str, string search, string replacement)
```

```
string strreplace(string str, int search, int replacement)
```

Die Funktion ersetzt alle Vorkommen der Teilzeichenkette oder des Zeichens *search* in *str* durch den Wert von *replacement*.

Parameter:

- *str*: Zeichenkette, auf die sich die Operation bezieht
- *search*: Zeichenkette oder Zeichen, nach der/dem gesucht werden soll
- *replacement*: Zeichenkette oder Zeichen, durch die/das *search* ersetzt werden soll

Rückgabewert:

Modifizierte Zeichenkette nach dem Ersetzen.

Hinweis:

Der Rückgabewert ist der modifizierte Wert von *str*, keine Kopie.

Funktion `strsplit`:

```
vector strsplit(string source, string delimiter,  
    bool storeEmptyTokens = true, bool storeDelimiterChar = true)
```

Die Funktion zerlegt die in *source* übergebene Zeichenkette in Teilketten entsprechend den in *delimiter* angegebenen Trennzeichen.

Parameter:

- *source*: zu zerlegende Zeichenkette
- *delimiter*: Zeichenkette mit einem oder mehreren Trennzeichen
- *storeEmptyTokens*: Wenn `true` werden leere Tokens mit im Ergebnis gespeichert.
- *storeDelimiterChar*: Wenn `true` werden die jeweiligen Trennzeichen mit im Ergebnis gespeichert.

Rückgabewert:

Die Funktion gibt einen Vektor zurück, der die einzelnen Teilzeichenketten (Token).sowie ggf. die Trennzeichen enthält.

Hinweise:

- Wenn *source* leer ist, ist auch der Ergebnisvektor leer.
- Trennzeichen werden (falls *storeDelimiterChar* gesetzt ist) als Zeichencodes (Typ `int`) im Ergebnis abgelegt.
- Wenn *storeEmptyTokens* gesetzt ist und *source* mit einem Trennzeichen endet, so wird an das Ergebnis eine leere Zeichenkette angehängt.

Beispiel:

```
/* strsplit example */  
void listvec(vector v)  
{  
    int n = size(v);  
    print("size: ",n,"\n");  
    for (int i=0;i<n;++i) print(""+v[i],"\n");  
    print("-----\n");  
}  
  
main() {  
    string str = "This is a sample string."  
    listvec(strsplit(str, " ."));  
    listvec(strsplit(str, " .", false, true));  
    listvec(strsplit(str, " .", true));  
    listvec(strsplit(str, " .", false, true));  
    listvec(strsplit(str, " .", true, true));  
}
```

Beispiel 66 Verwenden von `strsplit`

Funktion `strupr`:

```
string strupr(string s)
```

Die Funktion konvertiert alle Kleinbuchstaben in `s` in Großbuchstaben.

Parameter:

- `s`: Zeichenkette

Rückgabewert:

Modifizierte Zeichenkette nach der Umwandlung

Hinweis:

Das Verhalten der Funktion hängt von den aktuellen Lokalisierungseinstellungen ab, die bei Bedarf mit Hilfe der Funktion `setlocale` eingestellt werden können.

Funktion `substr`

```
string substr(string str, uint start, int len = -1)
```

Die Funktion liefert eine Teilzeichenkette aus `str`.

Parameter:

- `str`: Zeichenkette, aus der ein Ausschnitt extrahiert werden soll
- `start`: Index des ersten Zeichens der Teilzeichenkette in `str`
- `len`: Maximallänge der Teilzeichenkette

Rückgabewert:

Neue Zeichenkette mit dem angegebenen Ausschnitt aus `str`.

Hinweise:

- Ist `len` negativ oder größer als die Anzahl der in `str` vorhandenen Zeichen ab der Position `start`, so werden alle Zeichen aus `str`, beginnend ab `start`, in das Ergebnis übernommen.
- Die Quellzeichenkette `str` wird nicht geändert.

11.5 Datum und Uhrzeit

Funktion `ctime`:

```
string ctime(long t)
```

Die Funktion erzeugt aus der numerischen Datums- und Zeitangabe in *t* eine druckbare Zeichenkette. Sie verhält sich wie die gleichnamige C-Funktion.

Parameter:

- *t*: numerischer Datums- und Zeitwert (wie *time_t* in C)

Rückgabewert:

Zeichenkettendarstellung von *t* in der Form
Sat May 10 21:52:54 2008

Hinweis:

Unter MS-DOS und Windows-CE ist das Argument vom Typ *int*.

Funktion *localtime*:

`vector localtime(long t)`

Die Funktion erzeugt aus der numerischen Datums- und Zeitangabe in *t* einen Vektor mit den Bestandteilen der lokalisierten Zeitangabe.

Parameter:

- *t*: numerischer Datums- und Zeitwert (wie *time_t* in C)

Rückgabewert:

Vektor aus neun ganzzahligen Elementen entsprechend der *tm*-Struktur aus C mit folgender Bedeutung:

Index	0	1	2	3	4	5	6	7	8
Wert	sec	min	hour	mday	mon	year	wday	yday	isdst

Dabei erfolgt die Jahresangabe (*year*) relativ zum Jahr 1900. Das Feld *isdst* hat den Wert 1, wenn sich die Zeitangabe auf Sommerzeit bezieht, sonst 0.

Hinweise:

- Informationen über den Tag innerhalb des Jahres (*yday*) und die Sommerzeit (*isdst*) werden von Windows CE nicht unterstützt. Deshalb haben hier beide Felder stets den Wert -1.
- Unter MS-DOS und Windows-CE ist das Argument vom Typ *int*.

Funktion *time*:

`long time()`

Die Funktion liefert die aktuelle Zeit in Sekunden seit dem 1. Januar 1970 0:00 Uhr GMT. Sie verhält sich wie die gleichnamige C-Funktion.

Rückgabewert:

Vergangene Sekunden seit dem 1. Januar 1970 0:00 Uhr GMT

Hinweis:

Der Typ des Rückgabewertes ist plattformabhängig. Auf 16-Bit-Plattformen (DOS) wird ein 32-Bit-Wert (`int`), auf anderen Plattformen ein 64-Bit-Wert (`long`) zurückgegeben.

Funktion timer:

```
uint timer()
```

Die Funktion ist zur Verwendung für Zeitmessungen vorgesehen.

Ihr Verhalten ist dabei plattformabhängig. Auf Windows-Plattformen (einschließlich CE) liefert sie die vergangene Zeit seit den Systemstart, während auf DOS-Plattformen die Zeit seit dem Start des aktuellen Programms gemessen wird.

Rückgabewert:

Vergangene Zeit in Millisekunden.

Hinweis:

Abhängig von der verwendeten Plattform kann die Auflösung geringer als eine Millisekunde sein, unter MS-DOS liegt sie bei etwa 55 ms.

Beispiel:

```
main() {
    uint t1 = timer();
    // do anything here
    ...
    uint t2 = timer();
    print(t2-t1, " ms needed for operation.\n");
}
```

Beispiel 67 Verwenden von `timer()` zur Laufzeitmessung

11.6 Mathematische Funktionen

Wenn nicht anders angegeben, liefern die Funktionen dieses Abschnitts ein Ergebnis vom Typ `double` und akzeptieren Argumente beliebigen Typs, sofern sie in den Typ `double` konvertierbar sind. Dies schließt auch Objekttypen ein, die einen `double`-Konvertierungsoperator bereitstellen (vgl. Abschnitte 10.10.9 und 11.2.1).

Im Falle eines Überlaufs des Zahlbereiches oder der Verwendung unzulässiger Argumentwerte können die Funktionen die Fehlercodes `ERANGE` oder `EDOM` setzen. Ein Anwendungsprogramm kann dies mit Hilfe der Funktion `errno` (siehe 11.7) abfragen.

Funktion abs:

```
var abs(var x)
```

Die Funktion berechnet den Absolutbetrag des als Argument übergebenen Wertes.

Parameter:

- x : Argument

Rückgabewert:

Betrag von x . Der Typ des Rückgabewertes entspricht dem des Arguments.

Hinweis:

Für das Argument sind alle numerischen Datentypen zulässig. Eine implizite Typumwandlung erfolgt hier nicht.

Funktion atan:

```
double atan(var x)
```

Die Funktion berechnet den Arcus-Tangens des als Argument übergebenen Wertes.

Parameter:

- x : Argument

Rückgabewert:

Winkel im Bogenmaß im Bereich $(-\pi/2, +\pi/2)$

Funktion cos:

```
double cos(var rad)
```

Die Funktion berechnet den Kosinus des als Argument übergebenen Winkels.

Parameter:

- rad : Winkelangabe im Bogenmaß

Rückgabewert:

Kosinus von rad

Funktion exp:

```
double exp(var x)
```

Die Funktion berechnet Wert der Exponentialfunktion e^x .

Parameter:

- x : Exponent

Rückgabewert:

Wert von e^x

Funktion log:

```
double log(var x)
```

Die Funktion berechnet natürlichen Logarithmus $\ln(x)$.

Parameter:

- x : Argument; der Argumentwert muss positiv sein.

Rückgabewert:

Wert von $\ln(x)$

Funktion pow:

```
double pow(var x, var y)
```

Die Funktion berechnet Wert der Exponentialfunktion x^y .

Parameter:

- x : Basis
- y : Exponent

Rückgabewert:

Wert von x^y

Hinweis:

Wenn der Exponent y nicht ganzzahlig ist, muss die Basis x ein positiver Wert sein.

Funktion sin:

```
double sin(var rad)
```

Die Funktion berechnet den Sinus des als Argument übergebenen Winkels.

Parameter:

- rad : Winkelangabe im Bogenmaß

Rückgabewert:

Sinus von rad

Funktion sqrt:

```
double sqrt(var x)
```

Die Funktion berechnet die Quadratwurzel von x .

Parameter:

- x : Argument; der Argumentwert darf nicht negativ sein.

Rückgabewert:

Quadratwurzel von x

Funktion tan:

```
double tan(var rad)
```

Die Funktion berechnet den Tangens des als Argument übergebenen Winkels.

Parameter:

- rad : Winkelangabe im Bogenmaß

Rückgabewert:

Tangens von rad

11.7 Systemfunktionen

Funktion addr:

```
intptr addr(var)
```

Die Funktion liefert die Adresse der Variablen var . Sie wird u. a. für den Aufruf anderer Systemfunktionen und für Interrupt-Aufrufe benötigt, die Adressen als Eingabeparameter erwarten.

Parameter:

- var : Variablenbezeichner, zulässig sind alle Typen außer `null`.

Rückgabewert:

Adresse der angegebenen Variablen.

Der zurückgelieferte Wert hängt vom Datentyp der als Argument übergebenen Variablen ab, wobei folgendes gilt:

- Für den Typ `string` ist der Rückgabewert die Adresse des Anfangs der Zeichenkette innerhalb des String-Objekts. Dies ermöglicht die Verwendung eines Strings als Zeichenpuffer.
- Für die Typen `buffer` und `charbuffer` ist der Rückgabewert die Adresse des Anfangs des Datenbereichs.
- Für die numerischen Typen wird einfach der Wert in den Typ `intptr` umgewandelt (analog zum Aufruf der Funktion `intptr`).

- Für alle anderen (Referenz-)Typen ist das Ergebnis der Wert selbst (als Zeiger zu interpretieren). Dies gestattet die Übergabe von Datenreferenzen an ‚native‘ Funktionen, z. B. in einer dynamischen Bibliothek.

Funktion CallLibFunc:

```
intptr CallLibFunc(intptr proc, ...)
```

Die Funktion ruft die durch *proc* spezifizierte Funktion in einer dynamischen Bibliothek auf.

Parameter:

- *proc*: Zeiger auf die Funktion. Als Wert ist das Resultat eines vorangegangenen Aufrufs der Funktion `GetProcAddress` zu übergeben.
- Weitere Argumente werden in der Reihenfolge ihrer Angabe an die aufgerufene Funktion weitergeleitet. Ihre Art und Anzahl hängt von den Erfordernissen der jeweiligen Funktion ab.

Rückgabewert:

Rückgabewert des Funktionsaufrufes.

Hinweise:

- B++ weiß nichts darüber, wie eine bestimmte Funktion in einer DLL aufgerufen werden muss und kann deshalb selbst keinerlei Überprüfungen vornehmen. Die Bereitstellung der korrekten Parameter liegt allein in der Verantwortung des Benutzers, der also ggf. die Dokumentation der Bibliothek und der von ihr exportierten Funktionen zu Rate ziehen sollte.
- Die aufgerufene Funktion muss unter MS-DOS eine nach C-, sonst eine nach STDCALL-Konvention implementierte Funktion⁵⁵ sein, deren Parameter und Rückgabewert jeweils *intptr* (also je nach Plattform 32 bzw. 64 Bits breit) sind. Aus Sicht des aufrufenden B++-Programms können die übergebenen Werte Zeiger oder Zahlen repräsentieren. Um Referenztypen, insbesondere Zeichenketten, als Zeiger zu übergeben, sollte das aufrufende Programm die Funktion `addr` zur Konvertierung verwenden.
- Der Rückgabewert und die Art seiner Interpretation hängen von der Implementierung der aufgerufenen Funktion ab. Wenn die Bibliotheksfunktion keinen Wert liefert, ist der Ergebniswert undefiniert.
- Unter Windows-CE und auf x64-Plattformen ist die Anzahl der möglichen Argumente auf 16 begrenzt.

Zur Demonstration der Verwendung wird angenommen, dass eine dynamische Bibliothek für den Zugriff auf eine Datenbank existiert, die „database.dll“ heißt und eine Anmeldefunktion mit der C-Signatur

```
int login(const char* userName, const char* passwd)
```

⁵⁵Unter MS-DOS muss der Aufruf über FAR-Zeiger erfolgen. Genau genommen funktionieren unter MS-DOS und in der 32-Bit-Windows-Desktopversion Implementierungen nach beiden Konventionen, da hier beim Aufruf der Stackpointer gesichert und anschließend restauriert wird.

unter dem Namen „login“ exportiert. Diese Funktion erwartet also als Argumente einen Benutzernamen und ein Passwort (beides Zeichenketten). Sie soll bei erfolgreicher Anmeldung eine 1, sonst eine 0 zurückgeben.

Der Code-Ausschnitt in Beispiel 68 zeigt alle zum Aufruf der Funktion nötigen Schritte:

```
// Example for DLL function call
void main()
{
    // first load dynamic library
    intptr hdll = LoadLibrary("database.dll");
    // get address of exported function
    intptr proc = GetProcAddress(hdll, "login");
    // now ask user for name and password
    print("Enter your name: ");
    string name = gets();
    print("Enter your password: ");
    string pwd = gets();
    // try to login
    // don't forget to pass addresses of strings
    if (CallLibFunc(proc, addr(name), addr(pwd)))
    {
        // do anything
    }
    else
        print("Database access denied!\n");
    // release library
    FreeLibrary(hdll);
}
```

Beispiel 68 Aufruf einer DLL-Funktion aus B++

Funktion `errno`:

```
int errno()
int errno(int err)
```

Die Funktion liefert (Variante 1) oder setzt (Variante 2) den plattformspezifischen Fehlercode der zuletzt aufgerufenen Funktion des Laufzeitsystems.

Parameter:

- `err`: Fehlercode, der gesetzt werden soll

Rückgabewert:

Fehlercode des letzten Aufrufs einer Systemfunktion bzw. gesetzter Wert.

Hinweise:

- Die von dieser Funktion gelieferten Codes entsprechen den Ergebniscodes der Funktionen der C-Laufzeitumgebung der Ausführungsplattform. Dabei steht der Wert Null (0) für eine fehlerfreie Ausführung. Zur Ermittlung einer zugehörigen Fehlermeldung kann die Funktion `strerror` verwendet werden.
- Die Fehlercodes sind plattformspezifisch und bis auf `EDOM` (Domain Error) und `ERANGE` (Range Error), die in B++ als vordefinierte Konstanten (Literele) verfügbar sind, nicht standardisiert.

- Die zweite Variante der Funktion ist vor allem dazu gedacht, einen Fehlercode eines vorherigen Funktionsaufrufs zurückzusetzen. Das Ergebnis ist hier der nach dem Aufruf tatsächlich gesetzte Wert, der von *err* verschieden sein kann, wenn ein vom System nicht akzeptierter Code übergeben wurde.

Funktion FreeLibrary:

```
int FreeLibrary(intptr hdl1)
```

Die Funktion gibt eine mit `LoadLibrary` geladene dynamische Bibliothek wieder frei.

Parameter:

- *hdl1*: Handle der freizugebenden Bibliothek

Rückgabewert:

1 bei Erfolg, sonst (d. h. wenn *hdl1* ungültig war) 0

Funktion getenv:

```
dictionary getenv()
```

```
string getenv(string varname)
```

Die Funktion dient dem Lesezugriff auf Umgebungsvariablen. Dabei liefert die erste Variante ein Dictionary mit einer Kopie aller Umgebungsvariablen des aktuellen Prozesses, während die zweite Variante den Wert einer explizit benannten Umgebungsvariablen zurückgibt.

Parameter:

- *varname*: Name der Umgebungsvariablen, deren Wert abgefragt werden soll (Variante 2).

Rückgabewert:

Tabelle aller Umgebungsvariablen (Variante 1) bzw. Wert der abgefragten Variablen (Variante 2).

Hinweis:

Existiert eine mit Variante 2 abgefragte Umgebungsvariable nicht, ist das Ergebnis eine leere Zeichenkette.

Funktion GetProcAddress:

```
intptr GetProcAddress(intptr hdl1, string funcname)
```

Die Funktion liefert die Adresse der unter dem Namen *funcname* aus der durch *hdl1* spezifizierten DLL exportierten Funktion.

Parameter:

- *hdl1*: Handle der DLL, die die Funktion exportiert

- `funcname`: Bezeichnung der exportierten Funktion in der Exporttabelle der DLL

Rückgabewert:

Adresse (FAR Pointer) der exportierten DLL-Funktion. Der Rückgabewert wird als Argument für den Aufruf der Funktion `CallLibFunc` benötigt.

Hinweise:

- Existiert die angegebene Funktion nicht, so wird unter MS-DOS das aktuelle Programm unter Ausgabe einer entsprechenden Fehlermeldung abgebrochen, in den Windows-Versionen wird 0 als Ergebnis zurückgegeben.
- Exportiert die Bibliothek unter dem angegebenen Namen keine Funktion, sondern einen Datenbereich, so ist der Rückgabewert ein Zeiger auf diesen Datenbereich.
- Unter Windows kann `funcname` ein Fragezeichen (?) enthalten, das für die Unicode-Version in ein ‚W‘, sonst in ein ‚A‘ übersetzt wird.

Funktion `LoadLibrary`:

```
intptr LoadLibrary(string name)
```

Die Funktion lädt die dynamische Bibliothek `name` in den aktuellen Prozessraum und liefert ein Handle auf die Bibliothek.

Parameter:

- `name`: Dateiname der Bibliothek. Der Name muss mit Erweiterung angegeben werden und kann außerdem eine absolute oder relative Pfadangabe enthalten. Die Suche nach der Datei erfolgt zuerst ausgehend vom aktuellen Verzeichnis und dann entlang der in der Umgebungsvariablen `PATH` angegebenen Pfade.

Rückgabewert:

Handle auf die Bibliothek

Hinweise:

- Das zurückgegebene Handle wird als Argument für Aufrufe der Funktionen `GetProcAddress` und `FreeLibrary` benötigt.
- Kann die angegebene Bibliothek nicht geladen werden, wird das aktuelle Programm unter MS-DOS mit Ausgabe einer entsprechenden Fehlermeldung abgebrochen, unter Windows wird der Wert 0 zurückgegeben..
- Eine mit `LoadLibrary` geladene Bibliothek sollte, sobald sie nicht mehr benötigt wird, mit Hilfe der Funktion `FreeLibrary` wieder freigegeben werden.
- Wenn `dllname` ein Sternchen (*) enthält, wird dieses Zeichen durch eine Build-spezifische Kennung ersetzt, die folgende Bestandteile enthalten kann:
 - `u` : bezeichnet eine Unicode-Version (UTF-16)
 - `d` : bezeichnet eine Debug-Version
 - `_64` : bezeichnet eine 64-Bit-Version

Beispiel 69 veranschaulicht die Verwendung von `LoadLibrary` am Beispiel der Anzeige einer Windows-Messagebox.

```
// This sample program works for Win32 Desktop platforms.
// For Windows CE load coredll.dll instead of user32.dll .
#define MB_ICONINFORMATION 0x40
// function shows a simple message box
ShowMessage(string title, string msg)
{
    intptr hinst = LoadLibrary("user32.dll");
    // ? will be replaced by 'W' or 'A', depending if unicode build or not
    intptr fptr = GetProcAddress(hinst, "MessageBox?");
    // use addr function to get C-character pointers from strings
    CallLibFunc(fptr, 0, addr(msg), addr(title), MB_ICONINFORMATION);
    FreeLibrary(hinst);
}
main()
{
    ShowMessage("Hello World", "This is a message\nfrom B++.");
}
```

Beispiel 69 Ausgabe einer Windows-Messagebox

Funktion `memmove`:

```
intptr memmove(intptr dst, intptr src, uint n)
```

Die Funktion kopiert einen Datenbereich innerhalb des Arbeitsspeichers. Sie verhält sich wie die gleichnamige C-Funktion, d. h., es werden *n* Bytes ab der durch *src* gegebenen Adresse in den mit der Adresse *dst* beginnenden Speicherbereich kopiert. Eine Überlappung der Speicherbereiche ist dabei zulässig.

Parameter:

- *dst*: Zieladresse der Operation
- *src*: Quelladresse der Operationen
- *n*: Anzahl der zu kopierenden Bytes

Rückgabewert:

Zieladresse (Wert von *dst*)

Hinweise:

- Die Funktion führt beim Aufruf ggf. implizite Typkonvertierungen der Argumente durch.
- Auch auf DOS-Plattformen können mit `memmove` Speicherblöcke von mehr als 64 kB kopiert werden.
- Im Falle ungültiger Argumenttypen oder eines Fehlers während der Kopieroperation wird eine Ausnahme vom Typ `string` ausgelöst.
- **Warnung:** Die Funktion gestattet beliebige Manipulationen des Arbeitsspeichers, was bei fehlerhafter Anwendung zum Systemabsturz oder zu Datenverlusten führen kann!.

Funktion `nerr`:

```
int nerr()
```

Die Funktion liefert die Anzahl der vom System definierten Standard-Fehlercodes.

Rückgabewert:

Anzahl der definierten Fehlercodes. Der Wert hängt von der verwendeten Ausführungsumgebung ab.

Hinweis:

Die Funktion kann verwendet werden, um die Gültigkeit eines Fehlercodes vor dem Aufruf von `strerror` zu überprüfen oder um die verfügbaren Fehlermeldungen auszulesen.

Beispiel 70 illustriert die Verwendung von `nerr` und `strerror` zur Ausgabe der System-Fehlermeldungen.

```
void main()
{
    int n = nerr();
    for (int i=0; i<n; ++i)
        print(i, "\t", strerror(i), "\n");
}
```

Beispiel 70 Verwenden der Funktionen `nerr()` und `strerror(int)`

Funktion `setenv`:

```
int setenv(string varname, string value)
```

Die Funktion definiert eine neue Umgebungsvariable oder ändert den Wert einer vorhandenen.

Parameter:

- `varname`: Name der Umgebungsvariablen
- `value`: neuer Variablenwert

Rückgabewert:

1 bei Erfolg, sonst 0

Hinweise:

- Die Funktion kann scheitern, wenn `varname` leer oder kein syntaktisch gültiger Bezeichner einer Umgebungsvariablen ist bzw. wenn der notwendige Platz im Umgebungs-Variablenbereich vom System nicht bereitgestellt werden kann.
- Wird für `value` eine leere Zeichenkette angegeben, wird die betreffende Umgebungsvariable gelöscht.

Funktion `setlocale`:

```
string setlocale(int category, string locale)
```

Die Funktion setzt das Lokalisierungsschema der Laufzeitumgebung oder fragt dessen aktuelle Einstellungen ab. Sie verhält sich wie die gleichnamige C-Funktion⁵⁶.

Parameter:

- `category`: Kategorie, für die Einstellungen festgelegt oder bestimmt werden sollen, zulässige Werte sind:
 - `LC_ALL (0)`: alle Kategorien
 - `LC_COLLATE (1)`: Zeichenkettenvergleiche
 - `LC_CTYPE (2)`: Funktionen für einzelne Zeichen
 - `LC_MONETARY (3)`: Währungsformat
 - `LC_NUMERIC (4)`: numerische Formate
 - `LC_TIME (5)`: Zeitformat
- `locale`: Zeichenkette mit der Lokalisierungsbeschreibung, z. B. „en-US“, „German_Germany.437“, „.OCP“ oder „C“. Eine leere Zeichenkette bewirkt die Einstellung auf ‚System default settings‘; Die Angabe von `null` bewirkt das Lesen der aktuellen Einstellungen, ohne daran Änderungen vorzunehmen.

Hinweise:

- Das konkrete Verhalten der Funktion (insbesondere die Interpretation der Angabe in `locale`) hängt vom verwendeten Betriebssystem ab.
- Auf DOS-Plattformen ist die Funktion zwar vorhanden, bewirkt aber nichts, da dort nur die „C“-Lokalisierung unterstützt wird.
- Die „C“-Lokalisierung ist die Standardeinstellung, die beim Start des Bytecode-Intepreters gültig ist.

Funktion `strerror`:

```
string strerror()
```

```
string strerror(int errcode)
```

Die Funktion liefert eine Textbeschreibung des zuletzt aufgetretenen (erste Variante) oder des angegebenen Fehlercodes.

Parameter:

- `errcode`: Fehlercode, für den eine Beschreibung bestimmt werden soll.

Rückgabewert:

Fehlerbeschreibung

⁵⁶ Die Implementierung in B++ ruft die C-Funktion direkt auf.

Hinweise:

- Der Aufruf ohne Parameter entspricht einem Aufruf von `strerror(errno())`.
- Wird für `errcode` ein ungültiger Wert angegeben, ist das Ergebnis „Unknown error“.

Funktion system:

```
int system(string cmd)
```

Die Funktion führt die in `cmd` angegebene Kommandozeile als synchronen Kindprozess aus. Dazu wird der systemeigene Kommandozeileninterpreter mit dem angegebenen Kommando aufgerufen.

Parameter:

- `cmd`: Kommandozeile, die an das System übergeben werden soll. Die Zeile kann den Namen (optional mit Pfad) eines Programms oder ein internes Kommando des Kommandoprozessors, evtl. gefolgt von Parametern, enthalten.

Rückgabewert:

Ergebniswert der Kommandoausführung

Hinweise:

- Die Funktion ist unter Windows-CE nicht verfügbar, da es dort keinen Standard-Befehlsprozessor gibt.
- Die Lokalisierung des Kommandozeileninterpreters erfolgt über die Umgebungsvariable `COMSPEC`.
- Wenn der Aufruf scheitert, wird eine Ausnahme vom Typ `string` ausgelöst. Ein Anwendungsprogramm kann die Ausnahme abfangen und bei Bedarf mit den Funktionen `errno` bzw. `strerror` die Fehlerursache bestimmen.

11.7.1 MS-DOS-Funktionen

Die Funktionen dieses Abschnitts stellen Hilfsmittel zur Systemprogrammierung unter MS-DOS bereit und sind dementsprechend nur in der DOS-Version verfügbar.

Funktion getreg:

```
int getreg(string regname)
```

Die Funktion liest den aktuellen Wert der angegebenen Register-Variablen.

Parameter:

- `regname`: Name eines Prozessorregisters, möglich sind:
`ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh, cflag, flags, ds, es, ss, cs`.
Die Groß- und Kleinschreibung wird bei der Angabe der Registernamen ignoriert.

Rückgabewert:

aktueller Wert der angegebenen Register-Variablen

Hinweis:

Die Funktion liest nicht die tatsächlichen aktuellen Werte der Prozessorregister aus, sondern greift auf eine interne Datenstruktur zu, die vorher innerhalb des Programms explizit (mit `setreg` bzw. `segreg`) oder implizit (in Folge eines `int86`- oder `int86x`-Aufrufs) initialisiert wurde.

Funktion inport:

```
int inport(int portid)
```

Die Funktion liest ein Wort (16 Bit) ab Portadresse `portid`. Dabei wird das Low-Byte direkt von `portid` und das High-Byte von `portid+1` gelesen.

Parameter:

- `portid`: Port-Nummer

Rückgabewert

gelesenes Wort

Funktion inportb:

```
int inportb(int portid)
```

Die Funktion liest ein einzelnes Byte von Portadresse `portid`.

Parameter:

- `portid`: Port-Nummer

Rückgabewert:

gelesenes Byte

Funktion int86:

```
int int86(int intr)
```

Die Funktion löst einen Aufruf des durch `intr` spezifizierten x86-Interrupts aus. Vor dem Aufruf werden die Prozessorregister `AX`, `BX`, `CX`, `DX`, `DI`, `SI` und `FLAGS` mit den Inhalten der entsprechenden internen Registervariablen geladen. Nach dem Aufruf werden die Registerwerte in die internen Variablen übernommen und die ursprünglichen Registerinhalte wieder hergestellt.

Parameter:

- `intnr`: Nummer des aufzurufenden Interrupts

Rückgabewert:

Wert des Registers `AX` nach dem Interrupt-Aufruf

Funktion `int86x`:

```
int int86x(int intnr)
```

Die Funktion löst einen Aufruf des durch `intnr` spezifizierten x86-Interrupts aus. Vor dem Aufruf werden die Prozessorregister `AX`, `BX`, `CX`, `DX`, `DI`, `SI`, `FLAGS`, `DS` und `ES` mit den Inhalten der entsprechenden internen Registervariablen geladen. Nach dem Aufruf werden die Registerwerte in die internen Variablen übernommen und die ursprünglichen Registerinhalte wieder hergestellt.

Parameter:

- `intnr`: Nummer des aufzurufenden Interrupts

Rückgabewert:

Wert des Registers `AX` nach dem Interrupt-Aufruf

Hinweis:

Vor Benutzung der Funktion sollten die internen Variablen der Segmentregister mit Hilfe der Funktion `segread` auf die tatsächlichen Werte gesetzt werden.

Beispiel 71 Illustriert die Verwendung der Funktion zur Ausgabe des aktuellen Verzeichnispfades.

```
// interrupt call example - prints current directory path
main(;buf,bufptr)
{
    buf = newstring(256); // buffer for result
    segread(); // init variables for seg-registers
    setreg("ah",0x47); // dosfunc nr.
    setreg("dl",0); // get path for current drive
    bufptr = addr(buf); // get address of buffer contents
    setreg("ds",bufptr >> 16); // high word of buffer address -> segment
    setreg("si",bufptr & 0xFF); // low word of buffer address -> offset
    int86x(0x21); // call DOS-Interrupt
    print(buf,"\n");
}
```

Beispiel 71 Verwenden von Interrupt-Aufrufen unter MS-DOS

Funktion `memsize`:

```
int memsize()
```

Die Funktion ermittelt die Größe des noch verfügbaren freien Speichers.

Rückgabewert:

Größe des verfügbaren freien Speichers in Bytes

Hinweis:

Genau genommen ist das Ergebnis die Größe des größten zusammenhängenden freien Speicherbereichs auf dem Heap.

Funktion output:

```
void output(int portid, int value)
```

Die Funktion schreibt ein Wort (16 Bit) nach Portadresse *portid*. Dabei wird das Low-Byte direkt nach *portid* und das High-Byte nach *portid+1* ausgegeben.

Parameter:

- *portid*: Port-Nummer
- *value*: auszugebender Wert

Funktion outportb:

```
void outportb(int portid, int value)
```

Die Funktion schreibt ein einzelnes Byte *nach* Portadresse *portid*.

Parameter:

- *portid*: Port-Nummer
- *value*: auszugebender Wert

Funktion segread:

```
void segread()
```

Die Funktion lädt die aktuellen Werte der Segmentregister des Prozessors (DS, ES, SS, CS) in die entsprechenden internen Variablen.

Funktion setreg:

```
void setreg(string regname, int value)
```

Die Funktion setzt den Wert der angegebenen internen Registervariablen.

Parameter:

- `regname`: Name eines Prozessorregisters, möglich sind:
ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh, cflag, flags, ds, es, ss, cs.
Die Groß- und Kleinschreibung wird bei der Angabe der Registernamen ignoriert.
- `value`: neuer Wert

Hinweis:

Die Werte der internen Register-Variablen werden beim Aufruf der Funktionen `int86` bzw. `int86x` in die entsprechenden Prozessorregister übernommen.

11.7.2 Windows-Funktionen

Die Funktionen dieses Abschnitts sind nur auf Windows-Plattformen verfügbar und erleichtern den Zugriff auf das Windows-API von B++ aus.

Funktion `GetLastError()`

```
uint GetLastError()
```

Die Funktion liefert den Code des im aufrufenden Thread zuletzt aufgetretenen Fehlers. Ihre Implementierung ist ein einfacher Wrapper um die gleichnamige Windows-API-Funktion.

Rückgabewert:

zuletzt aufgetretener Fehlercode des aufrufenden Threads

Hinweise:

- Die Funktion dient, analog zu `errno()`, der näheren Bestimmung aufgetretener Fehler, bezieht sich aber speziell auf Windows-API-Aufrufe. Die von der Funktion gelieferten Fehlercodes unterscheiden sich von denen der Funktion `errno()`.
- Die Funktion sollte unmittelbar nach der Funktion aufgerufen werden, die einen Fehler verursacht hat, weil weitere erfolgreiche API-Aufrufe den Fehlercode möglicherweise zurücksetzen.

Funktion `GetSysErrorMsg`

```
string GetSysErrorMsg()
```

```
string GetSysErrorMsg(uint errcode)
```

Die Funktion liefert eine Textbeschreibung des zuletzt aufgetretenen (erste Variante) oder des angegebenen Fehlercodes eines Windows-API-Aufrufs.

Parameter:

- `errcode`: Fehlercode, für den eine Beschreibung bestimmt werden soll.

Rückgabewert:

Fehlerbeschreibung

Hinweise:

- Der Aufruf ohne Parameter entspricht einem Aufruf von `GetSysErrorMessage(GetLastError())`.
- Wird für `errcode` ein ungültiger Wert angegeben, ist das Ergebnis eine leere Zeichenkette.

Funktionen GetXXXFunc:

Zur Vereinfachung des Zugriffs auf die wichtigsten Windows-API-Funktionen verwaltet B++ die Standardbibliotheken des Windows-API automatisch, so dass hier kein expliziter Aufruf von `LoadLibrary` bzw. `FreeLibrary` erforderlich ist. Die `GetXXXFunc`-Funktionen entsprechen dem Aufruf von `GetProcAddress` für die jeweilige Standardbibliothek.

```
IntPtr GetCoreFunc (string name)
IntPtr GetGdi32Func (string name)
IntPtr GetKernel32Func (string name)
IntPtr GetOle32Func (string name)
IntPtr GetShell32Func (string name)
IntPtr GetUser32Func (string name)
```

Die Funktionen liefern die Adresse eines Symbols (Funktion oder Variable) aus einer der Bibliotheken `corelib.dll`, `gdi32.dll`, `kernel32.dll`, `ole32.dll`, `shell32.dll` bzw. `user32.dll`.

Parameter:

- `name`: Name eines exportierten Symbols aus der betreffenden Bibliothek

Rückgabewert:

Adresse des durch `name` spezifizierten Symbols (Funktions- oder Datenzeiger) bei Erfolg, sonst (wenn das durch `name` bezeichnete Symbol nicht existiert) 0.

Hinweise:

- Der Ergebniswert kann direkt als Argument für die Funktion `CallLibFunc` verwendet werden
- Die Bibliothek `corelib.dll` existiert nur unter Windows-CE. In den Windows-Desktop-Versionen von B++ ist die Funktion `GetCoreFunc` deshalb lediglich ein Alias auf `GetUser32Func`.
- Unter Windows-CE sind die Funktionen `GetGdi32Func`, `GetKernel32Func`, `GetShell32Func` und `GetUser32Func` Aliase auf `GetCoreFunc`.
- Liefert ein Aufruf keine Funktion, sondern die Adresse eines Datenbereichs, so ist das Funktionsergebnis die Anfangsadresse des betreffenden Bereichs.
- Enthält `name` ein Fragezeichen (?), so wird dies durch ‚W‘ in Unicode-Versionen, sonst durch ‚A‘ ersetzt.

11.8 MS-DOS-Grafikfunktionen

Die Grafikfunktionen ermöglichen elementare Ausgaben im Grafikmodus. Intern werden BIOS-Aufrufe verwendet.

Alle Funktionen dieses Abschnitts stehen nur in der MS-DOS-Version zur Verfügung.

Funktion `setscrmode`:

```
void setscrmode(int mode)
```

Die Funktion löscht den Bildschirm und schaltet die Anzeige in den angegebenen Modus um.

Parameter:

- `mode`: neuer Bildschirmmodus

Hinweis:

Die konkrete Bedeutung von `mode` hängt vom verwendeten Grafik-Adapter bzw. dessen BIOS ab. Allgemein steht der Wert 7 für den MDA-Modus (monochromer Textmodus).

Funktion `setpixel`:

```
void setpixel(int x, int y, int color)
```

Die Funktion setzt einen einzelnen Bildschirmpunkt auf den mit `color` spezifizierten Farbwert.

Parameter:

- `x`: x-Koordinate
- `y`: y-Koordinate
- `color`: Farbwert

Hinweise:

- Die Funktion ist nur bei eingeschaltetem Grafikmodus sinnvoll verwendbar.
- Die Werte für die Koordinaten sollten nicht negativ sein und werden nach oben durch die jeweilige Bildschirmauflösung begrenzt, die wiederum vom eingestellten Grafikmodus abhängt. Der Punkt (0, 0) bezeichnet die linke obere Bildschirmecke.
- Beim in `color` angegebenen Wert sind nur die niederwertigsten 8 Bit signifikant. Die konkrete Bedeutung hängt vom Bildschirmmodus ab.

Funktion `line`:

```
void line(int x1, int y1, int x2, int y2, int color)
```

Die Funktion zeichnet eine Linie in der mit `color` spezifizierten Farbe vom Punkt (`x1`, `y1`) nach Punkt (`x2`, `y2`).

Parameter:

- `x1`: x-Koordinate des Anfangspunktes
- `y1`: y-Koordinate des Anfangspunktes
- `x2`: x-Koordinate des Endpunktes
- `y2`: y-Koordinate des Endpunktes
- `color`: Farbwert

Hinweise:

- Die Funktion ist nur bei eingeschaltetem Grafikmodus sinnvoll verwendbar.
- Die Werte für die Koordinaten sollten nicht negativ sein und werden nach oben durch die jeweilige Bildschirmauflösung begrenzt. Der Punkt (0, 0) bezeichnet die linke obere Bildschirmecke.
- Beim in `color` angegebenen Wert sind nur die niederwertigsten 8 Bit signifikant. Die konkrete Bedeutung hängt vom Bildschirmmodus ab.

Beispiel 72 illustriert den Einsatz der Grafikfunktionen, wobei der CGA-Farbgrafikmodus 4 (320x200 Pixel, 4 Farben) verwendet wird. Dabei wird die für den Aufbau der Grafik benötigte Zeit gemessen, um nach dem anschließenden Umschalten in den Textmodus (MDA-Modus 7) die Zeichengeschwindigkeit (Linien pro Sekunde) auszugeben.

```
/* Example for using graphic functions */
main()
{
    // resolution
    int px=320;
    int py=200;
    // switch to graphic mode
    setscrmode(4);
    // get current time
    uint ms = timer();
    uint n=0;
    // draw some lines
    for (int x=0; x<px; x+=2)
    {
        ++n;
        int x2= px -x;
        line(x,0,x2,py-1,1);
    }
    for (int y=0; y<py; y+=2)
    {
        ++n;
        int y2= py-1 -y;
        line(0,y,px-1,y2,2);
    }
    // get elapsed time
    ms = timer() - ms;
    uint r = n*1000 / ms;
    gets(); // wait for ENTER key
    // switch to text mode
    setscrmode(7);
    print(r, " lines per second\n");
}
```

Beispiel 72 Zeichnen von Linien im Grafikmodus unter MS-DOS

11.9 Sonstige Funktionen

In diesem Abschnitt werden einige Funktionen beschrieben, die sich keiner der obigen Kategorien zuordnen lassen.

Funktion `arg`:

```
var arg(int idx)
```

Die Funktion liefert das Argument mit dem Index `idx` der aufrufenden Funktion. Sie kann in Verbindung mit der Funktion `argcnt` zur Konstruktion von Funktionen mit variablen Argumentlisten verwendet werden.

Parameter:

- `idx`: Index eines Arguments der aufrufenden Funktion

Rückgabewert:

Argument mit dem Index `idx`

Hinweis:

Die Funktion prüft die Gültigkeit des in `idx` übergebenen Wertes. Bei einem ungültigen Wert wird eine Ausnahme vom Typ `string` ausgelöst.

Beispiel 73 Illustriert die Verwendung von `arg` und `argcnt` zur Implementierung einer Funktion mit variabler Argumentliste.

```
/* using functions argcnt() and arg(idx) */
var f(var x) // only first argument has a name
{
    int n = argcnt();
    print("\n",n," arguments:\n");
    for (int i=0;i<n;++i)
        print("\t",arg(i));
    print("\n---\n");
    return x;
}

void main()
{
    print(f(1,3,"test"));
    /* should print
    3 arguments :
        1      3      test
    ---
    1
    */
}
```

Beispiel 73 Implementieren einer Funktion mit variabler Argumentzahl

Funktion `argc`:

```
int argc()
```

Die Funktion liefert die Anzahl der Argumente, die der aufrufenden Funktion übergeben wurden. Sie kann in Verbindung mit der Funktion `arg` zur Implementierung von Funktionen mit variabler Argumentzahl in B++ verwendet werden.

Rückgabewert:

Anzahl der durch die aufrufende Funktion übergebenen Argumente

Hinweise:

- Beim Aufruf nicht-statischer Elementfunktionen (einschließlich Operatorfunktionen) von Objekten wird implizit der `this`-Verweis des jeweiligen Objekts als erstes Argument übergeben. Somit liefert `argc` hier immer mindestens den Wert 1.
- In den Kommandozeilenversionen von B++ wird die Einsprungfunktion `main` mit allen auf der Kommandozeile übergebenen Parametern, einschließlich des Programmnamens als erstem Wert, aufgerufen.

Funktion `argv`:

```
vector argv()
```

Die Funktion liefert einen Vektor mit den aktuellen Argumenten, die der aufrufenden Funktion übergeben wurden. Ihr wichtigster Einsatzzweck ist die Weitergabe einer variablen Menge von Argumenten an eine andere Funktion, insbesondere im Zusammenhang mit der Funktion `call`.

Rückgabewert:

Vektor aller aktuellen Parameter der aufrufenden Funktion

Hinweis:

Wenn die aufrufende Funktion eine nicht-statische Elementfunktion (Methode eines Objekts) ist, ist das erste Element des Ergebnisvektors der Objektverweis.

Funktion `call`:

```
var call(function f, vector args = null)
```

Die Funktion führt einen Aufruf der Funktion `f` mit den Elementen von `args` als aktuellen Aufrufparametern durch. Dabei kann `f` eine einfache Funktion, aber auch eine nicht-statische Elementfunktion eines Objekts sein.

Typische Einsatzfälle für `call` sind die Konstruktion von Funktionen mit variabler Argumentzahl, die ihrerseits ebenfalls Funktionen mit variabler Argumentzahl aufrufen, sowie indirekte Aufrufe über Funktions- und/oder Methodenzeiger.

Parameter:

- `f` : aufzurufende Funktion (Funktions- oder Methodenzeiger
- `args` : Vektor der Argumente; falls `f` ein Methodenzeiger ist, muss `args` in jedem Fall angegeben werden und als erstes Element den zugehörigen Objektverweis enthalten.

Rückgabewert:

Ergebnis der aufgerufenen Funktion; liefert `f` kein Ergebnis, ist der Rückgabewert undefiniert.

Beispiel 74 illustriert die Verwendung von `call` zur Nachbildung der `printf`-Funktion aus C.

```
int printf(string format)
{
    vector v = argv();
    print(call(newstring,v));
    return int(size(v))-1;
}
```

Beispiel 74 Nachbildung der `printf`-Funktion

Funktion `clone`:

```
var clone(var val)
```

Die Funktion erzeugt eine Kopie eines Wertes⁵⁷ beliebigen Typs.

Sie dient der Erzeugung einer vollständigen (tiefen) Kopie von Werten der Containertypen (`vector`, `buffer`, `charbuffer` bzw. `dictionary`), des Typs `string` oder von Objektinstanzen. Für alle anderen Argumenttypen gibt sie das Argument selbst zurück.

Parameter:

- `arg` : zu kopierender Wert

Rückgabewert:

Kopie von `arg`.

Funktion `compile`:

```
bool compile(FILE fp, string srcname = "", bool debug = false)
bool compile(string src, string srcname = "", bool debug = false)
```

Die Funktion übersetzt zur Laufzeit Quellcode aus einer geöffneten Textdatei (Variante 1) oder einer übergebenen Zeichenkette (Variante 2) in Bytecode und bindet das Ergebnis in das aktuelle Programm ein.

Parameter:

- `fp`: Handle einer geöffneten Textdatei

⁵⁷ Im Fall einer Weak Reference wird nur die Referenz, nicht das referenzierte Datum kopiert.

- `src`: Zeichenkette mit dem zu übersetzenden Quelltext
- `srcname`: Bezeichnung der Quelle; wird für Fehlerausgaben während der Übersetzung sowie für Quelltextbezüge im Bytecode (falls `debug` gesetzt ist) verwendet.
- `debug`: Flag für zusätzliche Debug-Informationen. Ist es gesetzt, werden in den erzeugten Bytecode zusätzlich Bezüge zu den Zeilen des Quelltextes sowie die Namen lokaler Variablen aufgenommen.

Rückgabewert:

`true` (1) bei Erfolg, sonst `false` (0)

Dabei bedeutet „Erfolg“ das erfolgreiche Übersetzen der Quelle in Bytecode.

Hinweise:

- Die Funktion unterscheidet sich grundsätzlich von `include`-Anweisungen anderer Programmiersprachen und auch von der `#include`-Direktive in C++, indem der Code nicht zur Übersetzungs-, sondern zur Laufzeit eingebunden wird.
- Der Aufruf von `compile` kann an jeder beliebigen Stelle erfolgen, an der ein Funktionsaufruf möglich ist. Die Einbindung des erzeugten Programmfragments erfolgt allerdings *immer in den globalen Kontext* des Programms. Bereits vorhandene gleichnamige Symbole (Bezeichner) werden dabei durch die neuen ersetzt. Diese Eigenschaft lässt sich verwenden, um z. B. abhängig von Bedingungen bestimmte Funktionen zur Laufzeit auszutauschen, ohne den sie verwendenden Clientcode ändern zu müssen.

Beispiel 75 illustriert den Einsatz von `compile` in der zweiten Variante zum Ersetzen einer Funktion zur Laufzeit.

```
// define a test function
testfunc(n) {
    print("testfunc(\"",n,"\" ) called.\n");
}

main(;fstr) {
    // call testfunc
    testfunc("Hello World");
    // create source of alternate function
    fstr =
        R"(testfunc(n) {
            print("alternate version of testfunc called:",n,"\\n");
        })";
    if (compile(fstr))
        testfunc("Hello World");
}
```

Beispiel 75 Dynamisches Ersetzen einer Funktion mit `compile`

Funktion `compilefn`:

```
function compilefn(string src, string args = "", string rtype = "",
    var ownerClass = null, string srcname = "",
    bool debug = false)
```

Die Funktion übersetzt den in `src` übergebenen Code zur Laufzeit in eine Funktion bzw. eine Elementfunktion einer Klasse, ohne das Ergebnis unmittelbar in das laufende Programm zu integrieren.

Parameter:

- `src`: Quellcode des Funktionskörpers
- `args`: Inhalt der formalen Argumentliste der Funktion. Die Notation erfolgt in der gleichen Weise, wie bei einer ‚normalen‘ Funktionsdeklaration.
- `rtype`: Bezeichner für den Typ des Rückgabewerts der Funktion. Eine leere Zeichenkette bedeutet dabei, dass kein konkreter Typ spezifiziert wird.
- `ownerClass`: Typ oder Klassenname der Klasse, für die eine Elementfunktion erzeugt werden soll. Fehlt die Angabe, wird eine einfache Funktion erzeugt.
- `srcname`: Bezeichnung der Quelle; wird für Fehlerausgaben während der Übersetzung sowie für Quelltextbezüge im Bytecode (falls *debug* gesetzt ist) verwendet.
- `debug`: Flag für zusätzliche Debug-Informationen. Ist es gesetzt, werden in den erzeugten Bytecode zusätzlich Bezüge zu den Zeilen des Quelltextes sowie die Namen lokaler Variablen aufgenommen.

Rückgabewert:

Bei Erfolg liefert die Funktion als Ergebnis die erzeugte Bytecode-Funktion (Typ `function`), andernfalls `null`.

Hinweise:

- Die Funktion ermöglicht es, zur Laufzeit beliebige Funktionen zu generieren und einer Variablen zuzuweisen bzw. sie direkt aufzurufen, ohne dass sich daraus Seiteneffekte für den Code des ausgeführten Programms ergeben.
- Innerhalb des Funktionskörpers (dem Inhalt von `src`) sind alle globalen Symbole sowie die Elemente von `ownerClass` (falls spezifiziert), jedoch nicht die lokalen Symbole des Aufrufers sichtbar.
- Wenn mit `compilefn` eine Elementfunktion einer Klasse erzeugt wird, muss diese Funktion im Kontext der betreffenden Klasse aufgerufen werden.

In Beispiel 76 wird `compilefn` zur interaktiven Berechnung von Ausdrücken verwendet.

```
// Evaluate expressions at runtime
var evalExpr(string expr)
{
    var f = compilefn("return (" + expr + ");");
    if (f) return f();
    return null;
}

void main()
{
    while (true)
    {
        print("Expression: ");
        string s = gets();
        if (size(s) == 0) break;
        print(" Result: ", evalExpr(s), "\n")
    }
}
```

Beispiel 76 Einsatz der Funktion `compilefn`

Funktion `getargs`:

```
vector getargs()
```

Die Funktion liefert einen Vektor mit allen beim Aufruf von B++ übergebenen Kommandozeilenargumenten zurück. Dabei ist das Element mit dem Index 0 der Name des ausgeführten Programms (z. B. `bp2.exe`).

Rückgabewert:

Vektor mit allen Kommandozeilenargumenten. Alle Elemente des Ergebnisvektors sind vom Typ `string`.

Hinweis:

Wird B++ nicht als Kommandozeilenversion, sondern als Bibliothek (d. h. als Skriptsprache innerhalb eines anderen Programms) verwendet, ist das aufrufende Programm dafür verantwortlich, bei Bedarf Argumente bereitzustellen – siehe hierzu die Methoden `BppVirtualMachine::execute` bzw. `BppInterpreter::execute` in der Referenzdokumentation.

Funktion `getusrargs`:

```
vector getusrargs()
```

Die Funktion liefert einen Vektor mit allen beim Aufruf übergebenen Benutzer-Argumenten zurück. Dabei ist das Element mit dem Index 0 das erste Kommandozeilenargument, das hinter dem Kennzeichen `#` angegeben wurde.

Rückgabewert:

Vektor mit allen Benutzer-Argumenten. Alle Elemente des Ergebnisvektors sind vom Typ `string`.

Funktion `litexpr`:

```
var litexpr (string src, string srcname="")
```

Die Funktion wertet den in `src` als Zeichenkette übergebenen literalen Ausdruck oder Literalwert aus und gibt das Ergebnis der Auswertung zurück.

Parameter:

- `src`: Zeichenkette mit auszuwertendem Literal
- `srcname`: Bezeichnung der Quelle; wird für Fehlerausgaben während der Übersetzung verwendet.

Rückgabewert:

Ergebnis des Ausdrucks in `src`

Hinweise:

- Wenn `src` nicht ausgewertet werden kann, d. h. einen syntaktischen Fehler aufweist, wird eine Ausnahme vom Typ `string` mit einer Fehlerbeschreibung ausgelöst.
- Die Funktion kann zum Lesen von JSON-Datenstrukturen verwendet werden, da jede gültige JSON-Struktur ein gültiges `dictionary`-Literal darstellt.

Funktion `loadmodule`:

```
bool loadmodule(string filename)
```

Die Funktion bindet zur Laufzeit ein bereits vorkompiliertes Bytecode-Modul in das aktuelle Programm ein.

Parameter:

- `filename`: Name der Datei, die das zu ladende Modul enthält.

Rückgabewert:

`true` (1) bei Erfolg, sonst `false` (0)

Dabei bedeutet „Erfolg“ das erfolgreiche Lesen des Bytecodes.

Hinweise:

- Die Funktion unterscheidet sich von der Verarbeitungsanweisung `#use` insofern, als der in `filename` enthaltene Bytecode nicht zur Übersetzungs-, sondern zur Laufzeit eingebunden wird.
- Der Aufruf von `loadmodule` kann an jeder beliebigen Stelle erfolgen, an der ein Funktionsaufruf möglich ist. Die Einbindung des geladenen Bytecodes erfolgt immer in den globalen Kontext des Programms. Bereits vorhandene gleichnamige Symbole (Bezeichner) werden dabei durch die neuen ersetzt. Diese Eigenschaft lässt sich verwenden, um z. B. abhängig von Bedingungen bestimmte Funktionen zur Laufzeit auszutauschen, ohne den sie verwendenden Clientcode ändern zu müssen.

Funktion `rand`:

```
int rand()
```

Die Funktion berechnet eine Pseudo-Zufallszahl. Sie verhält sich wie die gleichnamige C-Funktion.

Rückgabewert:

Zufallszahl zwischen 0 und 0x7FFF

Hinweis:

Bei mehreren Aufrufen der Funktion ergibt sich eine annähernd gleichverteilte Zufallsfolge, die vom Initialwert des Zufallsgenerators abhängt. Ohne explizite Initialisierung des Zufallsgenerators (vgl. Funktion `srand`) wird nach dem Programmstart immer die gleiche Folge generiert.

Funktion `srand`:


```
void srand(int seed)
```

Die Funktion initialisiert den internen Zufallsgenerator mit dem Wert *seed*. Sie verhält sich wie die gleichnamige C-Funktion.

Parameter:

- *seed*: Initialwert für den Zufallsgenerator

Hinweis:

Beim Programmstart wird der Zufallsgenerator implizit mit dem Wert 1 initialisiert.

Um tatsächlich „zufällige Zufallsfolgen“ zu erhalten, ist eine Initialisierung mit einem zeitabhängigen Wert üblich.

In Beispiel 77 werden die Funktionen *srand* und *rand* für die Ausgabe einer Zufallsfolge verwendet.

```
/* random number generator example */
main()
{
    for(uint l=0;l<5;++l)
    {
        var seed = timer();
        srand(seed);
        print("initial value: ",seed,"\n");
        for (uint i=0;i<10;++i)
        {
            for (uint j=0;j<10;++j)
                print(rand()," ");
            print("\n");
        }
    }
}
```

Beispiel 77 Zufallszahlen mit den Funktionen *rand* und *srand*

12 Reguläre Ausdrücke

Zur erleichterten Verarbeitung von Texten unterstützt B++ ab der Version 1.1 reguläre Ausdrücke. Dies erfolgt mit Hilfe der vordefinierten Klasse `Regex`, die ihrerseits auf die Bibliothek Tiny-Rex (Tiny-Rex 1.3) [6] aufsetzt. Der dabei unterstützte Sprachumfang entspricht im Wesentlichen einer Teilmenge der ECMA-Syntax für reguläre Ausdrücke (siehe hierzu auch [7]) mit einigen implementierungsbedingten Besonderheiten.

12.1 Pattern-Syntax

Wie C++11 (und anders als z. B. JavaScript) verwendet B++ keinen besonderen Literaltyp für reguläre Ausdrücke. Die eingesetzten Suchmuster (Pattern) werden stattdessen als Zeichenketten angegeben, wobei sich für die Notation als Literal häufig die Verwendung von ‚Raw-Strings‘ (siehe Abschnitt 4.3.2, S. 16) anbietet.

Nachfolgend werden die in B++ verfügbaren syntaktischen Elemente regulärer Ausdrücke übersichtsartig dargestellt.

Sonderzeichen

Damit sind spezielle Zeichen bzw. Escape-Sequenzen gemeint, die in einem Pattern eine Sonderbedeutung haben. Dabei handelt es sich entweder – analog zur ‚normalen‘ B++-Syntax – um Repräsentationen nicht druckbarer Zeichen oder um einzelne Zeichen mit besonderer Bedeutung in einem Pattern.

Im Einzelnen sind folgende Sonderzeichen definiert:

Zeichen	Bedeutung
<code>\</code>	Hebt die Sonderbedeutung des nächsten Zeichens auf; insbesondere steht <code>\\</code> für die Repräsentation eines Backslashes.
<code>.</code>	Der Punkt steht für ein beliebiges Zeichen außer dem Zeilenumbruch.
<code>\t</code>	Horizontaler Tabulator
<code>\v</code>	Vertikaler Tabulator
<code>\n</code>	Zeilenumbruch
<code>\r</code>	Wagenrücklauf
<code>\f</code>	Seitenumbruch
<code> </code>	Trennt zwei alternative Teilausdrücke (ODER).

Tabelle 17 Sonderzeichen in regulären Ausdrücken

Hinweis:

Escape-Sequenzen für beliebige Zeichencodes in der Form `\xhhh` oder `\uhhhh` werden nicht direkt unterstützt. Allerdings lassen sich ‚normale‘ B++-Escape-Sequenzen in regulären Ausdrücken einsetzen, sofern ein Pattern nicht als Raw-String-Literal notiert wird.

Etwa könnte man in einem String-Literal „`\x20`“ (in B++-Notation) schreiben um ein einzelnes Leerzeichen zu adressieren, während „`\\s`“ für ein beliebiges Whitespace-Zeichen stünde.

Annahmen (Assertions)

Auch dies sind Zeichen oder Escape-Sequenzen, die jedoch eine besondere Interpretation eines benachbarten Zeichens festlegen.

Es sind folgende Assertions definiert:

Zeichen	Bedeutung
^	Das folgende Zeichen steht am Textanfang.
\$	Das davor notierte Zeichen steht am Textende.
\b	Das davor notierte Zeichen gehört zu einem Wort und das folgende nicht (oder umgekehrt); es wird also eine Wortgrenze verlangt.
\B	Dies ist die Umkehrung von \b – die benachbarten Zeichen gehören also beider zu einem Wort oder nicht.

Tabelle 18 Assertions in regulären Ausdrücken

Klammern

Klammern werden zur Gruppierung von Teilausdrücken, für die Definition benutzerdefinierter Zeichenklassen sowie für benutzerdefinierte Quantifizierer verwendet. Tabelle 19 zeigt die Varianten:

Klammerpaar	Bedeutung
()	Es wird ein gespeicherter Teilausdruck definiert. Der Inhalt einer darauf passenden Teilzeichenkette kann explizit abgefragt werden.
(?:)	Es wird ein nicht gespeicherter Teilausdruck definiert, z. B. in Verbindung mit Alternativen ().
[]	Eckige Klammern definieren benutzerdefinierte Zeichenklassen..
{ }	Geschwungene Klammern dienen der Definition von Quantifizierern.

Tabelle 19 Klammern in regulären Ausdrücken

Hinweis:

Aus der Verwendung der T-Rex-Bibliothek ergibt sich eine Einschränkung bei der Verwendung gruppierter Teilausdrücke. Verwendet man gespeicherte Teilausdrücke zusammen mit Alternativen (durch | getrennt) so werden die Submatch-Gruppen nur für die erste Alternative gefüllt.

Zeichenklassen

Zeichenklassen definieren einen Zeichensatz, dem ein innerhalb eines Ausdrucks gültiges (einzelnes) Zeichen angehören kann.

Dabei ist zwischen *vordefinierten* und *benutzerdefinierten* Klassen zu unterscheiden.

Die vordefinierten Zeichenklassen sind in Tabelle 20 zusammengestellt.

Symbol	Bedeutung
\l	Kleinbuchstaben
\u	Großbuchstaben
\a	Buchstaben (Vereinigungsmenge von \l und \u)
\A	Alle Zeichen, die keine Buchstaben sind
\w	Wort-Zeichen. Dies beinhaltet alle Buchstaben sowie Dezimalziffern und den Unterstrich (_).
\W	Umkehrung von \w, also alle Zeichen, die nicht zu \w gehören
\s	Whitespaces, hierzu gehören Leerzeichen, horizontaler und vertikaler Tabulator, Zeilenumbruch, Wagenrücklauf und Seitenumbruch.
\S	Umkehrung von \s, also alle Zeichen, die nicht zu \s gehören.
\d	Dezimalziffern
\D	Umkehrung von \d, also alle Zeichen, die keine Dezimalziffern sind.
\x	Hexadezimalziffern, also Dezimalziffern sowie a .. f und A .. F.
\X	Umkehrung von \x, also alle Zeichen, die keine Hexadezimalziffern sind.
\c	Steuerzeichen, hierzu gehören alle Zeichen mit einem Zeichencode zwischen 0x00 und 0x1f sowie der Zeichencode 0x7f (Del).
\C	Umkehrung von \c, also alle Zeichen, die keine Steuerzeichen sind.
\p	Sonderzeichen, beinhaltet alle Zeichen, die nicht zu den Klassen \a, \s oder \c gehören.
\P	Umkehrung von \p, also alle Zeichen, die keine Sonderzeichen sind.

Tabelle 20 Vordefinierte Zeichenklassen

Hinweis:

Für die Bildung der Standardklassen werden intern Funktionen der C-Bibliothek verwendet. Daraus können sich Abweichungen auf Grund der aktuellen Lokalisierungseinstellungen ergeben.

Benutzerdefinierte Zeichenklassen werden nach der Syntax

```
\[ \ [^] character set \]
```

gebildet.

Es wird also eine in eckige Klammern eingeschlossene Menge von Zeichen notiert, die der jeweiligen Klasse angehören. Wird dieser Aufzählung ein ^-Symbol vorangestellt, gehören zur Zeichenklasse alle Zeichen, die nicht in der Aufzählung enthalten sind.

Neben einzelnen Zeichen können in *character set* auch Zusammenfassungen aufeinander folgender Zeichen in der Form *start-end*, Sonderzeichen aus Tabelle 17 sowie vordefinierte Zeichenklassen aus Tabelle 20 stehen.

In Tabelle 21 sind einige Beispiele benutzerdefinierter Zeichenklassen angegeben.

Definition	Bedeutung
[0-8]	Oktalziffern
[\d\.]	Dezimalziffern oder ein Punkt
[ÄäÖöÜü]	Umlaute
[^a-zAÖÜß]	Alle Zeichen, die keine Kleinbuchstaben (einschließlich deutscher Umlaute und ß) sind.
[/\[\]\(\)\{\}\<>]	Schrägstrich (Slash), Backslash und Klammern
[+-* /]	Operatoren der Grundrechenarten
[^]	Beliebiges Zeichen außer dem Leerzeichen
[\s\n]	Whitespace oder Zeilenumbruch

Tabelle 21 Beispiele benutzerdefinierter Zeichenklassen

Quantifizierer

Quantifizierer bestimmen, wie oft ein bestimmtes einzelnes Zeichen oder ein Teilausdruck an einer bestimmten Stelle innerhalb eines regulären Ausdrucks vorkommen darf bzw. muss. Quantifizierer stehen immer unmittelbar hinter dem Symbol (Zeichen oder Teilausdruck), auf das sie sich beziehen.

Die verfügbaren Quantifizierer und ihre Bedeutung sind in Tabelle 22 zusammengestellt.

Symbol	Häufigkeit des Vorkommens
*	Das Symbol kommt nicht oder beliebig oft vor.
+	Das Symbol kommt mindestens einmal vor.
?	Das Symbol kommt höchstens einmal vor.
{ <i>n</i> }	Das Symbol kommt genau <i>n</i> -mal vor.
{ <i>n</i> , }	Das Symbol kommt mindestens <i>n</i> -mal vor.
{ , <i>m</i> }	Das Symbol kommt höchstens <i>m</i> -mal vor.
{ <i>n</i> , <i>m</i> }	Das Symbol kommt mindestens <i>n</i> -mal und höchstens <i>m</i> -mal vor.

Tabelle 22 Quantifizierer

Hinweis:

In obiger Tabelle sind *n* und *m* natürliche Dezimalzahlen, wobei $n \leq m$ gilt.

12.2 Die vordefinierte Klasse RegEx

Die Klasse `RegEx` stellt die Unterstützung regulärer Ausdrücke für B++ bereit. Sie stellt einen Wrapper um die interne Klasse `BppRegEx` dar (vgl. Referenzdokumentation) und kann als Beispiel für die in Abschnitt 15.2.2.2 (S. 191) beschriebene Vorgehensweise zur Implementierung von Bibliotheksklassen gelten.

Nachfolgend werden die öffentlichen Elemente der Klasse `RegEx` im Sinne einer kurzen Referenz erklärt.

Hinweis:

Im Gegensatz zu den meisten vordefinierten Funktionen (siehe Kapitel 11) verhalten sich Argumente der öffentlichen Methoden von `Regex` wie statisch typisierte Argumente von B++-Funktionen. Das bedeutet, dass sie, falls notwendig und möglich, implizite Typkonvertierungen durchführen. Im Falle ungültiger Argumente wird eine Ausnahme von `string`-Typ ausgelöst.

12.2.1 Methoden

Konstruktor

```
Regex(string pattern = null, string flags = null)
```

Der Konstruktor initialisiert eine Instanz (ein Objekt) der Klasse `Regex`.

Parameter:

- `pattern`: Suchmuster (Pattern), mit dem das Objekt initialisiert werden soll. Ist das Argument angegeben und nicht leer, so wird implizit die Methode `compile` aufgerufen.
- `flags`: Optionen für die Anwendung des Suchmusters zur Textsuche. Die Zeichenkette kann die Buchstaben `,m'` (multiline), `,i'` (ignore case) und `,g'` (global search) enthalten; alle anderen ggf. vorhandenen Zeichen werden ignoriert.

Methode `compile`

```
bool compile(string pattern, string flags = null)
```

Die Methode übersetzt den in `pattern` angegebenen Ausdruck.

Parameter:

- `pattern`: Suchmuster (Pattern), das übersetzt werden soll.
- `flags`: Optionen für die Anwendung des Suchmusters zur Textsuche. Die Zeichenkette kann die Buchstaben `,m'` (multiline), `,i'` (ignore case) und `,g'` (global search) enthalten; alle anderen ggf. vorhandenen Zeichen werden ignoriert.

Rückgabewert:

Bei Erfolg `true` (1), sonst `false` (0). Liefert der Aufruf den Wert `false`, kann mit der Methode `getError` eine Fehlerbeschreibung abgerufen werden.

Hinweise:

- Der Aufruf von `compile` muss vor dem Aufruf aller anderen Elementfunktionen erfolgen. Implizit geschieht dies, wenn bereits im Konstruktor ein Pattern angegeben wird.
- Die Methode kann auch mehrfach aufgerufen werden. In dem Fall wird das Ergebnis der vorangegangenen Übersetzung (der kompilierte Ausdruck) durch den neuen ersetzt.

- Die in *flags* angegebenen Optionen werden zunächst nur gespeichert und haben keinen Einfluss auf die Übersetzung. Sie können bei Bedarf später durch einen Aufruf von `setFlags` geändert werden.

Methode `getError`

```
string getError()
```

Die Methode liefert die Fehlerbeschreibung des letzten Aufrufs von `compile`.

Rückgabewert:

Fehlerbeschreibung des letzten `compile`-Aufrufs. Wurde `compile` noch nicht aufgerufen oder war der Aufruf erfolgreich, ist das Resultat eine leere Zeichenkette.

Hinweis:

Der Rückgabewert ist eine Kopie der intern gespeicherten Meldung. Er kann also bei Bedarf vom Aufrufer ohne Seiteneffekte modifiziert werden.

Methode `getFlags`

```
string getFlags()
```

Die Methode liefert die aktuell eingestellten Optionen als Zeichenkette.

Rückgabewert:

Zeichenkette mit Optionen für die Anwendung zur Textsuche. Die Zeichenkette kann die Buchstaben `,m'` (multiline), `,i'` (ignore case) und `,g'` (global search) enthalten.

Hinweis:

Der Rückgabewert kann bei Bedarf vom Aufrufer ohne Seiteneffekte modifiziert werden.

Methode `setFlags`

```
void setFlags(string flags)
```

Die Methode legt Optionen für die Anwendung des regulären Ausdrucks zur Textsuche fest.

Parameter:

- `flags`: Optionen für die Anwendung des Ausdrucks zur Textsuche. Die Zeichenkette kann die Buchstaben `,m'` (multiline), `,i'` (ignore case) und `,g'` (global search) enthalten; alle anderen ggf. vorhandenen Zeichen werden ignoriert.

Hinweis:

Die Funktion kann (nach dem Aufruf von `compile` beliebig oft aufgerufen werden und beeinflusst das Verhalten nachfolgender Suchoperationen.

Methode match

```
bool match(string txt)
```

Die Methode wendet das Suchmuster auf den gesamten in *txt* übergebenen Text an, prüft also die Übereinstimmung von *txt* mit dem regulären Ausdruck.

Parameter:

- *txt*: Text, auf den das Suchmuster angewendet werden soll.

Rückgabewert:

true (1) im Fall einer Übereinstimmung, sonst *false* (0).

Hinweise:

- Da die Methode stets einen Vergleich des Suchmusters mit dem gesamten Text durchführt, sind die Flags *m* (multiline), und *g* (global search) hier bedeutungslos.
- Im Fall einer Übereinstimmung (*match*) können durch einen nachfolgenden Aufruf der Methode `getSubExpressions(vector)` ggf. vorhandene Submatches (als Resultat gespeicherter Teilausdrücke) abgefragt werden.

Beispiel 78 illustriert die Verwendung von *match* zur Überprüfung einer Mail-Adresse.

```
// checks if adr is a valid mail address
bool checkMailAddress(string adr)
{
    RegEx rex("\\w+[\\w\\.]*\\w+@[\\w\\.\\a+");
    return rex.match(adr);
}
```

Beispiel 78 Verwenden der Methode *match*

Methode search

```
bool search(string txt, uint start=0, int len=-1)
```

Die Methode sucht das erste Vorkommen einer Teilzeichenkette in *txt*, die dem Suchmuster entspricht.

Parameter:

- *txt*: Text, auf den das Suchmuster angewendet werden soll.
- *start*: Startposition für die Suche in *txt*.
- *len*: Länge der Teilzeichenkette in *txt*, in der nach einer Übereinstimmung gesucht werden soll; ist *len* negativ oder reicht über das Ende von *txt* hinaus, so wird bis zum Ende von *txt* gesucht.

Rückgabewert:

`true` (1) im Fall einer Übereinstimmung, sonst `false` (0).

Hinweise:

- Die Methode findet stets (und nur) die erste Übereinstimmung in `txt`, unabhängig davon, ob das `.g'-Flag` gesetzt ist.
- Ist das Resultat des Aufrufs `true`, so kann die Methode `getSubExpressions(vector)` aufgerufen werden, um Position und Inhalt der Übereinstimmung sowie ggf. vorhandene Submatches (Treffer für geklammerte Teilausdrücke im Pattern) zu bestimmen. Die Methode `next()` kann verwendet werden, um nach weiteren Übereinstimmungen zu suchen.

Beispiel 79 Veranschaulicht die typische Verwendung der Methoden `search` und `next`. Hier werden beide Methoden in einer `for`-Schleife benutzt um die in einer Zeichenkette vorkommenden Ziffern zu zählen. Die letzte Zeile verwendet außerdem die Methode `getError`. Hier wird eine Fehlermeldung ausgegeben, falls der Ausdruck, der `compile` als Argument übergeben wurde, nicht übersetzt werden konnte, also syntaktische Fehler enthält.

```
// Count number of digits in string
Regex rex;
if (rex.compile("\\d")) {
    int cnt = 0;
    for(bool ok = rex.search("abc123x4y5z"); ok; ok=rex.next()) ++cnt;
    print("Number of matches: ", cnt, "\n");
}
else print(getError(), "\n");
```

Beispiel 79 Verwendung von `Regex::search` und `Regex::next`

Methode `next`

`bool next()`

Die Methode wird normalerweise in Verbindung mit der Methode `search` verwendet. Sie findet die nächste Übereinstimmung für das Suchmuster – vgl. Beispiel 79.

Rückgabewert:

`true` (1) falls eine weitere Übereinstimmung existiert, sonst `false` (0).

Hinweise:

- Ist das Ergebnis des Aufrufs `true`, so kann die Methode `getSubExpressions(vector)` aufgerufen werden, um Position und Inhalt der Übereinstimmung sowie ggf. vorhandene Submatches (Treffer für geklammerte Teilausdrücke im Pattern) zu bestimmen.
- Wie `search` ist auch `next` unabhängig vom `.g'-Flag`.

Methode `getMatches`

```
uint getMatches(string txt, vector dst, uint start=0, int len=-1)
```

Die Methode findet alle Teilzeichenketten im Suchtext `txt`, die dem Suchmuster entsprechen.

Parameter:

- `txt`: Text, auf den das Suchmuster angewendet werden soll.
- `dst`: Container (`vector`), in dem die Resultate gespeichert werden sollen. Dabei wird für jeden Treffer eine Kopie der entsprechenden Teilzeichenkette an `dst` als neues Element angehängt.
- `start`: Startposition für die Suche in `txt`.
- `len`: Länge der Teilzeichenkette in `txt`, in der nach Übereinstimmungen gesucht werden soll; ist `len` negativ oder reicht über das Ende von `txt` hinaus, so wird bis zum Ende von `txt` gesucht.

Rückgabewert:

Anzahl der gefundenen und an `dst` angehängten Übereinstimmungen.

Hinweise:

- Das Verhalten der Methode hängt vom `,g'-Flag` ab. Ist es gesetzt, werden alle Übereinstimmungen in `txt` gefunden, andernfalls ist die Suche nachdem ersten Treffer beendet.
- Eventuell vor dem Aufruf bereits in `dst` enthaltene Elemente werden nicht verändert. Ist also `dst` vor dem Aufruf nicht leer, so ist der Rückgabewert kleiner als die Anzahl der Elemente in `dst`.

Methode `getMatchPositions`

```
uint getMatchPositions(string txt, vector dst, uint start=0, int len=-1)
```

Die Methode findet alle Teilbereiche im Suchtext `txt`, die dem Suchmuster entsprechen. Sie ähnelt in ihrem Verhalten der Methode `getMatches`. Der wesentliche Unterschied besteht darin, dass `getMatchPositions` nicht den Inhalt der gefundenen Übereinstimmungen, sondern deren Startpositionen in im Suchtext und Längen bestimmt.

Parameter:

- `txt`: Text, auf den das Suchmuster angewendet werden soll.
- `dst`: Container (`vector`), in dem die Resultate gespeichert werden sollen. Dabei wird für jeden Treffer ein Tupel (ebenfalls vom Typ `vector`) an `dst` als neues Element angehängt, das die Startposition des Treffers als erstes und dessen Länge als zweites Element enthält.
- `start`: Startposition für die Suche in `txt`.
- `len`: Länge der Teilzeichenkette in `txt`, in der nach Übereinstimmungen gesucht werden soll; ist `len` negativ oder reicht über das Ende von `txt` hinaus, so wird bis zum Ende von `txt` gesucht.

Rückgabewert:

Anzahl der gefundenen und an *dst* angehängten Übereinstimmungen.

Hinweise:

- Das Verhalten der Methode hängt vom ‚g‘-Flag ab. Ist es gesetzt, werden alle Übereinstimmungen in *txt* gefunden, andernfalls ist die Suche nachdem ersten Treffer beendet.
- Eventuell vor dem Aufruf bereits in *dst* enthaltene Elemente werden nicht verändert. Ist also *dst* vor dem Aufruf nicht leer, so ist der Rückgabewert kleiner als die Anzahl der Elemente in *dst*.
- Die im Ergebnis abgelegten Positionen der Treffer beziehen sich auf den Anfang von *txt* (und nicht auf den durch *start* gegebenen Beginn des Suchbereiches).

Methode `getSubExpressions` (Variante 1)

```
uint getSubExpressions(vector &dst)
```

Die Methode hängt Beschreibungsstrukturen für den aktuellen Treffer (eine Übereinstimmung mit dem Suchmuster) in einem Suchtext und alle ggf. vorhandenen Submatches (Treffer für geklammerte Teilausdrücke im Pattern) an den Ergebnisvektor *dst* an.

Parameter:

- *dst*: Container (`vector`), in dem die Resultate gespeichert werden sollen. Dabei wird für jeden Treffer ein Element vom Typ `dictionary` an *dst* als neues Element angehängt, das als Eintrag ‚pos‘ die Startposition des Treffers und als Eintrag ‚text‘ den betreffenden Text (als Kopie) enthält. Dabei ist das zuerst eingetragene Element der (vollständige) Treffer selbst, gefolgt von den Submatches entsprechend der im Pattern definierten Reihenfolge.

Rückgabewert:

Anzahl der an *dst* angehängten Elemente.

Hinweise:

- Die Methode wird im Anschluss an den (erfolgreichen) Aufruf von `search`, `next`, oder `match` verwendet.
- Eventuell vor dem Aufruf bereits in *dst* enthaltene Elemente werden nicht verändert. Ist also *dst* vor dem Aufruf nicht leer, so ist der Rückgabewert kleiner als die Anzahl der Elemente in *dst*.

Beispiel 80 illustriert die Verwendung von `getSubmatches`. Hier werden alle Ziffernfolgen gefunden, die unmittelbar auf einen Buchstaben folgen. Die Ausgabe beinhaltet die vollständigen Treffer sowie (als Submatch) die jeweils zugehörige Ziffernfolge mit der entsprechenden Zeichenposition im Suchtext. Zu beachten ist, dass hier in Zeile 5 der Ergebnis-Container *v* explizit als leerer Vektor initialisiert wird, um die Ergebnisse des vorangegangenen Schleifendurchlaufs zu entfernen. Dies ist erforderlich, weil in B++ keine lokalen Variablen auf Blockebene existieren (vgl. Abschnitt 5.1, S. 28). In Zeile 9 wird der Punkt-Operator für den Zugriff auf die Dictionary-Einträge der einzelnen Ergebnisse verwendet (vgl. Abschnitt 6.7, S. 46).

```

1 // Find numbers behind a letter
2 RegEx rex;
3 if (rex.compile("\\a(\\d+)")) {
4     for(bool ok = rex.search("abc123x4y5z");ok;ok=rex.next()) {
5         vector v = [];
6         uint cnt = rex.getSubExpressions(v);
7         for (uint i=0;i<cnt;++i) {
8             dictionary d = v[i];
9             print(i, " pos: ", d.pos, " text: ", d.text, "\n");
10        }
11    }
12 }
13 else print(getError(),"\n");

```

Beispiel 80 Bestimmen von Submatches mit `getSubExpressions`

Methode `getSubExpressions` (Variante 2)

```
uint getSubExpressions(string txt, vector dst, uint start=0, int len=-1)
```

Die Methode kombiniert die Aufrufe von `search`, `next` und `getSubExpressions` (in Variante 1). Sie iteriert über den in `txt` angegebenen Suchtext und trägt alle Treffer einschließlich ihrer Submatches in `dst` ein.

Parameter:

- `txt`: Text, auf den das Suchmuster angewendet werden soll.
- `dst`: Container (`vector`), in dem die Resultate gespeichert werden sollen. Dabei wird für jeden Treffer ein Element vom Typ `vector` an `dst` als angehängt, das seinem Inhalt nach dem Ergebnis des Aufrufs der Variante 1 von `getSubExpressions` entspricht.
- `start`: Startposition für die Suche in `txt`.
- `len`: Länge der Teilzeichenkette in `txt`, in der nach Übereinstimmungen gesucht werden soll; ist `len` negativ oder reicht über das Ende von `txt` hinaus, so wird bis zum Ende von `txt` gesucht.

Rückgabewert:

Anzahl der an `dst` angehängten Elemente (Anzahl gefundener Übereinstimmungen).

Hinweise:

- Die Methode findet stets alle Übereinstimmungen in `txt`, unabhängig davon, ob das `g'`-Flag gesetzt ist.
- Eventuell vor dem Aufruf bereits in `dst` enthaltene Elemente werden nicht verändert. Ist also `dst` vor dem Aufruf nicht leer, so ist der Rückgabewert kleiner als die Anzahl der Elemente in `dst`.

Beispiel 81 verwendet die zweite Variante von `getSubmatches` für die gleiche Aufgabe wie in Beispiel 80 und benutzt `foreach` (siehe 11.1.2, S. 92) mit anonymen Funktionen (vgl. 9.7, S. 63) zur Iteration durch die Ergebnis-Container.

```

1 // Find numbers behind a letter
2 RegEx rex;
3 if (rex.compile("\\a(\\d+)")) {
4     vector dst; // stores results
5     rex.getSubExpressions("abc123x4y5z",dst)
6     foreach(dst,function(k,v) {
7         foreach(v, function(i,d) {
8             print(i, " pos: ", d.pos, " text: ", d.text, "\\n");
9         });
10    });
11 }
12 }
13 else print(getError(),"\\n");

```

Beispiel 81 Verwenden der zweiten Variante `getSubExpressions`

Methode `replace`

`string replace(string txt, string repl)`

Die Methode erzeugt eine neue Zeichenkette aus `txt`, wobei eine Ersetzung von Übereinstimmungen des Suchmusters mit dem in `repl` spezifizierten Ersatztext vorgenommen wird.

Parameter:

- `txt`: Text, auf den die Ersetzung angewendet werden soll.
- `repl`: Ersatzausdruck; dieser Ausdruck ist eine Zeichenkette, die optional folgende Platzhalter/Variablen enthalten kann:
 - `$&` oder `$0` : gesamter Text der Übereinstimmung
 - `$`` (back tick) : gesamter Text vor der Übereinstimmung
 - `$'` (Apostroph) : gesamter Text hinter der Übereinstimmung
 - `$n` : Text des n -ten Submatches, wobei n eine positive Dezimalzahl ist. Ist n größer als die Anzahl der vorhandenen (im Pattern definierten) Submatches, wird die Zeichenkettenrepräsentation von n eingefügt.
 - `$$` : fügt das Dollarzeichen selbst ein

Rückgabewert:

Ergebnis der Ersetzungsoperation als neue Zeichenkette.

Hinweis:

Das Verhalten der Methode hängt vom `'g'`-Flag ab. Ist es gesetzt, werden alle Übereinstimmungen in `txt` ersetzt, andernfalls ist das Ersetzen nachdem ersten Treffer beendet.

Methode `strReplace`

```
uint strReplace(string txt, string repl)
```

Die Methode ersetzt Übereinstimmungen des Suchmusters in `txt` durch den in `repl` spezifizierten Ersatztext.

Parameter:

- `txt`: Text, auf den die Ersetzung angewendet werden soll.
- `repl`: Ersatzausdruck; dieser Ausdruck ist eine Zeichenkette, die optional folgende Platzhalter/Variablen enthalten kann:
 - `$&` oder `$0` : gesamter Text der Übereinstimmung
 - `$`` (back tick) : gesamter Text vor der Übereinstimmung
 - `$'` (Apostroph) : gesamter Text hinter der Übereinstimmung
 - `$n` : Text des n -ten Submatches, wobei n eine positive Dezimalzahl ist. Ist n größer als die Anzahl der vorhandenen (im Pattern definierten) Submatches, wird die Zeichenkettenrepräsentation von n eingefügt.
 - `$$` : fügt das Dollarzeichen selbst ein

Rückgabewert:

Anzahl der durchgeführten Ersetzungen.

Hinweise:

- Das Verhalten der Methode hängt vom `g`-Flag ab. Ist es gesetzt, werden alle Übereinstimmungen in `txt` ersetzt, andernfalls ist das Ersetzen nachdem ersten Treffer beendet.
- Die Zeichenkette in `txt` wird durch die Ersetzung modifiziert. Um stattdessen eine neue Zeichenkette zu erzeugen, sollte `replace` verwendet werden.

12.2.2 Operatoren

Funktionsoperator

```
bool operator()(string txt, uint start=0, int len=-1)
```

bzw.

```
bool operator()()
```

Der Operator ist ein Alias für die Methode `search` (Variante mit Argumenten) bzw. `next` (parameterlose Form) und dient der syntaktische Vereinfachung von Iterationen über Matches (Übereinstimmungen des Suchmusters mit Teilzeichenketten in `txt`).

Parameter:

- `txt`: Text, auf den das Suchmuster angewendet werden soll.
- `start`: Startposition für die Suche in `txt`.

- `len`: Länge der Teilzeichenkette in `txt`, in der nach einer Übereinstimmung gesucht werden soll; ist `len` negativ oder reicht über das Ende von `txt` hinaus, so wird bis zum Ende von `txt` gesucht.

Rückgabewert:

`true` (1) falls eine (weitere) Übereinstimmung existiert, sonst `false` (0).

Hinweis:

Der Operator verhält sich exakt wie die Methoden `search` bzw. `next`. Auch hier kann nach einem erfolgreichen Aufruf (Ergebnis `true`) die Methode `getSubExpressions(vector)` aufgerufen werden, um Position und Inhalt der Übereinstimmung sowie ggf. vorhandene Submatches (Treffer für geklammerte Teilausdrücke im Pattern) zu bestimmen.

Beispiel 82 modifiziert den Code aus Beispiel 79 so, dass statt der Methoden `search` und `next` der Funktionsoperator verwendet wird.

```
// Count number of digits in string using operator()
Regex rex;
if (rex.compile("\\d")) {
    int cnt = 0;
    for(bool ok = rex("abc123x4y5z"); ok; ok=rex()) ++cnt;
    print("Number of matches: ",cnt, "\n");
}
else print(getError(), "\n");
```

Beispiel 82 Iteration über Treffer mit dem Funktionsoperator

Property-Operator (Punktoperator)

`var operator.(string key)` (Lesezugriff)

bzw.

`var operator.=(string key, var value)` (Schreibzugriff)

Der Operator dient dem vereinfachten Zugriff auf einige Eigenschaften eines `Regex`-Objekts.

Parameter:

- `key`: Property-Bezeichner
- `value`: neuer Wert der durch `key` bezeichneten Eigenschaft (Schreibzugriff)

Rückgabewert:

- Wert der durch `key` bezeichneten Eigenschaft (Lesezugriff)
- zugewiesener Wert (Schreibzugriff)

Die möglichen Bezeichner für `key` sowie die zugehörigen Rückgabe- bzw. Argumenttypen sind in Tabelle 23 zusammengestellt.

Lesezugriff		
Schlüssel (key)	Rückgabetyt	Beschreibung
flags	string	Flags als Zeichenkette, kann ,g', ,m' und ,i' enthalten
global	bool	true, wenn ,g'-Flag gesetzt
ignoreCase	bool	true, wenn ,i'-Flag gesetzt
multiline	bool	true, wenn ,m'-Flag gesetzt
source	string	Kopie des Pattern-Strings des regulären Ausdrucks
text	string	Kopie des Textes, der für die letzte Suche verwendet wurde
pos	uint	Startposition für die nächste Suche
len	int	Länge des Suchbereichs
Schreibzugriff		
Schlüssel (key)	Typ	Beschreibung
flags	string	Setzen aller als Zeichenkette (wie <code>setFlags</code>)
global	bool	Setzen/Rücksetzen des ,g'-Flags
ignoreCase	bool	Setzen/Rücksetzen des ,i'-Flags
multiline	bool	Setzen/Rücksetzen des ,m'-Flags
text	string	Setzen des Textes für die nächste Suchoperation
pos	uint	Setzen der Startposition für die nächste Suche
len	int	Setzen der Länge des Suchbereichs

Tabelle 23 Pseudo-Properties der Klasse RegEx

Hinweise:

- Das Setzen von Eigenschaften, die Flags beeinflussen, wirkt sich – analog zu `setFlags` – auf die nächstfolgende Suchoperation aus.
- Die Eigenschaften `text`, `pos` und `len` entsprechen den Argumenten der `search`-Methode. Man könnte damit die entsprechenden Werte explizit setzen und für die anschließende Suche ausschließlich `next()` bzw. den parameterlosen Funktionsoperator aufrufen.

13 Speicherverwaltung

In Abschnitt 5.2 wurde eine Unterscheidung der in B++ vorkommenden Datentypen nach Wert- und Referenztypen vorgenommen.

Während Instanzen der Werttypen unmittelbar an die Existenz der sie tragenden Value-Objekte gebunden sind, werden dort für die Referenztypen nur Verweise auf einen zugehörigen Datenbereich abgelegt. Damit stellt sich die Frage nach der Verwaltung des für die eigentlichen Daten benötigten (dynamischen) Speichers.

Einen Sonderfall stellt dabei zunächst der Typ `FILE` dar, dessen Instanzen lediglich einen Verweis auf eine vom Betriebssystem bereitgestellte Beschreibungsstruktur beinhalten, die mit Hilfe der Standardfunktionen `fopen` und `fclose` (vgl. Abschnitt 11.3) angefordert bzw. freigegeben wird.

Für alle anderen Referenztypen verwendet B++ standardmäßig eine automatische Speicherverwaltung mit Referenzzählung⁵⁸. Das heißt, jeder Wert eines Referenztyps trägt einen internen Referenzzähler, der bei einer Zuweisung an eine Variable inkrementiert und bei der Freigabe (wenn die Variable ihren Gültigkeitsbereich verlässt oder ihr ein anderer Wert zugewiesen wird) dekrementiert wird. Erreicht ein Referenzzähler dabei den Wert Null, so bedeutet dies, dass der betreffende Wert von keiner Stelle im Programm mehr erreichbar ist. In diesem Fall wird er (und mit ihm der für den Datenbereich verwendete Speicher) automatisch freigegeben.

Durch diese Verhaltensweise ist die Lebensdauer von Variablen in B++ streng determiniert. Insbesondere ist garantiert, dass beim Verlassen einer Funktion oder beim Auslösen einer Ausnahme (vgl. Abschnitt 8.4) alle temporären Variablen im jeweiligen Kontext freigegeben und damit implizit die Destruktoren von Objekten aufgerufen werden.

Beispiel 83 illustriert dies:

```
class A {
    static uint _id = 0;
    uint id;
    A() { id = ++_id; print(__func__, " id:", id, "\n"); }
    ~A() { print(__func__, " id:", id, "\n"); }
    operator string() { return newstring("class: %s id: %u",
                                         getclassname(this), id); }
}

void testfunc() {
    A a;
}

void testfunc2() {
    A a;
    throw new A();
}

main()
{
    try {
        print("enter testfunc\n");
        testfunc();
        print("enter testfunc2\n");
        testfunc2();
    }
    catch(e) {
        print(e, " caught\n");
    }
}
```

Beispiel 83 Objektfreigabe beim Verlassen des Gültigkeitsbereichs

⁵⁸ Im Unterschied zu vielen anderen Skriptsprachen verwendet B++ aus Gründen der Laufzeiteffizienz keinen echten Garbage-Collector.

Hier werden zwei Funktionen (`testfunc` bzw. `testfunc2`) definiert, die jeweils ein Objekt der Klasse `A` anlegen. Während `testfunc` einfach zum Aufrufer zurückkehrt, wirft `testfunc2` eine Ausnahme mit einer weiteren Instanz der Klasse `A` als Ausnahmeobjekt. Im Hauptprogramm werden beide Funktionen in einem geschützten Abschnitt (`try-catch`) aufgerufen, wobei folgende Ausgabe entsteht:

```
enter testfunc
A::A id:1
A::~dtor id:1
enter testfunc2
A::A id:2
A::A id:3
A::~dtor id:2
class: A id: 3 caught
A::~dtor id:3
```

Das in `testfunc` erzeugte Objekt (ID=1) wird beim Verlassen der Funktion freigegeben. In `testfunc2` wird zunächst ebenfalls ein Objekt lokal angelegt (ID=2) und anschließend eine weitere Instanz von `A` (ID=3) als Ausnahmeobjekt erzeugt. Die automatische Freigabe des lokalen Objekts (ID=2) erfolgt nun beim Verlassen von `testfunc2` (unmittelbar nach dem Auslösen der Ausnahme), während das Ausnahmeobjekt (ID=3) beim Verlassen des `catch`-Blocks freigegeben wird.

Hinweis:

In Beispiel 83 könnte `testfunc2` das lokale Objekt `a` (ID=2) auch direkt als Ausnahmeobjekt verwenden. In diesem Fall würde beim Aufruf von `throw` dessen Referenzzähler inkrementiert, so dass eine Freigabe erst beim Austritt aus dem `catch`-Block in `main` erfolgt.

13.1 Weak References und die unären Operatoren `&` bzw. `*`

Ein Problem bei der Speicherverwaltung per Referenzzählung stellen zyklische Referenzen dar, wie in Abbildung 3 (oben) anhand einer doppelt verketteten Liste illustriert wird:

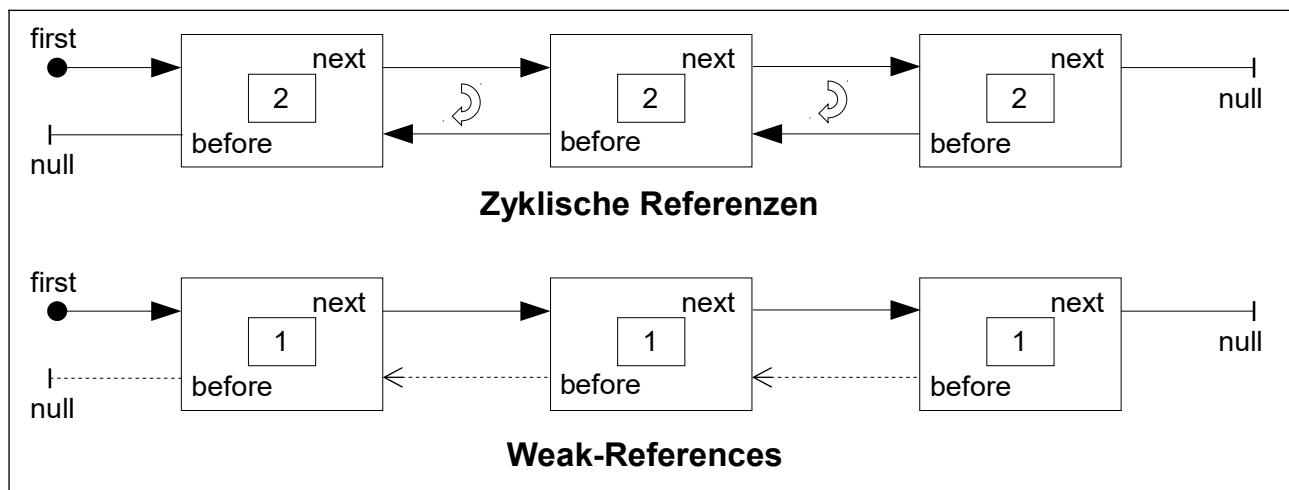


Abbildung 3 Zyklische Referenzen und deren Auflösung

Durch die wechselseitigen Verweise zwischen den Listenelementen haben die Referenzzähler der Elemente jeweils den Wert Zwei. Gibt man den Verweis auf den Listenanfang (die Variable `first`) frei, wird der Referenzzähler des ersten Elements zwar vermindert, erreicht aber nicht den Wert Null, so dass das Element (bzw. die ganze Liste) nicht freigegeben wird, obwohl vom Programm aus keine Möglichkeit des Zugriffs darauf mehr besteht.

Abhilfe schaffen hier Verweise, die explizit von der Referenzzählung ausgenommen werden (oft auch als *Weak References* bezeichnet), wie im unteren Teil von Abbildung 3 dargestellt.

In B++ werden solche Verweise mit dem unären Operator & erzeugt (vgl. 6.7), der den nachgestellten Wert bei der anschließenden Zuweisung an eine Variable für das jeweilige Zuweisungsziel von der Referenzzählung ausnimmt und so eine Weak Reference erzeugt.

Der unäre Operator * ist das Gegenstück dazu. Er wandelt eine Weak Reference wieder in eine „echte“ Referenz um, d. h., eine Variable, an die das Resultat zugewiesen wird, wird wieder in die Referenzzählung einbezogen. Beispiel 84 demonstriert die praktische Anwendung von Weak References etwas ausführlicher.

```
// List class example
// -----
// List element
class ListElement
{
private:
    var _prev = null;
    ListElement _next = null;
    var _value = null;
public:
    // ctor
    ListElement( var v = null; ) { _value = v; }
    // Define operator. for member access
    var operator.(string key) {
        switch (key)
        {
            case "prev": return _prev;
            case "next": return _next;
            case "value": return _value;
        }
        throw "invalid key " + key;
    }
    void operator.=(string key, var val) {
        switch (key)
        {
            case "prev":
                prev = &val;
                break;
            case "next":
                _next = val;
                break;
            case "value":
                _value = val;
                break;
            default:
                throw "invalid key " + key;
        }
    }
}

// List iterator class
class List;
class List_iterator {
private:
    var _lst = null;
    var _item = null;
public:
    // ctor
    List_iterator(List lst, var item = null) { _lst = &lst; _item = &item; }
    // Increment
    List_iterator operator++() {
        if (_item) _item = &_item.next;
    }
}
```

```

        else _item = &_lst.first();
        return this;
    }
    // Post-Increment
    List_iterator operator++(int) {
        List_iterator it(_lst, _item);
        return ++it;
    }
    // Decrement
    List_iterator operator--() {
        if (_item) _item = &_item.prev;
        else _item = &_lst.last();
        return this;
    }
    // Post-Decrement
    List_iterator operator--(int) {
        List_iterator it(_lst, _item);
        return --it;
    }
    var item() { return _item; }
    var lst() {
        return _lst;
    }
    // Equality operators
    bool equals(List_iterator other) {
        if (_lst != other->(List_iterator::lst)()) return false;
        return _item == other->(List_iterator::item)();
    }
    // Gets referenced list item.
    ListElement operator*() { return *_item; }
    // Delegates operator. to referenced list element
    var operator.(string key) { return _item->(ListElement::OP_PREF)(key); }
    void operator.=(string key, var val) {
        _item->(ListElement::OP_PSET)(key, val);
    }
}

// List class
class List
{
private:
    ListElement _first = null;
    var _last = null;
    uint _size = 0;
    // Internal helpers (protected methods)
    // Initialization
    void _init (var first, var last, uint siz) {
        _first = first; _last = last; _size = siz;
    }
    // Removes an element.
    ListElement _remove(var el) {
        if (!_size) throw "List is empty";
        if (!el) throw "Invalid argument";
        if (!el.prev) {
            var f = _first;
            _first = el.next;
        }
        else {
            var e = el.prev.next;
            el.prev.next = el.next;
        }
        if (!el.next) _last = el.prev;
        else el.next.prev = el.prev;
        --_size;
    }

```

```

        return el.next;
    }
    // Inserts an element
    ListElement _insert(var val, var beforeEl) {
        ListElement el(val);
        if (!beforeEl) {
            if (_last) {
                el.prev = _last;
                _last.next = el;
            }
            else _first = el;
            _last = &el;
        }
        else {
            if (!_size) throw "List is empty";
            el.next = beforeEl;
            if (!beforeEl.prev) _first = el;
            else {
                beforeEl.prev.next = el;
                el.prev = beforeEl.prev;
            }
            beforeEl.prev = &el;
        }
        ++_size;
        return el;
    }
public:
    // Ctor - allows initialization from vector
    List( vector init = null) {
        if (init) {
            uint n = size(init);
            for (uint i=0;i<n;++i) this->(_insert)(init[i],null);
        }
    }
    // STL-like functions
    ListElement front() { return _first; }
    ListElement back() { return _last ? *_last : null; }
    List_iterator begin() { return new List_iterator(this, _first); }
    List_iterator end() { return new List_iterator(this); }
    void push_back(var val) { this->(_insert)(val); }
    void push_front(var val) { this->(_insert)(val, _first); }
    void pop_back() { this->(_remove)(_last); }
    void pop_front() { this->(_remove)(_first); }
    List_iterator insert(List_iterator before, var val) {
        if (before->(List_iterator::lst)() != this) throw "Invalid iterator";
        return this->(_insert)(val, before->(List_iterator::item)());
    }
    List_iterator erase(List_iterator first, List_iterator last = null) {
        if (first->(List_iterator::lst)() != this) throw "Invalid iterator";
        if (!last)
            return new List_iterator(this, this->(_remove)(
                first->(List_iterator::item)()));
        if (last->(List_iterator::lst)() != this) throw "Invalid iterator";
        var item = first->(List_iterator::item)();
        var litem = last->(List_iterator::item)();
        while (item) {
            bool end = item == litem;
            item = &this->(_remove)(item);
            if (end) break;
        }
        return new List_iterator(this, item);
    }
    uint size() { return _size; }
    void clear() { _first = null; _last = null; _size = 0; }

```

```

void swap(List other)
{
    var ofirst = other->(List::front)();
    var olast = other->(List::back)();
    uint osize = other->(List::size)();
    other->(List::_init)(_first, _last, _size);
    _first = ofirst; _last = &olast; _size = osize;
}
// Conversion to string
operator string() {
    string result="";
    for (var el = _first; el; el = el.next)
    {
        result += ::string(el.value);
        if (el.next) result += ", ";
    }
    return result;
}
}

main()
{
    List l([1,2,3,4,5,6,7]);
    List l2([9,8,7,6]);
    l->insert(l->end(),"00"); // appends "00" to l
    auto i = l->begin();
    ++i;
    auto j = clone(i);
    ++j;
    l->erase(i,++j); // removes second and third element from l
    for (auto it = l->(List::begin)(); it != l->(List::end)(); ++it)
        print(*it.value, "\n"); // prints contents of l
    for (it = l2->(List::begin)(); it != l2->(List::end)(); ++it)
        print(*it.value, "\n"); // prints contents of l2
    print(l, "\n", l2, "\n"); // prints l and l2 using string conversion
    print("-----\n");
    l->swap(l2); // exchanges contents of l and l2
    for (it = l->(List::begin)(); it != l->(List::end)(); ++it)
        print(*it.value, "\n");
    for (it = l2->(List::begin)(); it != l2->(List::end)(); ++it)
        print(*it.value, "\n");
    print(l, "\n", l2, "\n");
}

```

Beispiel 84 Implementierung einer doppelt verketteten Liste

Hier wird eine doppelt verkettete Liste in Anlehnung an die C++-Standardbibliothek implementiert. Das Beispiel illustriert darüber hinaus eine Reihe der in Kapitel 10 beschriebenen Techniken zur objektorientierten Programmierung, insbesondere die Definition benutzerdefinierter Operatoren sowie die statische (nicht virtuelle) Bindung von Elementfunktionen.

13.2 Explizite Speicherverwaltung

Neben dem Erzeugen von Weak-Referenzen zur Auflösung von zirkulären Referenzen kann der unäre &-Operator auch zum Erzeugen von Variablen bzw. Objekten (genauer: Werten, die eigentlich der Referenzzählung unterliegen) verwendet werden, die von der Referenzzählung ausgenommen sind und deshalb mit Hilfe des `delete`-Operators explizit freigegeben werden müssen. Dies kann in Ausnahmefällen sinnvoll sein, wenn die Zerstörung eines bestimmten Objekts zu einem genau festgelegten Zeitpunkt erfolgen soll, der vor der impliziten Zerstörung beim Verlassen des Gültigkeitsbereichs der den Wert haltenden Variablen liegt. Insbesondere trifft

dies auf globale (bzw. statische) Variablen zu, die im Normalfall erst bei der Freigabe der B++-VM zerstört werden, wobei u. U. Destruktoren von Objekten nicht mehr aufgerufen werden.

Eine explizit verwaltete Variable entsteht, indem man einen Wert eines Referenztyps unmittelbar bei der ersten Zuweisung an die Variable in eine Weak Reference umwandelt, z.B.:

```
var obj = &new MyClass();  
var vec = &newvector(1,2,3);
```

Werden so initialisierte Variablen nicht mehr benötigt, müssen sie mit Hilfe des Operators `delete` freigegeben werden:

```
delete obj;  
delete vec;
```

oder besser:

```
obj = delete obj;  
vec = delete vec;
```

Im Unterschied zu C++ gibt der `delete`-Operator in B++ stets den Wert `null` zurück, der der Variablen wieder zugewiesen werden kann, um eine spätere versehentliche Verwendung des nun ungültig gewordenen Objektverweises zu verhindern.

Hinweise:

- Wie alle Weak-Referenzen können explizit verwaltete Werte nicht mit statischer Typisierung verwendet werden.
- Es ist möglich, den Wert einer ursprünglich explizit verwalteten Variablen nachträglich durch Anwendung des Dereferenzierungsoperators `*` und Zuweisung an eine andere Variable in einen automatisch verwalteten Wert umzuwandeln, die Umkehrung gilt jedoch nicht. **Vorsicht:** Eine solche Dereferenzierung führt zur sofortigen Objektfreigabe (analog zum Aufruf von `delete`), wenn keine Zuweisung an eine Variable erfolgt.

14 Interne Datenstrukturen

In diesem Kapitel werden die wichtigsten internen Datenstrukturen von B++ erklärt. Die Darstellung dient einerseits der Dokumentation, andererseits sollte die Kenntnis der wichtigsten Datenstrukturen dem Leser die Realisierung von Erweiterungen an B++ (siehe hierzu Kapitel 15) erleichtern.

14.1 Globale Symboltabellen und Typen von Symbolen

Die Virtuelle Maschine (VM) von B++ verwendet zwei globale Symboltabellen, die während der Übersetzung vom Compiler aufgebaut werden – eine enthält die im Programm definierten Klassen, die zweite alle übrigen Symbole. Die getrennte Verwaltung der Klassen hat dabei vor allem einen technischen Hintergrund – sie vereinfacht das Laden vorkompilierter Bytecode-Module. Darüber hinaus ermöglicht sie die Verwendung globaler Symbole (Funktionen, Variablen), die namensgleich mit Klassen sind – siehe Abschnitte 5.5 und 10.10.9.

Die Einträge in den Symboltabellen bestehen wesentlich aus dem *Symbolbezeichner*, dem *Symboltyp* sowie einem *Wert*. Tabelle 24 gibt eine Übersicht der in B++ verwendeten Symboltypen.

Symboltyp	Numerischer Wert	Beschreibung
ST_NONE	0	Kein Symbol (Dictionary-Eintrag)
ST_CLASS	1	Klasse
ST_DATA	2	Membervariable einer Klasse, lokale Variable
ST_SDATA	3	Statische Variable, Klassenvariable, globale Variable
ST_FUNCTION	4	Elementfunktion einer Klasse
ST_SFUNCTION	5	Statische Methode, globale Funktion

Tabelle 24 Übersicht der Symboltypen

Die Symboltabellen sind auf Basis von Dictionaries implementiert (Klassen `BppDictionaryEntry` und `BppDictionary` – siehe Referenz). Der Symboltyp `ST_NONE` wird für Dictionary-Einträge verwendet, die nicht die Bedeutung von Symbolen haben, also einfach Daten innerhalb eines Programms repräsentieren.

14.2 Struktur von Klassen und Objekten

Klassen besitzen ebenfalls zwei Symboltabellen, eine (`_dataMembers`) mit den Definitionen der Datenelemente (Member- und Klassenvariablen) und eine (`_funcMembers`) mit den Definitionen der Methoden (statische Methoden und Elementfunktionen). Darüber hinaus besitzen sie eine Information über den Klassennamen, einen Verweis auf die Basisklasse (falls vorhanden) sowie einen Vektor (`_defaultValues`) mit Standardwerten und ggf. statischen Typinformationen der nicht statischen Membervariablen. Kopien der Standardwerte werden zur Initialisierung der Membervariablen beim Erzeugen einer neuen Objektinstanz verwendet (vgl. Abbildung 4).

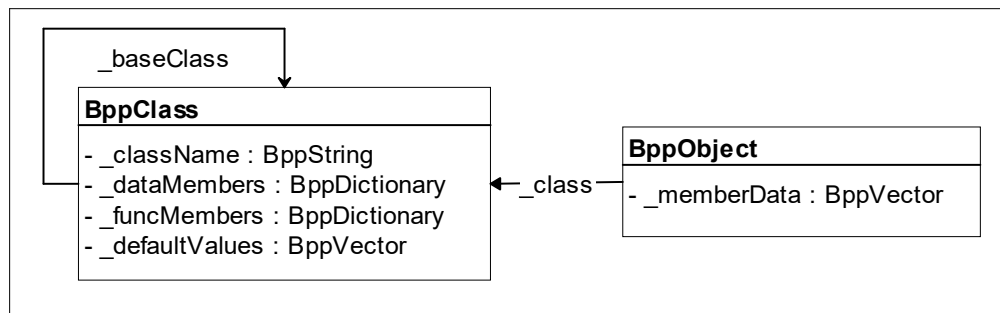


Abbildung 4 Struktur von Klassen und Objekten

In der `_dataMembers`-Tabelle bestehen die Werte der Einträge für Membervariablen jeweils aus einem zweielementigen Vektor, dessen erster Eintrag (Index 0) den Index der Variablen innerhalb des `_memberData`-Vektors der Klasseninstanzen enthält, während der zweite Eintrag den Initialwert und ggf. die statische Typinformation der Membervariablen aufnimmt. Der `_defaultValues`-Vektor wird vom Compiler nach Abschluss der Übersetzung der Klassendeklaration (nachdem alle Membervariablen definiert sind) aufgebaut. Er dient ausschließlich als „Kopiervorlage“ zur schnellen Erzeugung von Instanzen.

Objekte (Instanzen von Klassen) besitzen einen Verweis auf die sie definierende Klasse sowie einen Vektor aller (nicht statischen) Membervariablen. Beim Kopieren eines Objekts werden Klassenverweis und Membervariablen (letztere als tiefe Kopie) aus dem Quellobjekt übernommen⁵⁹.

14.3 Struktur von Funktionen und Aufbau des Bytecodes der VM

Für die Struktur von Funktionen und die Bytecode-Organisation übernimmt die VM von B++ mit einigen Erweiterungen das Konzept von BOB [2]. Die wichtigste Besonderheit dieses Konzepts besteht darin, dass die VM keinen globalen Codebereich kennt. Stattdessen existiert jeglicher ausführbarer Code innerhalb von Funktionen, die jeweils ihren eigenen Codebereich (Bytecode-Puffer) besitzen. Dadurch lässt sich eine Funktion als in sich geschlossenes Objekt auffassen und wie ein „gewöhnlicher“ Wert behandeln. Insbesondere entfällt dadurch die Notwendigkeit der Verwaltung von Einsprungs- und Rückkehradressen für Funktionsaufrufe.

Intern wird eine Funktion⁶⁰ in Form eines Vektors abgebildet, der jedoch die Typ-ID `BVT_BYTECODE` trägt (vgl. Tabelle 2 auf Seite 20). Der Aufbau dieses Vektors ist in Tabelle 25 zusammengefasst.

⁵⁹Der Verweis auf die Klasse wird im ersten Element (Index 0) des `_memberdata`-Vektors eines Objekts gespeichert, was seine automatische Sicherung und Wiederherstellung beim Speichern/Laden kompilierter Bytecode-Module gewährleistet.

⁶⁰Gemeint ist hier eine Bytecode-Funktion; externe „native“ Funktionen werden einfach durch einen Zeiger auf ihren Code repräsentiert.

Index	Symbol ⁶¹	Beschreibung						
0	IDX_CODE	Bytecode-Puffer (BppBuffer)						
1	IDX_CLASS	Klassenverweis (BppClass) bei Elementfunktionen, sonst null						
2	IDX_NAME	Funktionsname						
3	IDX_DEFAULTARGS	Vektor mit Standard-Argumentwerten						
4	IDX_SELFREFERENCE	Rückverweis (BVT_BYTECODEREF) auf die Funktion selbst						
...		Werte aller innerhalb der Funktion definierten Literale und lokalen statischen Variablen sowie Verweise auf innerhalb der Funktion referenzierte globale Variablen						
(last)		Vektor mit Debug-Informationen folgender Struktur <table><tr><th>Index</th><th>Beschreibung</th></tr><tr><td>0</td><td>Vektor der Namen aller formalen Parameter (in der Reihenfolge der Angabe in der Parameterliste)</td></tr><tr><td>1</td><td>Vektor der Namen der nicht-statischen lokalen Variablen; falls lokale statische Variablen existieren, enthält dieser Vektor als letztes Element ein Dictionary mit den Zuordnungen von deren Namen zu den zugehörigen Indizes im Bytecode-Vektor.</td></tr></table>	Index	Beschreibung	0	Vektor der Namen aller formalen Parameter (in der Reihenfolge der Angabe in der Parameterliste)	1	Vektor der Namen der nicht-statischen lokalen Variablen; falls lokale statische Variablen existieren, enthält dieser Vektor als letztes Element ein Dictionary mit den Zuordnungen von deren Namen zu den zugehörigen Indizes im Bytecode-Vektor.
Index	Beschreibung							
0	Vektor der Namen aller formalen Parameter (in der Reihenfolge der Angabe in der Parameterliste)							
1	Vektor der Namen der nicht-statischen lokalen Variablen; falls lokale statische Variablen existieren, enthält dieser Vektor als letztes Element ein Dictionary mit den Zuordnungen von deren Namen zu den zugehörigen Indizes im Bytecode-Vektor.							

Tabelle 25 Struktur des Bytecode-Vektors einer Funktion

Der Vektor der Standard-Argumentwerte enthält für statisch typisierte Argumente auch die Information über deren Datentyp. Die Anzahl und Reihenfolge seiner Elemente entspricht der der formalen Parameter bei der Funktionsdeklaration. Elemente, die Parameter ohne Vorgabewert repräsentieren, sind mit einem Flag als ungültig gekennzeichnet, sie werden lediglich zur Typprüfung beim Funktionsaufruf verwendet.

Der Rückverweis (`IDX_SELFREFERENCE`) wird intern für den rekursiven Aufruf anonymer Funktionen mittels `self` verwendet (siehe Abschnitt 9.7).

Das letzte Element mit den Debug-Informationen existiert nur, wenn beim Übersetzen der jeweiligen Funktion die Erzeugung von Debug-Informationen eingeschaltet war. Zur Identifizierung hat der darin enthaltene Vektor die Typ-ID `BVT_LOCALVARINFO`.

Der Bytecode-Puffer enthält die Bytecodes (*Opcode-Bytes*) der vom Compiler generierten Befehle sowie in einigen Fällen Zusatzdaten für Indizes und Typinformationen.

Ein Opcode-Byte besteht aus einem 6 Bit breiten Befehlscode (niederwertige Bits) sowie einer 2 Bit breiten Information über zusätzliche Daten. Letztere gibt die Länge eines auf den Opcode folgenden Datenwert an:

- 0 : Es folgen keine Daten.
- 1 : Es folgt ein Datenbyte.
- 2 : Es folgt ein 16-Bit-Datenwort (2 Bytes).
- 3 : Es folgt ein 32-Bit-Datenwort (4 Bytes).

⁶¹Die symbolischen Namen sind in den Quellen von B++ definiert und können bei der Programmierung von Erweiterungen für den Zugriff auf Elemente des Bytecode-Vektors verwendet werden. Sie sind keine Literale in B++-Programmen.

Aus diesem Aufbau ergibt sich, dass in einem Opcode-Byte höchstens 64 unterschiedliche Befehlscodes direkt kodiert werden können. Um diese Beschränkung aufzuheben, gibt es eine gesonderte Behandlung des Opcodes mit dem Wert Null (OP_NOP):

Ist beim Opcode Null der Wert der Dateninformation nicht Null, so wird ein 8-Bit breiter (erweiterter) Opcode als OR-Verknüpfung der Dateninformation (höchstwertige Bits) und dem Opcode-Anteil des nächstfolgenden Opcode-Bytes gebildet. Auf diese Weise können wieder bis zu 256 Opcodes erzeugt werden⁶². Von dieser Erweiterung macht B++ allerdings nur für vier spezielle Befehle Gebrauch, die überdies nur im Kontext von Debug-Ausgaben Verwendung finden⁶³.

Tabelle 26 Gibt einen Überblick über die Opcodes der B++-VM und ihre Bedeutung. In der Tabelle werden folgende Abkürzungen bzw. Symbole verwendet:

- PC : Befehlszähler (Index im Bytecode-Puffer)
- SP : Stack-Pointer
- FP : Frame-Pointer (Zeiger im Stack auf das Element vor der ersten lokalen Variablen einer Funktion)
- BP : Base-Pointer (Zeiger im Stack auf das Element vor dem ersten Argument einer Funktion)
- VP : Value-Pointer (temporärer Hilfszeiger auf ein Element im Stack)
- TYPE: Datentyp (Typ-ID) eines Wertes
- CODEVEC : Bytecode-Vektor der Funktion entsprechend Tabelle 25
- nargs : Anzahl der Argumente einer Funktion
- nvar : Anzahl lokaler Variablen einer Funktion
- Index : Indexwert
- MDATA : Vektor der Elementvariablen eines Objekts

Die in der Spalte „Symbol“ mit * gekennzeichneten Einträge bezeichnen Befehle, die für Objekte mit Hilfe gleichnamiger Elementfunktionen (Operatorfunktionen) explizit definiert werden können (vgl. Abschnitt 10.10). Mit ** gekennzeichnete Befehle (Vergleichsoperationen) können für Objekte indirekt mit Hilfe der Elementfunktionen `equals` bzw. `compare` definiert werden (siehe 10.10.7).

Code (hex)	Symbol	Daten	Beschreibung
00	OP_NOP	(erw. Befehl)	Keine Operation / Erweiterter Befehl
01	OP_BRT	PC*	Bedingter Sprung: Wenn [SP] == true → PC := PC*
02	OP_BRF	PC*	Bedingter Sprung: wenn [SP] == false → PC := PC*
03	OP_BR	PC*	Unbedingter Sprung: PC := PC*
04	OP_NIL	(TYPE)	Löschen des Wertes von SP und ggf. Setzen des statischen Typs: [SP] := null wenn TYPE != 0 → TYPE([SP]) := TYPE
05	OP_PUSH	(TYPE)	Leeren Wert auf den Stack legen und ggf. statischen Typ setzen: [++SP] := null wenn TYPE != 0 → TYPE([SP]) := TYPE

⁶²256 Befehle waren auch in der BOB-Urversion möglich, dort gab es aber keine Spezifikation für Folgedaten. Für einige Befehle wurde stattdessen implizit ein nachfolgendes Indexbyte angenommen, was z. B. adressierbare Feldgrößen auf 256 Einträge beschränkte.

⁶³Das Dekodieren der erweiterten Befehle im Bytecode-Interpreter ist geringfügig aufwendiger als das der einfachen Befehle. Deshalb ist ihre Verarbeitung theoretisch etwas langsamer, praktisch dürfte der Unterschied aber verschwindend sein.

Code (hex)	Symbol	Daten	Beschreibung
06	OP_NOT*		Logische Negation des Wertes bei SP: wenn [SP] == true \rightarrow [SP] = 0, sonst [SP] := 1
07	OP_NEG*		Numerische Negation des Wertes bei SP (unäres Minus): [SP] := - [SP]
08	OP_ADD*		Addition der beiden obersten Stack-Einträge: [SP-1] := [SP-1] + [SP] --SP
09	OP_SUB*		Subtraktion der beiden obersten Stack-Einträge: [SP-1] := [SP-1] - [SP] --SP
0A	OP_MUL*		Multiplikation der beiden obersten Stack-Einträge: [SP-1] := [SP-1] * [SP] --SP
0B	OP_DIV*		Division der beiden obersten Stack-Einträge: [SP-1] := [SP-1] / [SP] --SP
0C	OP_REM*		Divisionsrest der beiden obersten Stack-Einträge: [SP-1] := [SP-1] % [SP] --SP
0D	OP_BAND*		Bitweises AND der beiden obersten Stack-Einträge: [SP-1] := [SP-1] & [SP] --SP
0E	OP_BOR*		Bitweises OR der beiden obersten Stack-Einträge: [SP-1] := [SP-1] [SP] --SP
0F	OP_XOR*		Bitweises XOR der beiden obersten Stack-Einträge: [SP-1] := [SP-1] ^ [SP] --SP
10	OP_BNOT*		Bitweises NOT (Einerkomplement) des obersten Stack-Eintrags: [SP] := ~[SP]
11	OP_SHL*		Bitweises Linksschieben des vorletzten Stack-Eintrags um Wert des obersten Stack-Eintrags: [SP-1] := [SP-1] << [SP] --SP
12	OP_SHR*		Bitweises Rechtsschieben des vorletzten Stack-Eintrags um Wert des obersten Stack-Eintrags: [SP-1] := [SP-1] >> [SP] --SP
13	OP_LT**		Vergleich der obersten beiden Werte des Stacks (kleiner) wenn [SP-1] < [SP] \rightarrow [SP-1] := 1, sonst [SP-1] := 0 --SP
14	OP_LE**		Vergleich der obersten beiden Werte des Stacks (kleiner / gleich) wenn [SP-1] <= [SP] \rightarrow [SP-1] := 1, sonst [SP-1] := 0 --SP
15	OP_EQ**		Vergleich der obersten beiden Werte des Stacks (Gleichheit) wenn [SP-1] == [SP] \rightarrow [SP-1] := 1, sonst [SP-1] := 0 --SP

Code (hex)	Symbol	Daten	Beschreibung
16	OP_NE**		Vergleich der obersten beiden Werte des Stacks (Ungleichheit) wenn [SP-1] != [SP] → [SP-1] := 1, sonst [SP-1] := 0 --SP
17	OP_GE**		Vergleich der obersten beiden Werte des Stacks (größer / gleich) wenn [SP-1] >= [SP] → [SP-1] := 1, sonst [SP-1] := 0 --SP
18	OP_GT**		Vergleich der obersten beiden Werte des Stacks (größer) wenn [SP-1] > [SP] → [SP-1] := 1, sonst [SP-1] := 0 --SP
19	OP_INC*		Inkrementieren des obersten Stack-Eintrags: [SP] := [SP] + 1
1A	OP_DEC*		Dekrementieren des obersten Stack-Eintrags: [SP] := [SP] - 1
1B	OP_LIT	Index	Laden eines Literals in oberstes Stack-Element: [SP] := [CODEVEC[Index]]
1C	OP_RETURN	(TYPE)	Rückkehr aus Funktion, ggf. mit Typprüfung des Rückgabewertes [SP]
1D	OP_CALL*	nargs	Funktionsaufruf (nargs ist Argumentzahl): VP := SP-nargs Aufruf der Funktion bei [VP] [VP] := [SP] SP := VP
1E	OP_REF	Index	Laden des Wertes einer globalen Variablen: [SP] := [CODEVEC[Index]]->value
1F	OP_SET	Index	Setzen des Wertes einer globalen Variablen: CODEVEC[Index]]->value := [SP]
20	OP_VREF*		Laden des Wertes eines Containerelements: [SP-1] := [SP-1] [[SP]] --SP
21	OP_VSET*		Setzen des Wertes eines Containerelements: [SP-2] [[SP-1]] := [SP] [SP-2] := [SP] SP := SP-2
22	OP_MREF		Laden des Wertes einer Member-Variablen: [SP] := [[BP+1]->MDATA[[SP]]]
23	OP_MSET		Setzen des Wertes einer Member-Variablen [[BP+1]->MDATA[[SP]]] := [SP]
24	OP_AREF	Index	Laden eines Argumentwertes: [SP] := [BP+Index+1]
25	OP_ASET	Index	Setzen eines Argumentwertes: [BP+Index+1] := [SP]
26	OP_TREF	Index	Laden eines lokalen (temporären) Variablenwertes: [SP] := [FP+Index+1]
27	OP_TSET	Index	Setzen eines lokalen (temporären) Variablenwertes: [FP+Index+1] := [SP]
28	OP_TSPACE	nvar	Lokalen Variablenbereich reservieren: FP := SP SP := SP+nvar

Code (hex)	Symbol	Daten	Beschreibung
29	OP_SEND	nargs	Aufruf einer Elementfunktion eines Objekts: VP := SP-nargs Aufruf der Funktion bei [VP+1] auf Objekt bei [VP] [VP] := [SP] SP := VP
2A	OP_DUP2		Duplizieren der obersten beiden Stack-Einträge: SP := SP+2 [SP-1] := [SP-3] [SP] := [SP-2]
2B	OP_NEW		Erzeugen eines neuen Objekts (Klasseninstanz) [SP] := new BppObject([SP])
2C	OP_DELETE		Löschen eines Objekts: Wenn [SP] Weak-Reference mit Referenzzähler 0 → delete [SP] [SP] := null
2D	OP_DUP		Duplizieren des obersten Stack-Eintrags: ++SP [SP] := [SP-1]
2E	OP_POP		Entfernen des obersten Stack-Eintrags: --SP
2F	OP_MKREF		Umwandeln von [SP] in Weak-Reference
30	OP_TRY	PC' (catch)	Start try-Block: Installation ErrorHandler(PC',SP)
31	OP_ENDTRY	PC' (end catch)	Ende try-Block: PC := PC'
32	OP_THROW		Ausnahme auslösen: [FP+1] := [SP]
33	OP_ENDCATCH		Ende Ausnahmebehandlung: Deinstallation ErrorHandler [FP+1] := null (oder offene äußere Ausnahme)
34	OP_SWITCH	PC' (end switch)	Start switch-Anweisung: ohne Funktion
35	OP_SREF	Index	Laden des Wertes einer statischen lokalen Variablen: [SP]:= [CODEVEC[Index]]
36	OP_SSET	Index	Setzen des Wertes einer statischen lokalen Variablen: [CODEVEC[Index]] := [SP]
37	OP_PINC*		Inkrementieren des obersten Stack-Eintrags (post): [SP] := [SP] + 1
38	OP_PDEC*		Dekrementieren des obersten Stack-Eintrags (post): [SP] := [SP] + 1
39	OP_PREF*		Laden des Property-Wertes: [SP-1] := [SP-1] [[SP]] --SP
3A	OP_PSET*		Setzen eines Property-Wertes: [SP-2] [[SP-1]] := [SP] [SP-2] := [SP] SP := SP-2
3B	OP_POPC		Kopieren und Entfernen des obersten Stack-Eintrags: [SP-1] := [SP] --SP

Code (hex)	Symbol	Daten	Beschreibung
3C	OP_PLUS*		Unäres Plus: wenn [SP] vom Objekttyp → [SP] := obj->OP_PLUS() sonst wenn [SP] numerisch → keine Funktion sonst Typfehler
3D	OP_UNREF*		Umwandeln von [SP] von Weak-Reference in Wert
Erweiterte Befehle zur Debugging-Unterstützung			
FC	OP_TRON		Trace On
FD	OP_TROFF		Trace Off
FE	OP_TRSTEP		Einzelschrittbetrieb
FF	OP_NEWLINE		Neue Zeile im Quelltext

Tabelle 26 Übersicht der Opcodes der B++-VM

Hinweis:

Die in Tabelle 26 enthaltenen Befehlsbeschreibungen sind der besseren Übersicht wegen etwas vereinfacht wiedergegeben. Generell nicht dargestellt ist hier das Inkrementieren des Befehlszählers nach jeder Befehlsausführung (außer bei Sprüngen). Darüber hinaus sind Typprüfungen und Aufrufe von benutzerdefinierten Operatorfunktionen für Objekte nicht aufgeführt.

14.4 Organisation des Stacks der VM

Die B++-VM verwendet einen vollständig dynamischen (und damit theoretisch beliebig großen) Stack, der als Vektor (Klasse `BppVector`) implementiert ist und „nach oben“ wächst, indem neue Einträge an das Ende des Vektors angefügt werden.

Während der Ausführung einer B++-Funktion im Bytecode-Interpreter der VM werden drei spezielle Variablen zur Verwaltung von Stack-Einträgen verwendet:

- SP : Stack-Pointer (Zeiger auf den obersten Stack-Eintrag)
- FP : Frame-Pointer (Zeiger im Stack auf das Element vor der ersten lokalen Variablen einer Funktion)
- BP : Base-Pointer (Zeiger im Stack auf das Element vor dem ersten Argument einer Funktion)

Dabei sind diese Zeiger streng genommen wahlfreie Iteratoren auf dem Stack.

Beim Aufruf einer Funktion vollzieht sich (vereinfacht) folgender Ablauf:

- Die aktuellen Argumente des Aufrufs werden in der Reihenfolge ihrer Angabe auf den Stack gelegt. Im Falle eines Methodenaufrufs wird vor dem ersten Argument zusätzlich der Verweis auf das jeweilige Objekt auf dem Stack abgelegt.
- BP wird auf SP - nargs (Anzahl der Aufrufparameter) gesetzt und zeigt damit vor das erste Argument bzw. (im Falle eines Methodenaufrufs) auf das aufrufende Objekt.
- Ist die Zahl der übergebenen Argumente kleiner als die der deklarierten, werden dafür die Standardwerte aus der Funktionsdeklaration (sofern vorhanden, andernfalls wird ein Laufzeitfehler erzeugt) hinzugefügt.
- Sind die formalen Funktionsparameter statisch typisiert, erfolgt bei Bedarf (und sofern möglich) eine Typkonvertierung.
- FP wird auf den aktuellen Wert von SP gesetzt und zeigt damit vor die erste lokale Variable der Funktion⁶⁴.

⁶⁴Der Platz für die lokalen Variablen wird in jeder Bytecode-Funktion mit dem ersten ausgeführten Opcode (OP_TSPACE) reserviert. Dabei existiert stets mindestens eine solche Variable, adressierbar über FP[1], die für die Ausnahmebehandlung reserviert ist und einen Verweis auf eine noch unbehandelte Ausnahme aufnimmt.

- Es wird eine Hilfsstruktur (`BppProcInfo`⁶⁵) angelegt, die für die Behandlung von Ausnahmen sowie für das Debugging benötigt wird. Sie beinhaltet Verweise auf die aktuelle Funktion, deren Aufrufer sowie ggf. (bei Verwendung von `try-catch`) einen Errorhandler-Stack). Die `BppProcInfo`-Strukturen bilden ihrerseits einen navigierbaren Stack von Funktionsaufrufen.

Das Ergebnis – der Stack-Frame beim Eintritt in eine Funktion – ist in Tabelle 27 zusammengefasst.

Index	Beschreibung
BP	Objektverweis (bei Methoden)
BP+1	1. Argument
BP+nargs	letztes Argument
FP (=BP+nargs)	Zeiger vor 1.lokale Variable
FP+1	1. lokale Variable (reserviert für Exception)
...	weitere lokale Variablen
SP	aktueller Stack-Pointer

Tabelle 27 Stack-Frame beim Eintritt in eine Funktion

⁶⁵Siehe hierzu die Referenz.

15 Erweitern von B++

Der Satz an vordefinierten Funktionen, den B++ bereitstellt, sollte es prinzipiell ermöglichen, nahezu beliebige Anwendungen zu entwickeln. Allerdings ist sein Umfang auf das Nötigste begrenzt, da ein wichtiges Entwicklungsziel darin bestand, einen kleinen, überschaubaren und ressourcenschonenden Interpreter bereitzustellen. Eine größere Menge vordefinierter Funktionen hätte dem zwangsläufig entgegen gestanden. Hinzu kommt, dass es einerseits unmöglich ist, alle denkbare Funktionalität für beliebige Anwendungsfälle bereitzustellen, und andererseits in einer konkreten Anwendung meist nur eine verhältnismäßig geringe Anzahl von Funktionen tatsächlich benötigt wird.

15.1 B++-Bibliotheksmodule

Die einfachste Möglichkeit, wiederverwendbare Bibliotheken für spezifische Aufgaben bereitzustellen, ist deren direkte Implementierung in B++. Solche Bibliotheken können ihrerseits Systembibliotheken von Windows⁶⁶ dynamisch einbinden bzw. unter MS-DOS Interrupt-Aufrufe für den Zugriff auf Betriebssystemfunktionen verwenden. Außerdem können sie selbstverständlich die weiter unten beschriebenen ‚Native‘-Erweiterungen, die selbst dynamische Bibliotheken sind, verwenden.

Ein B++-Bibliotheksmodul wird wie ein gewöhnliches B++-Programm geschrieben, d. h., es besteht aus einer oder mehreren B++-Quelldateien, die beliebige Objekte (Klassen, Funktionen, Variablen) implementieren. Der einzige wesentliche Unterschied besteht darin, dass ein Bibliotheksmodul im Normalfall keine eigene Eintrittsfunktion (also kein ‚Hauptprogramm‘) bereitstellt.

Beispiel 85 zeigt einen Ausschnitt aus einem Bibliotheksmodul zur Dateiverwaltung⁶⁷.

```
// filesystem.bpp
/** Filesystem class.
Class provides some simple functions for handling of files and directories.<br>
It may be used for MS-DOS and Windows platforms without modifications.<br>
All necessary initializations are done by init method, called from constructor.
*/
class FileSys
{
public:
    FileSys(); ///< Ctor - calls init() if needed.
    bool chdir(string dirname); ///< Changes working directory.
    string getcwd(); ///< Gets current working directory.
    bool mkdir(string dirname); ///< Creates new directory.
    bool rmdir(string dirname); ///< Removes a directory.
    bool remove(string fname); ///< Removes file.
    bool copyfile(string src, string dst, bool overwrite = true); ///< Copies
file
    bool rename(string oldname, string newname); ///< Renames file or directory.
private:
    static void init(); ///< Initialization
    static function _winFunc = null;
    static function _int86 = null;
    static function _segread = null;
    static function _setreg = null;
    static function _getreg = null;
    /// Helper function for DOS interrupt call.
    static int doscall(dictionary regs)
    {
        _segread();
    }
}
```

⁶⁶Dies würde auch für eine Linux-Version von B++ gelten, die allerdings zum gegenwärtigen Zeitpunkt noch nicht verfügbar ist.

⁶⁷ Der vollständige Quelltext ist in der B++-Distribution enthalten.

```

        vector keys = dictgetkeys(regs);
        vector values = dictgetvalues(regs);
        uint n = size(keys);
        for (int i=0;i<n;++i) _setreg(keys[i],values[i]);
        int result = _int86(0x21);
        if (!_getreg("cflag")) return 0;
        return result;
    }
};

FileSys::FileSys() {
    static bool initialized = false;
    if (!initialized) {
        init();
        initialized = true;
    }
}

FileSys::init() {
    // Assign system functions depending on OS.
    if (__OSTYPE__ == "DOS") {
        _int86 = getsymbol("int86x");
        _segread = getsymbol("segread");
        _getreg = getsymbol("getreg");
        _setreg = getsymbol("setreg");
    }
    else {
        _winFunc = getsymbol("GetKernel32Func");
    }
}

/**
@param dirname Absolute or relative directory path.
@return true on success.
*/
bool FileSys::chdir(string dirname)
{
    intptr adr = addr(dirname);
    if (_winFunc)
    {
        return CallLibFunc(_winFunc("SetCurrentDirectory?"),adr) != 0;
    }
    dictionary regs = { ah:0x3B };
    regs.ds = adr >> 16;
    regs.dx = adr & 0xffff;
    return doscall(regs) == 0;
}

```

Beispiel 85 Implementierung einer Bibliothek zur Dateiverwaltung (Auszug)

Zu erwähnen ist hier, dass das Modul ohne Änderung (auch ohne Neuübersetzung) auf MS-DOS- und Windows-Plattformen verwendbar ist, obwohl von systemspezifischen Funktionen Gebrauch gemacht wird. Erreicht wird dies mit Hilfe der vordefinierten Variablen `__OSTYPE__` zur Unterscheidung der Ausführungsplattform und der Funktion `getsymbol`, die innerhalb der `init`-Methode die Zuweisung der plattformspezifischen Standardfunktionen an statische Variablen der `FileSystem`-Klasse vornimmt.

Das Modul kann mit

```
bp2 -c filesys -o filesys.bpm (vgl. 2.4)
```

in ein Bytecode-Modul übersetzt und anschließend mit

```
#use "filesys.bpm" (vgl. 4.5.2)
```

in ein Anwendungsprogramm aufgenommen werden.

15.2 Erweitern von B++ mit dynamischen Bibliotheken

Wenngleich es mit den von B++ bereitgestellten Mitteln grundsätzlich möglich sein sollte, nahezu jede benötigte Funktionalität selbst zu implementieren, kann es wünschenswert sein, fest programmierte Erweiterungen bereitzustellen. Dies ist insbesondere dann in Erwägung zu ziehen, wenn zeitkritische Operationen auszuführen oder direkte Zugriffe auf Betriebssystem und/oder Hardware notwendig sind, die allein mit den Mitteln von B++ nur schwer realisiert werden können.

Derartige Erweiterungen werden normalerweise als dynamische Bibliotheken (DLLs) implementiert⁶⁸, wobei zwischen *allgemeinen Funktionsbibliotheken* und *B++-spezifischen Bibliotheken* zu unterscheiden ist.

15.2.1 Allgemeine Funktionsbibliotheken

Eine allgemeine Funktionsbibliothek stellt einen Satz von Funktionen oder einfachen Datenstrukturen (keine Klassen oder Objekte) bereit. Eine solche Bibliothek ist von B++ weitgehend unabhängig und kann prinzipiell in jeder denkbaren Programmiersprache und mit jedem Compiler, der dynamische Bibliotheken erzeugen kann, implementiert werden.

Die Verwendung solcher Bibliotheken aus B++ erfolgt mit Hilfe der vordefinierten Funktionen `LoadLibrary` (Laden einer Bibliothek), `FreeLibrary` (Entladen einer Bibliothek), `GetProcAddress` (Funktionszeiger holen) und `CallLibFunc` (Funktion aufrufen) – vgl. Abschnitt 11.7.

Die Namen der aus der Bibliothek exportierten Funktionen müssen der C-Konvention entsprechen. Der Aufruf der Funktionen erfolgt unter Windows nach `STDCALL`-, sonst ebenfalls nach C-Konvention.

Alle exportierten Funktionen müssen darüber hinaus einen Rückgabewert liefern, der zum `intptr`-Typ von B++ kompatibel ist und dürfen ausschließlich Argumente erwarten, die ebenfalls dem `intptr`-Typ entsprechen. Konkret heißt dies, dass Rückgabewerte und Argumente Zeiger oder numerische Werte sein müssen, deren Breite in Bit der eines Zeigers entspricht – also 32 Bit auf DOS- und Windows32-Plattformen bzw. 64 Bit auf Windows64-Plattformen.

Hinweise:

- Die Forderung für den Rückgabewert vom `intptr`-Typ erfüllt implizit jeder Rückgabewert von einem Zeigertyp oder einem ganzzahligen Typ, dessen Größe (in Bits) die eines Zeigers nicht überschreitet. Die Umwandlung in den `intptr`-Wert erfolgt automatisch. Somit können z. B. auch sämtliche Windows-API-Funktionen problemlos aufgerufen und damit Windows-API-DLLs als ‚allgemeine Funktionsbibliotheken‘ aufgefasst werden.
- Da es unter MS-DOS (und kompatiblen Systemen) keine standardmäßige Unterstützung für dynamische Bibliotheken gibt, sind für diese Systeme einige Besonderheiten und Einschränkungen zu beachten. Sie werden ausführlich in Abschnitt 15.2.3 behandelt.

15.2.2 B++-spezifische Bibliotheken

B++-spezifische Bibliotheken erlauben die direkte Erweiterung von B++ durch benutzerdefinierte Funktionen und Klassen. Sie können dazu direkt auf die B++-VM mit allen zugehörigen Datenstrukturen und Funktionen zugreifen.

⁶⁸ Grundsätzlich ist es natürlich auch möglich, zusätzliche Standardfunktionen (oder auch Klassen) direkt den Quellen von B++ hinzuzufügen und das ganze System neu zu übersetzen.

B++-spezifische Bibliotheken benutzen B++-Objekttypen und müssen deshalb in C++ implementiert und auf derselben Plattform wie B++ selbst übersetzt werden. Zudem werden sie dynamisch gegen die B++-Basisbibliothek `bpplib` gelinkt⁶⁹.

Im Gegensatz zu den allgemeinen Funktionsbibliotheken können die B++-spezifischen Bibliotheken Funktionen und Klassen bereitstellen, die in B++ direkt (also ohne den Umweg über `LoadLibrary` / `GetProcAddress` / `CallLibFunc`) verwendbar sind und keinen Einschränkungen hinsichtlich der verwendbaren Datentypen unterliegen.

Eine B++-spezifische Bibliothek wird mit der Verarbeitungsanweisung `#import` (vgl. 4.5.3) in ein Programm eingebunden und muss eine exportierte Funktion

```
void RegisterNativeModule(BppVirtualMachine* vm)
```

zur Registrierung der Bibliothek in der B++-VM bereitstellen. Diese Funktion wird beim Laden der Bibliothek automatisch aufgerufen.

Beispiel 86 zeigt die ‚typische‘ Deklaration der Funktion `RegisterNativeModule` anhand der Bibliothek `bp2class`.

```
#ifndef BP2CLASS_H
#define BP2CLASS_H

/** @addtogroup Bp2ClassLib
@{
*/

#ifdef WIN32
    #ifdef BP2CLASS_EXPORTS
        #define BP2CLASS_DLLEXP __declspec(dllexport)
    #else
        #define BP2CLASS_DLLEXP __declspec(dllimport)
    #endif
#else
    #define BP2CLASS_DLLEXP
#endif

#include <bppdef.h>

class BppVirtualMachine;

extern "C" {
    /** Module registration function.
    This function is called when library is loaded by virtual machine.
    It creates and registers classes implemented in library.
    @param vm pointer to virtual machine
    @see BppVirtualMachine::loadNativeModule
    */
    BP2CLASS_DLLEXP void BPPAPI RegisterNativeModule(BppVirtualMachine* vm);
}
/// @}

#endif // BP2CLASS_H
```

Beispiel 86 Header mit Registrierungsfunktion der Bibliothek `bp2class`

Für Windows-Plattformen wird dabei zunächst ein Makro `BP2CLASS_DLLEXP` mit den Spezifikationen für den DLL-Export definiert. Ein weiteres Makro (`BPPAPI`) entstammt dem Header `bppdef.h` und legt für Windows-Plattformen die Aufrufkonvention `STDCALL` fest.

Die Registrierungsfunktion selbst ist als C-Funktion deklariert. Ihre Implementierung hat die Aufgabe, bei der im Argument `vm` übergebenen B++-VM die von der Bibliothek bereitgestellten

⁶⁹ Weil dies unter MS-DOS nicht möglich ist, lassen sich auf dieser Plattform keine B++-spezifischen Bibliotheken erstellen.

Symbole (Funktionen und Klassen) zu registrieren, wofür in der Klasse `BppVirtualMachine` die Methoden `registerNativeFunction` bzw. `registerNativeClass` verwendet werden (vgl. Referenzdokumentation).

15.2.2.1 Funktionen in Bibliotheken

Die Implementierung von Funktionen in B++-Erweiterungsbibliotheken entspricht genau der der von B++ bereitgestellten Standardfunktionen.

Der Prototyp dieser Funktionen ist im Header `bppdef.h` als

```
typedef bool (*BPPUSERFUNC) (BppVirtualMachine* vm, int argc, ValPtr& sp)
definiert.
```

Dabei haben die Argumente folgende Bedeutung:

- `vm`: Zeiger auf die B++-VM, in deren Kontext die Funktion aufgerufen wird
- `argc`: Anzahl der auf dem Stack übergebenen Argumente
- `sp`: Stack-Pointer (Verweis auf das oberste Element des Stacks)

Der Typ `ValPtr` ist ein wahlfreier Iterator auf Vektoren (Klasse `BppVector`) und im Header `bppvec.h` definiert – siehe hierzu auch die Referenzdokumentation.

Als Rückgabewert liefert die Funktion `true` bei Erfolg. Im Fehlerfall muss sie eine der `returnError`-Methoden des B++-Interpreters aufrufen und `false` als Ergebnis liefern. Dadurch wird im B++-Programm eine Ausnahme ausgelöst.

Die Rückgabe des Funktionsergebnisses erfolgt auf dem Stack als dessen oberstes Element, also an der Position von `sp` nach der Rückkehr aus dem Aufruf.

Zur vereinfachten Implementierung von Funktionen sind im Header `bppfimpl.h` folgende Makros definiert:

- `CHKVM`: prüft das Argument `vm` und definiert eine lokale Variable `BppInterpreter* interpreter` (Zeiger auf den B++-Interpreter)
- `PR`: liefert einen Verweis auf die Ausgabeschnittstelle (Implementierung von `IBppPrinter`) der VM
- `THROW`: verlässt die Funktion mit einem Aufruf von `interpreter->returnError` und der angegebenen Argumentliste
- `CHKTYPE (VAL, TYPE)`: führt eine Typüberprüfung für einen Wert durch
- `ARGCMIN (MINCNT)`: prüft auf eine minimale Anzahl übergebener Argumente
- `ARGCMAX (MAXCNT)`: prüft auf eine maximale Anzahl übergebener Argumente
- `CHKARGC (CMIN, CMAX)`: prüft, ob die Anzahl übergebener Argumente in einem zulässigen Bereich liegt

Beispiel 87 illustriert die Implementierung anhand einer Funktion `copyfile`, die ihrem Zweck nach der Methode `FileSys::copyfile` aus Beispiel 85 ähnelt. Sie erwartet als Argumente Dateinamen bzw. -pfade der Quell- und der Zieldatei sowie ein optionales Flag, das angibt, ob eine bereits vorhandene Zieldatei ggf. überschrieben werden soll.

```

1 #include <bppvm.h>
2 #include <bppint.h>
3 #include <bppvalue.h>
4 #include <bppvec.h>
5 #include <bppstrng.h>
6 #include <bppprint.h>
7 #include <bppfimpl.h>
8 #include <windows.h>
9
10 // bool copyfile(string src, string dst, bool overwrite = true);
11 bool fn_copyfile(BppVirtualMachine* vm, int argc, ValPtr& sp)
12 {
13     CHKVM;          // defines interpreter variablen
14     CHKARGC(2,3); // two or three arguments allowed
15     ValPtr bp = sp; bp-=argc; // bp points before first argument
16     BppValue vsrc;
17     if (!interpreter->doStrCast(vsrc,bp[1])) return false;
18     BppValue vdst;
19     if (!interpreter->doStrCast(vdst,bp[2])) return false;
20     bool overwrite = true;
21     if (argc == 3) {
22         BppValue v;
23         if (!interpreter->doNumCast(v,bp[3],BVT_INT,_T("bool"))) return false;
24         overwrite = ((BppInt)v) != 0;
25     }
26     BppString* src = vsrc;
27     BppString* dst = vdst;
28     // call CopyFile on Windows-API
29     BOOL res = CopyFile(src->c_str(),dst->c_str(),overwrite ? FALSE : TRUE);
30     *bp = res ? 1 : 0;
31     sp = bp;
32     return true;
33 }
34
35 // Register function at VM
36 void RegisterNativeModule(BppVirtualMachine* vm)
37 {
38     vm->registerNativeFunction(_T("copyfile"),fn_copyfile);
39 }

```

Beispiel 87 Implementierung einer B++-Erweiterungsfunktion

Am Beginn des Funktionsrumpfes (Zeilen 13 und 14) wird mit Hilfe von Makros aus `bppfimpl.h` zunächst der interne `interpreter`-Verweis erzeugt und eine Überprüfung der zulässigen Parameterzahl vorgenommen.

Für den Zugriff auf die Funktionsargumente wird in Zeile 15 ein Verweis `bp` (für Base Pointer) angelegt, der vor das erste Argument auf dem Stack der VM zeigt (vgl. Tabelle 27). Entsprechend referenzieren die Zugriffe über `bp[1]`, `bp[2]` und `bp[3]` die Argumente `src`, `dst` bzw. `overwrite`. Dabei muss die Unterstützung optionaler Argumente mit ihren Vorgabewerten ‚per Hand‘ implementiert werden (Zeilen 20 bis 25). Eine implizite Unterstützung durch Compiler und Interpreter, wie das bei Bytecode-Funktionen der Fall ist, gibt es nicht.

Anders als die meisten Standardfunktionen, führt `copyfile` implizite Typkonvertierungen der Argumente durch (Zeilen 17, 19 und 23). Auch dies muss explizit implementiert werden. Im Beispiel werden zur Konvertierung die vom Interpreter bereitgestellten Methoden `doStrCast` und `doNumCast` verwendet. Ist eine Konvertierung nicht möglich, hinterlegen diese Methoden eine entsprechende Fehlermeldung im Interpreter. Das Verlassen der Funktion mit `false` als Ergebnis führt im aufrufenden Programm zu einer Ausnahme vom Typ `string`, wobei die Fehlermeldung als Wert übergeben wird.

In Zeile 29 wird durch den Aufruf der Windows-API-Funktion `CopyFile` die eigentliche Kopieroperation durchgeführt. Das Funktionsergebnis wird im Stack der VM an der Stelle, auf die `bp` verweist, abgelegt. Zum Schluss erfolgt eine Korrektur des Stack-Pointers (Zeile 31), so dass er ebenfalls auf das Funktionsergebnis verweist.

Zeilen 36 bis 39 zeigen eine Minimalimplementierung der `RegisterNativeModule`-Funktion der Bibliothek, in der hier nur ein einziger Aufruf der `registerNativeFunction`-Methode der VM steht. Eine ‚echte‘ Bibliothek würde an dieser Stelle typischerweise mehrere Funktionen (und/oder Klassen) registrieren.

Hinweise:

- Die Verwendung des `bp`-Verweises ist nicht zwingend erforderlich, sondern eher eine Konvention für den Zugriff auf die Argumente.
- Durch die Zuweisung des Stack-Pointers in Zeile 31 wird gewährleistet, dass `sp` nach der Rückkehr aus der Funktion auf das Funktionsergebnis zeigt. Dabei werden die Referenzen auf die Argumentwerte `src` und `dst` zunächst nicht freigegeben – dies erfolgt erst beim Überschreiben der Stack-Einträge im weiteren Programm. Wollte man eine explizite Freigabe (und damit ein vollständig determiniertes Verhalten) erzwingen, wäre dies durch zusätzliche Aufrufe `bp[1].dispose()` bzw. `bp[2].dispose()` möglich.
- Es ist auch möglich, auf die Korrektur von `sp` zu verzichten und das Ergebnis direkt an das Stack-Element, auf das `sp` zeigt, zuzuweisen. Die VM (bzw. der Interpreter) führt in diesem Fall nach dem Aufruf eine automatische Korrektur durch und ruft dabei implizit auch `dispose` auf freigegebenen Stack-Elementen auf. Der ‚Preis‘ dafür ist ein geringfügig langsames Verhalten zur Laufzeit.

15.2.2.2 Klassen in Bibliotheken

Neben Funktionen können in B++-Bibliotheken auch Klassen bereitstellen. Typische Anwendungsfälle hierfür sind:

- Klassen, die objektorientierte Datentypen (z. B. Container) für B++ bereitstellen,
- Wrapper-Klassen für systemeigene Datenstrukturen oder vorhandene C++-Klassen sowie
- Klassen, die eine objektorientierte Schnittstelle zum API des Betriebssystems bereitstellen.

Beispiele für all diese Anwendungsfälle finden sich in der Erweiterungsbibliothek `bp2class`, die Bestandteil der B++-Distribution ist. Auszüge aus dieser Bibliothek werden nachfolgend auch zur Veranschaulichung der Implementierung von ‚nativen‘ Klassen verwendet.

Wie in Abschnitt 14.2 dargestellt, besitzt eine B++-Klasse zwei `dictionary`-Objekte als Symboltabellen (`_dataMembers` und `_funcMembers`), die Definitionen der Daten und Methoden der jeweiligen Klasse aufnehmen. Für eine ‚native‘-Implementierung einer Klasse müssen im Wesentlichen diese beiden Symboltabellen „gefüllt“ werden. Hierfür stellt die interne Repräsentation von B++-Klassen (Klasse `BppClass`) eine Reihe von Hilfsfunktionen zur Verfügung – siehe Tabelle 28.

Methode	Beschreibung
<code>bool addDataMember(const BppChar* name)</code>	Hinzufügen einer Elementvariablen
<code>bool addDataMember(const BppChar* name, const BppValue& initValue)</code>	Hinzufügen einer Elementvariablen mit Initialwert
<code>bool addStaticDataMember(const BppChar* name)</code>	Hinzufügen einer Klassenvariablen
<code>bool addStaticDataMember(const BppChar* name, const BppValue& initValue)</code>	Hinzufügen einer Klassenvariablen mit Initialwert
<code>bool addMethod(const BppChar* name, BPPUSERFUNC pfunc, unsigned char flags = 0)</code>	Hinzufügen einer Elementfunktion (Methode)
<code>bool addStaticMethod(const BppChar* name, BPPUSERFUNC pfunc, unsigned char flags = 0)</code>	Hinzufügen einer Klassenfunktion

Tabelle 28 Hilfsmethoden zur Definition von Klassenelementen

Dabei erlauben die Methoden zur Datendefinition mit Initialwert neben der Angabe des eigentlichen Wertes mit Hilfe von Flags (siehe `BppValue::setFlags`, `BppValue::setProtectionLevel` und `enum BppValueFlags` in der Referenzdokumentation) auch die Festlegung von Zugriffsspezifizierern und/oder statischer Typisierung. Bei den Methodendefinitionen ist dies über das optionale Argument `flags` möglich, wobei hier nur die Festlegung des Zugriffsschutzes (`BVF_PRIVATE`, `BVF_PROTECTED` oder `BVF_PUBLIC`) sinnvoll ist.

```
class String
{
public:
    /// default ctor - creates empty string
    String();
    /// Creates string as a copy of another one.
    String(var assignment);
    /// Creates string from format specification.
    String(var formatstr, var arg0);
    /// Destructor releases internal string object.
    ~String();
    /// Left side concatenation operator implementation.
    String OP_ADD(var suffix);
    /// Right side concatenation operator implementation.
    String OP_ADD_R(var prefix);
    /// Access operator (read).
    int OP_VREF(int index);
    /// Access operator (write).
    null OP_VSET(int index, int charval);
    /// Compare function (case sensitive).
    int compare(var comparison);
    /// Compare function (case insensitive).
    int icompare(var comparison);
    /// Gets length of current string.
    int length();
    /// Gets C-string from current object.
    intptr c_str();
    /// Gets character buffer containing string contents.
    charbuffer c_buf();
    /// Gets byte buffer containing string contents (single byte).
    buffer a_buf();
    /// Gets byte buffer containing string contents (wide character).
    buffer w_buf();
    /// Clears all contents from string.
    null clear();
    /// Finds first position of substring or character.
    int find(var search, start = 0);
    /// Replaces part of string.
```



```

String replace(var search,var replacement);
/// Gets substring from current string.
String substr(int start,int len = -1);
/// Erases part of string.
String erase(int pos, int cnt = -1);
/// Inserts specified string into current one.
String insert(var str, int pos);
/// Inserts character(s) into current string.
String insert(int chr, int pos, int cnt = 1);
/// Appends string or single character to current string.
String append(var suffix);
/// Converts current object into a string value.
operator string();
protected:
    /// string value
    string _str;
}

```

Beispiel 88 Klasse String (B++-Pseudodeklaration)

Beispiel 88 zeigt die Pseudodeklaration der Klasse `String` aus der Bibliothek `bp2class`, die ein Zeichenkettenobjekt bereitstellt. In Beispiel 89 sind Auszüge aus ihrer Implementierung (Konstruktoren, linksseitiger Verkettungsoperator sowie die Funktion zum Aufbau der Klasse in der B++-VM) angegeben. Sie können als Vorlage für eine ‚typische‘ Implementierung von B++-Erweiterungsklassen angesehen werden.

```

static unsigned _strIndex = 0;

bool Bp2String_ctor(BppVirtualMachine* vm, int argc, ValPtr& sp)
{
    CHKVM;
    CHKOBJECT;
    BppValue& strVal = MEMBER(_strIndex);
    if (argc == 1)
    {
        strVal = new BppString();
        return true;
    }
    BppValue savedSpVal = *sp; // save TOS value before function call
    BppString* s = interpreter->getStringFromValueExtern(bp[2]);
    if (s == 0)
        return interpreter->badType(bp[2].getType(),BVT_STRING);
    if (argc == 2)
    {
        strVal = new BppString(*s);
    }
    else
    {
        // helper to prevent s from release
        BppValue savedStrVal = *sp;
        *sp = savedSpVal; // restore saved TOS value (last argument)
        ValPtr pv = bp;
        pv += 3;
        strVal = new BppString(s->c_str(),argc-2,pv);
    }
    *bp = pThis;
    sp = bp;
    return true;
}

bool Bp2String_OP_ADD(BppVirtualMachine* vm, int argc, ValPtr& sp)
{
    CHKVM;

```

```

CHKOBJECT;
CHKARGC(2,2);
BppString* s = interpreter->getStringFromValueExtern(bp[2]);
if (s == 0)
{
    if (!bp[2].isIntegralType())
        return interpreter->badType(bp[2].getType(), BVT_STRING);
}
BppValue& strVal = MEMBER(_strIndex);
BppString* str = strVal;
if (str == 0)
    return interpreter->badType(strVal.getType(), BVT_STRING);

BppObject* result = new BppObject(pThis->getClass());
BppValue& resValue = (*(result->getMemberData()))[_strIndex];

BppString* resString = new BppString(*str);
resValue = resString;
if ((s != 0))
    (*resString)+= s->c_str();
else
    (*resString)+= (BppChar)(BppInt)bp[2];
*bp = result;
sp = bp;
return true;
}

/* Function is called from RegisterNativeModule function.
It initializes the String class and registers it in virtual machine vm. */

void Bp2String_CreateClass(BppVirtualMachine& vm)
{
    BppClass* pClass = new BppClass(vm);
    if (!vm.registerNativeClass(_T("String"), pClass)) { delete pClass; return; }
    pClass->addDataMember(_T("_str"));
    pClass->addMethod(_T("String"), &Bp2String_ctor);
    pClass->addMethod(_T("OP_ADD"), &Bp2String_OP_ADD);
    pClass->addMethod(_T("OP_ADD_R"), &Bp2String_OP_ADD_R);
    pClass->addMethod(_T("OP_VREF"), &Bp2String_OP_VREF);
    pClass->addMethod(_T("OP_VSET"), &Bp2String_OP_VSET);
    pClass->addMethod(_T("compare"), &Bp2String_compare);
    pClass->addMethod(_T("icompare"), &Bp2String_icompare);
    pClass->addMethod(_T("length"), &Bp2String_length);
    pClass->addMethod(_T("c_str"), &Bp2String_c_str);
    pClass->addMethod(_T("c_buf"), &Bp2String_c_buf);
    pClass->addMethod(_T("a_buf"), &Bp2String_a_buf);
    pClass->addMethod(_T("w_buf"), &Bp2String_w_buf);
    pClass->addMethod(_T("clear"), &Bp2String_clear);
    pClass->addMethod(_T("substr"), &Bp2String_substr);
    pClass->addMethod(_T("find"), &Bp2String_find);
    pClass->addMethod(_T("replace"), &Bp2String_replace);
    pClass->addMethod(_T("insert"), &Bp2String_insert);
    pClass->addMethod(_T("append"), &Bp2String_append);
    pClass->addMethod(_T("toString"), &Bp2String_toString);
    pClass->addMethod(_T("string"), &Bp2String_toString);
    pClass->fixupDataDefs();
    _strIndex = pClass->getMemberIndex(_T("_str"));
}

```

Beispiel 89 Implementierung der Klasse String (Auszug)

Der Prototyp der Implementierungsfunktionen für Methoden von Klassen entspricht dem für ‚einfache‘ Funktionen (siehe Abschnitt 15.2.2.1). Zur Implementierung von Methoden sind im Header `bpplimpl.h` folgende weiteren Makros definiert:

- `CHKOBJECT` : prüft die Gültigkeit des Objektverweises und definiert die lokalen Variablen `ValPtr bp` (Stackzeiger vor den Objektverweis) und `BppObject* pThis` (Objektverweis),
- `ARG(I)` : liefert eine Referenz auf das `I`-te Argument; `ARG(0)` ist der Objektverweis,
- `MEMBER(IDX)` : liefert eine Referenz auf die Member-Variable (`BppValue&`) mit dem Index `IDX`,
- `CLMEMBER(NAME)` : liefert den Symboltabelleneintrag für das Klassenelement mit dem Namen `NAME` (`BppDictionaryEntry*`), wobei Basisklassen mit einbezogen werden,
- `OBJMEMBER(pOBJ, IDX)` : liefert eine Referenz auf die Member-Variable mit dem Index `IDX` im durch `pOBJ` angegebenen Objekt.

Die Funktion `Bp2String_CreateClass` in Beispiel 89 übernimmt den Zusammenbau der Klasse und wird von der Registrierungsfunktion der Bibliothek (vgl. Beispiel 86) aufgerufen. Hier wird zunächst eine neue Klasse angelegt und mit Hilfe der Methode `registerNativeClass` in der Virtuellen Maschine registriert. Anschließend werden der Klasse Datenelemente und Funktionen hinzugefügt. Der Aufruf von `fixupDataDefs()` auf der Klasse ordnet deren Datenelementen Indizes zu, über die sie im B++-Bytecode (und auch in den Methodenimplementierungen innerhalb der Bibliothek) adressiert werden können. Hierfür wird im Beispiel anschließend der Index des Datenelements `_str` bestimmt und der Variablen `_strIndex` zugewiesen.

15.2.3 Dynamische Bibliotheken für MS-DOS

B++ unterstützt auch für DOS-Plattformen ein Konzept dynamischer Bibliotheken (DLLs), das dem von Windows ähnelt und in der Dokumentation der DOS-DLL-Bibliothek [4] näher beschrieben wird. Solche Bibliotheken können beliebige zusätzliche Funktionen bereitstellen und prinzipiell in beliebigen Programmiersprachen geschrieben sein⁷⁰, sofern diese Programme im EXE-Format erzeugen können und Zeiger auf Funktionen ermöglichen.

Eine solche DLL wird zunächst wie ein ‚ganz normales‘ Programm geschrieben, das mindestens die Funktionen, die exportiert werden sollen, als globale FAR-Funktionen definiert. Darüber hinaus werden folgende Erweiterungen benötigt:

- eine Exporttabelle
- eine Registrierungsstruktur
- eine spezielle Einsprungfunktion

Für das Erstellen von DLLs in (Turbo-/Borland-) C/C++ definiert der DOS-DLL-Header `dosdll.h` Makros zur Implementierung dieser Erweiterungen. In anderen Sprachen müssen die entsprechenden Strukturen „per Hand“ aufgebaut werden.

Wie in [4] beschrieben, müssen DOS-DLLs mit dem Speichermodell LARGE übersetzt werden.

⁷⁰ Besonders geeignet sind C/C++ sowie Assembler, weshalb sie hier im Mittelpunkt stehen.

Im Folgenden wird die Implementierung einer DLL für die DOS-Plattform in C++ und Assembler⁷¹ demonstriert. Die Beispielbibliothek soll den Namen `SYSTEM.EXE`⁷² tragen und folgende Funktionen bereitstellen⁷³:

- `long execProgram(const char* cmd, const char* cmdArgs);`
Ausführen eines externen Programms
- `long getEnv(const char* name, long size, char* buf);`
Bestimmen des Wertes einer Umgebungsvariablen
- `long searchPath(const char* name, long size, char* buf);`
Suchen einer Datei im durch die Umgebungsvariable `PATH` angegebenen Pfad
- `long GetLastError(long size, char* errBuf);`
Ermitteln der Beschreibung des zuletzt aufgetretenen Fehlers in Textform.

Die Wahl der Signaturen der exportierten Funktionen ist dabei nicht beliebig. B++ stellt mit der Standardfunktion `CallLibFunc` (Abschnitt 11.7) einen allgemeinen Mechanismus zum Aufruf von DLL-Funktionen bereit, der naturgemäß keine Kenntnis über die konkrete Funktion hat. Deshalb wird davon ausgegangen, dass DLL-Funktionen ausschließlich Argumente von vom Typ `intptr` (mit 32 Bit Breite) übernehmen und als Ergebnis stets einen `intptr` zurückliefern. Demnach gilt:

- Argumente sind 32 Bit breite Ganz- bzw. Realzahlen (`float`) oder FAR-Zeiger.
- Der Rückgabewert einer DLL-Funktion ist immer eine 32 Bit breite Ganzzahl (die natürlich auch die Bedeutung eines `float`-Wertes oder eines Zeigers haben kann).

In der Praxis bedeutet dies lediglich die Einschränkung, dass keine Werte mit 64 Bit Breite (`long`, `ulong`, `double`) übergeben werden können. Alle anderen die Werttypen von B++ werden beim Aufruf implizit nach `intptr` umgewandelt. Die Adresse des Datenbereichs von Referenztypen kann mit Hilfe der Standardfunktion `addr` bestimmt werden.

15.2.3.1 Implementierung in C/C++

Die Implementierung in C/C++ nutzt die durch den Header `dosdll.h` aus der DOS-DLL Bibliothek bereitgestellten Makros sowie die C-Standardbibliothek⁷⁴. Dadurch fällt der Quelltext vergleichsweise kurz aus. Er ist nachfolgend (Beispiel 90) vollständig angegeben und wird im Anschluss diskutiert.

```
#define _NOFLOAT_

#include <string.h>
#include <stdlib.h>
#include <process.h>
#include <stdio.h>
#include <dir.h>
#include <dos.h>
#include "dosdll.h"

long huge GetLastError(long size, char* errBuf)
{
    strncpy(errBuf, strerror(errno), size);
    return 0;
}
```

⁷¹ Hier wird der Assembler A86 (Version 3.22) verwendet.

⁷² Die Erweiterung kann im Prinzip beliebig gewählt werden, naheliegender wäre möglicherweise `.DLL`. Die Turbo-/Borland C/C++-Compiler nennen jedoch automatisch alle ausführbaren Dateien `.EXE`, so dass es hier der Einfachheit halber dabei geblieben ist.

⁷³ Die Signaturen sind in C++-Syntax angegeben.

⁷⁴ Einschränkungen hinsichtlich der Nutzung der Standardbibliotheken sind in der Dokumentation zur DOS-DLL beschrieben.

```

long huge execProgram(const char* cmd, const char* cmdArgs)
{
    return (long) Spawn((char*)cmd, (char*)cmdArgs);
}

long huge getEnv(const char* name, long size, char* buf)
{
    long result = 0;
    const char* pvar = getenv(name);
    if (pvar != 0)
    {
        result = strlen(pvar);
        if (buf != 0)
            strncpy(buf, pvar, size);
    }
    return result;
}

long huge searchPath(const char* name, long size, char* buf)
{
    long result = 0;
    const char* pvar = searchpath(name);
    if (pvar != 0)
    {
        result = strlen(pvar);
        if (buf != 0)
            strncpy(buf, pvar, size);
    }
    return result;
}

BEGIN_EXPTABLE
    EXPENTRY(execProgram),
    EXPENTRY(getLastError),
    EXPENTRY(getEnv),
    EXPENTRY(searchPath),
END_EXPTABLE

IMPLEMENT_DEFAULT_ENTRYFUNC_ENV

REGISTERENTRYFUNC

#ifdef __TURBOC__ >= 0x0300
    static char dummybuf[255];
#endif

int main(int, char**)
{
    puts("Can't run standalone.\n");
    return 1;
}

```

Beispiel 90 Implementierung einer DLL-Funktionsbibliothek mit Turbo-C++

Die `_NOFLOAT_`-Definition am Anfang des Quelltextes verspricht, dass keinerlei Unterstützung für Gleitpunkt-Arithmetik benötigt wird (und macht den Umfang des Binärcodes um knapp 20 KB kleiner). Fehlt diese Definition, so muss die Übersetzung selbst mit eingeschalteter 8087-Emulation erfolgen.

Mit den `#include`-Anweisungen werden die Deklarationen der verwendeten Bibliotheksfunktionen sowie die Implementierungsmakros für die DLL-Schnittstelle (aus `dosdll.h`) importiert.

Die exportierten Funktionen selbst sind trivial und machen im Wesentlichen von vorhandenen Bibliotheksfunktionen Gebrauch. Ein paar Anmerkungen scheinen dennoch angebracht:

- Die `huge`-Modifizier in den Funktionsköpfen erzwingen FAR-Aufrufe der Funktionen auch dann, wenn die Bibliothek im COMPACT-Modell übersetzt wird⁷⁵.
- Die Funktionen `getLastError`, `getEnv` und `searchPath` legen ihr eigentliches Ergebnis (eine Zeichenkette) in einem Puffer ab, der vom aufrufenden Programm zur Verfügung gestellt werden muss. Der Funktionswert selbst ist von untergeordneter Bedeutung und wird lediglich als Erfolgsmeldung verwendet.
- Die Funktion `execProgram` ruft eine Funktion `spawn` auf, die von B++ (bzw. DOS-DLL) bereitgestellt und deren Adresse der DLL-Eintrittsfunktion übergeben wird. Der Grund dafür ist, dass Bibliotheksfunktionen, die dynamische Speicheranforderungen durchführen, innerhalb einer DLL nicht direkt aufgerufen werden dürfen – siehe hierzu [4].

Auf die Implementierung der eigentlichen Funktionen folgt die der DLL-Schnittstelle mit Hilfe von Makros, die **genau in der hier angegebenen Reihenfolge** verwendet werden müssen:

- Zunächst wird mit die Exporttabelle mit Hilfe von `BEGIN_EXPTABLE` (Kopf) und `END_EXPTABLE` (Fuß) aufgebaut. Die einzelnen Tabelleneinträge entsprechen den exportierten Funktionen und werden mit Hilfe des Makros `EXPENTRY(<funcname>)` definiert. Alternativ kann auch das – ebenfalls in `dosdll.h` definierte – Makro `EXPENTRY2(<ext_name>, <funcname>)` verwendet werden, das den Export einer Funktion unter einem anderen Namen erlaubt.
- Das Makro `IMPLEMENT_DEFAULT_ENTRYFUNC_ENV` implementiert die Eintrittsfunktion der DLL. Alternativ dazu existiert das Makro `IMPLEMENT_DEFAULT_ENTRYFUNC`. Der Unterschied zwischen den beiden Varianten besteht darin, dass die mit dem Suffix `_ENV` zusätzlich die Umgebungsvariablen des Hauptprogramms kopiert, was (wie in diesem Fall notwendig) die Verwendung von Bibliotheksfunktionen, die auf Umgebungsvariablen zugreifen, ermöglicht.
- Mit dem Makro `REGISTERENTRYFUNC` schließlich wird die Registrierungsstruktur der DLL erzeugt und die Adressen der Einsprungfunktion und der Exporttabelle darin eingetragen.

Was nun folgt, ist eine Verbeugung vor den 3.x-Versionen von Borland C++. Es wird ein statischer Datenbereich angelegt, der notwendig ist, um die Funktion der Standard-I/O-Funktionen der C-Bibliothek in der DLL zu gewährleisten⁷⁶. Für die (frei verfügbare) Turbo-C++-Version 1.01 ist dies nicht erforderlich und wird deshalb auch nicht mit übersetzt.

Die `main`-Funktion gibt es vor allem deshalb, weil der Compiler sie für eine ausführbare Datei verlangt. Hier wird sie einfach dazu verwendet, einen Anwender, der die DLL versehentlich direkt startet, auf seinen Irrtum hinzuweisen. Ein anderer sinnvoller Einsatzzweck könnte es sein, sie zum Test der exportierten Funktionen zu benutzen.

15.2.3.2 Implementierung in Assembler

Die Assembler-Variante der DLL kann weder die C-Standardbibliothek noch vordefinierte Makros zur Implementierung der DLL-Schnittstelle benutzen – hier ist Handarbeit notwendig. Dementsprechend ist der Quellcode der Assembler-Version auch wesentlich länger⁷⁷ – dafür wird man aber durch eine wesentlich kleinere DLL entschädigt.

⁷⁵ Es würde hier eigentlich auch der Modifizier `far` genügen, `huge` entspricht aber der Deklaration des Funktions-Prototypen in `dosdll.h`.

⁷⁶ Warum das so ist, wird dem Autor vermutlich für immer verborgen bleiben. Auch ist die notwendige Größe des Bereichs nicht gesichert – möglicherweise kann er auch kleiner sein.

⁷⁷ Deshalb werden hier auch nur einige Auszüge angegeben. Die vollständigen Quellen liegen der B++-Distribution bei.

Auch die Assembler-Version muss eine Datei im EXE-Format sein und mindestens ein Code- und ein Datensegment besitzen. Ein eigenes Stack-Segment ist dagegen nicht unbedingt erforderlich, da normalerweise der Stack des aufrufenden Programms mit benutzt wird.

Im Datensegment sind – wie nachfolgend gezeigt – mindestens

- die Namen der exportierten Funktionen,
- die Exporttabelle sowie
- die Registrierungsstruktur der Tabelle

zu definieren. Das Beispiel verwendet zusätzlich noch ein paar Zeichenketten zur Meldungsabgabe, einen Puffer für die Zwischenspeicherung von Dateipfaden sowie drei Zeiger auf Funktionen, die B++ bereitstellt und die in der Eintrittsfunktion (siehe weiter unten) initialisiert werden. Außerdem exportiert es gegenüber der C++-Version eine zusätzliche Funktion `searchEnv`, die sich ähnlich wie `searchPath` verhält, nur eben den Wert einer beliebigen Umgebungsvariablen als Suchpfad verwendet⁷⁸.

```
; data segment
DSEG SEGMENT 'DATA'
; some messages
msg          DB "Can't run standalone.",0DH,0AH,'$'
msg1         DB "Error messages not supported.",0DH,0AH,0
pathbuffer   DB 256 DUP (0)
pathvar      DB "PATH",0

;Pointers to C Functions exported by main program
pGetMem      DW 0, 0
pFreeMem     DW 0, 0
pSpawn       DW 0, 0

;names of exported functions
fname0       DB 'execProgram',0
fname1       DB 'getLastError',0
fname2       DB 'getEnv',0
fname3       DB 'searchPath',0
fname4       DB "searchEnv", 0

;export table
; each entry consists of far pointer to name followed by far pointer to function
;execProgram
funcEntry0   DW OFFSET fname0
             DW SEG DSEG
             DW execProgram
             DW SEG CSEG
;getLastError
funcEntry1   DW OFFSET fname1
             DW SEG DSEG
             DW getLastError
             DW SEG CSEG
;getEnv
funcEntry2   DW OFFSET fname2
             DW SEG DSEG
             DW getEnv
             DW SEG CSEG
;searchPath
funcEntry3   DW OFFSET fname3
             DW SEG DSEG
             DW searchPath
             DW SEG CSEG
```

⁷⁸ Genau genommen ist `searchPath` lediglich ein Spezialfall von `searchEnv`.

```

;searchPath
funcEntry4 DW OFFSET fname4
            DW SEG DSEG
            DW searchEnv
            DW SEG CSEG

;DllRegStruct - the registration structure
regTag      DB 'D_O_S_L_I_B0'    ; tag searched from main program
oEntryFunc  DW OFFSET entryFunc  ; offset of entry point function
sEntryFunc  DW SEG CSEG           ; segment of entry point function
oEntryArray DW funcEntry0         ;address of export table
sEntryArray DW SEG DSEG
entryCnt    DW 5                  ;number of entries in export table

DSEG ENDS

```

Beispiel 91 Datensegment einer DOS-DLL in Assembler

Die einzelnen Einträge der Exporttabelle bestehen jeweils aus einem FAR-Zeiger auf den exportierten Funktionsnamen, gefolgt von einem FAR-Zeiger auf den Anfang der jeweiligen Funktion.

Die Registrierungsstruktur beginnt immer mit einem festen Tag in Form der Zeichenfolge „D_O_S_L_I_B0“. Nach dieser Zeichenfolge sucht die DLL-Laderoutine, um die Adresse der Registrierungsstruktur zu bestimmen⁷⁹. Auf dieses Tag folgen unmittelbar die FAR-Zeiger auf die Einsprungfunktion und die Exporttabelle. Den Abschluss der Struktur bildet ein 16-Bit-Wort, das die Anzahl der Einträge in der Exporttabelle angibt.

Schreiben von Funktionen

Alle aus DLLs exportierten Funktionen sind FAR-Funktionen, die nach C-Konvention aufgerufen werden. Dies bedeutet:

- die Rücksprungadresse ist 32 Bit breit,
- die Funktionsargumente werden in umgekehrter Reihenfolge auf den Stack gelegt, also das letzte zuerst, und
- der Aufrufer ist für die Bereinigung des Stacks zuständig.

Zusätzlich gilt – wie schon erwähnt – die Festlegung, dass alle Argumente mit 32 Bit Breite übergeben werden. Demnach sieht der Stack unmittelbar nach dem Eintritt in eine Funktion wie folgt aus:

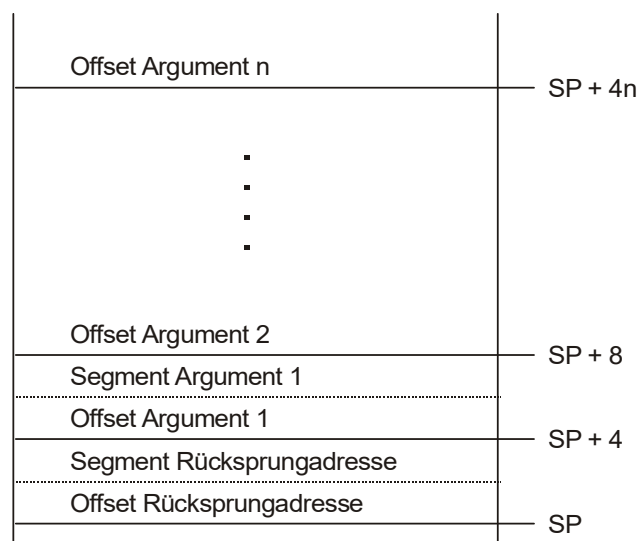


Abbildung 5 Aufbau des Stacks beim Eintritt in eine DOS-DLL-Funktion

⁷⁹ Zweifellos ist dies nicht besonders elegant. Eine schönere Lösung ist aber ohne zusätzliche Anforderungen an den internen Aufbau der DLL kaum denkbar.

Hinweise:

- Eine von einer DLL bereitgestellte Funktion sollte alle Register, die sie verwendet (mit Ausnahme der allgemeinen Register AX, BX, CX und DX), zwischenspeichern und am Ende wieder herstellen.
- Die Rückgabe des Funktionswertes erfolgt immer im Registerpaar DX:AX, wobei DX das höherwertige und AX das niederwertige Wort enthält.

Nachfolgend werden die Eintrittsfunktion der DLL (`entryFunc`) sowie die exportierte Funktion `execProgram` näher betrachtet. Diese Funktionen sind deshalb besonders interessant, weil `entryFunc` die Funktionszeiger auf die vom Hauptprogramm (also B++) bereitgestellten Funktionen initialisiert und `execProgram` eine dieser Funktionen benutzt.

Die Eintrittsfunktion:

Diese Funktion wird automatisch aufgerufen, wenn die DLL in den Prozess geladen wird⁸⁰. Beispiel 92 zeigt den Quelltext:

```
; Entry point function
; void __DllEntry(int mode, DLLGLOBSTRUCT* pGlobals)
entryFunc PROC FAR
    PUSH BP                ; save BP
    MOV BP, SP
    PUSH ES                ; save segment registers
    PUSH DS
    MOV AX, SS:[BP+6]       ; mode
    CMP AX, DllModeLoad    ; do nothing if mode not DllModeLoad
    JNE endEntryFunc
    MOV AX, DSEG            ; make DS and ES pointing to DSEG
    MOV ES, AX
    MOV DS, AX
    MOV DI, OFFSET pGetMem  ; ES:DI points to start of function pointer block
    LDS SI, SS:[BP+8]       ; DS:SI points to pGlobals
    MOV CX, 6              ; copy three function pointers (i.e. 6 words)
    CLD
    REP MOVSW
endEntryFunc:              ; restore segment registers
    POP DS
    POP ES
    POP BP                ; restore BP
    RET
entryFunc ENDP
```

Beispiel 92 Eintrittsfunktion einer DOS-DLL

Der Aufruf der Funktion erfolgt mit zwei Parametern. Der erste gibt den Aufrufmodus an, wobei hier nur der Modus `DllModeLoad` (d.h. `mode = 0`) behandelt wird. Der zweite ist ein Zeiger auf eine Struktur, die (in C++-Notation) wie folgt definiert ist:

⁸⁰ Genau genommen wird sie auch beim Entladen der DLL sowie bei jeder Änderung der Anzahl bestehender Referenzen aufgerufen. Dies spielt im Beispiel jedoch keine Rolle.

```

typedef void far* huge (*GetMemFunc)(unsigned long size);
typedef void huge (*FreeMemFunc)(void far* block);
typedef int huge (*SpawnFunc)(char far* cmd, char far* args);
struct DLLGLOBSTRUCT
{
    GetMemFunc huge GetMem;
    FreeMemFunc huge FreeMem;
    SpawnFunc huge Spawn;
    // more compiler specific variables may follow here
    // ...
};

```

An dieser Stelle ist wichtig, dass die Struktur mit Zeigern auf drei Funktionen beginnt, die B++ für DLLs bereitstellt, um ihnen die dynamische Anforderung und Freigabe von Speicher sowie das Ausführen externer Programme zu ermöglichen. Die Funktion `entryFunc` aus Beispiel 92 kopiert diese Zeiger in die Variablen `pGetMem`, `pFreeMem` und `pSpawn` im Datensegment. Damit können sie von den anderen Funktionen der DLL benutzt werden. Weil die drei Funktionszeiger im Datensegment in der gleichen Reihenfolge angeordnet sind, wie in der übergebenen Struktur, können sie einfach „im Block“ kopiert werden.

Die Funktion `execProgram`:

Die Implementierung der Funktion (Beispiel 93) ruft die von B++ bereitgestellte Funktion `Spawn` (über den Funktionszeiger `pSpawn` aus dem Datensegment) zurück.

```

; implementation of long execProgram(const char* cmd, const char* cmdArgs)
; we simply call Spawn function in main program
execProgram PROC FAR
    PUSH DS
    PUSH BP
    MOV BP, SP
    MOV AX, DSEG
    MOV DS, AX
    PUSH SS:[BP+14] ;push arguments to stack
    PUSH SS:[BP+12]
    PUSH SS:[BP+10]
    PUSH SS:[BP+8]
    CALL DWORD PTR DS:[pSpawn] ; call spawn, AX contains return value
    ADD SP, 8 ; remove arguments from stack
    POP BP
    POP DS
    CWD ; make dword result in DX:AX
    RET
execProgram ENDP

```

Beispiel 93 Funktionsimplementierung mit Rückruf nach B++ in Assembler

Dabei muss sie selbst für die Einhaltung der C-Konventionen beim Funktionsaufruf sorgen. Deshalb werden zunächst die Register `DS` und `BP` auf den Stack gerettet, dann die eigenen Funktionsargumente in umgekehrter Reihenfolge auf den Stack gelegt und anschließend die Rückruffunktion über den Zeiger `pSpawn` aufgerufen. Diese Funktion liefert im Register `AX` den Returncode des aufgerufenen Programms.

Nach dem Aufruf werden mit dem Befehl `ADD SP, 8` die Funktionsparameter wieder vom Stack geräumt, die Inhalte der Register `BP` und `DS` restauriert und schließlich mit `CWD` ein 32-Bit-Ergebnis im Registerpaar `DX:AX` erzeugt.

15.3 Verwenden dynamischer Bibliotheken

15.3.1 Allgemeine Funktionsbibliotheken

Der direkte Weg zur Verwendung von allgemeinen Funktionsbibliotheken (siehe 15.2.1) besteht in der Benutzung der von B++ bereitgestellten Funktionen `LoadLibrary`, `GetProcAddress`, `CallLibFunc` und `FreeLibrary` (vgl. Abschnitt 11.7). Nachfolgendes Beispiel zeigt diesen Weg, wobei eine Bibliothek „testdll.dll“ verwendet wird, die eine einfache Funktion „sayHello“ exportiert.

```
void main()
{
    print("main loads testdll.dll\n");
    // testdll will be handled directly
    // first load the library
    intptr hdl1=LoadLibrary("testdll.dll");
    if (!hdl1) {
        print("Loading 'testdll.dll' failed.\n");
        return;
    }
    // assign address of sayHello function
    intptr proc = GetProcAddress(hdl1,"sayHello");
    // call function and print its result
    if (proc) print(newstring("Result of sayHello: %d\n", CallLibFunc(proc)));
    else print("Function 'sayHello' not found in 'testdll.dll'.\n");
    // release library
    print("Release testdll.dll: ",FreeLibrary(hdl1)," \n");
}
```

Beispiel 94 Dynamisches Einbinden einer allgemeinen Funktionsbibliothek

Die einzelnen Schritte sind also:

- Laden der DLL
- Prüfen des Erfolgs
- Bestimmen der Adresse einer exportierten Funktion in der DLL
- Prüfen des Erfolgs
- Aufruf der Funktion
- Freigeben der DLL

Dies ist im Vergleich zum Aufruf eingebauter Funktionen ein recht aufwendiges Verfahren. Zudem ist insbesondere der Aufruf der `CallLibFunc`-Funktion nicht unkritisch, da hier die Übergabe korrekter Parameter allein in der Verantwortung des Programmierers liegt und Fehler an dieser Stelle leicht das gesamte Programm zum Absturz bringen.

Deshalb ist es oft vorteilhaft, die Benutzung solcher Funktionsbibliotheken auf der Seite von B++ in Klassen zu kapseln und so die Details für den Anwendungsprogrammierer zu verbergen. Darüber hinaus ergibt sich so auch die Möglichkeit statisch typisierter Aufrufe der Bibliotheksfunktionen.

In Beispiel 95 wird dies anhand einer B++-Klasse „System“ illustriert, die die Kapselung der in Abschnitt 15.2.3 vorgestellten Bibliothek übernimmt⁸¹:

```
/**
This class completely encapsulates using of SYSTEM.EXE library. This
includes loading and unloading of library as soon as memory management,
assignment of function pointers and calls to exported functions.
*/
class System
```

⁸¹ Die Klasse kann mit beiden Implementierungen der Bibliothek umgehen.

```

{
public:
    // method declarations
    System();           // ctor
    ~System();          // dtor
    int exec(string name, string cmdArgs=""); // execute child program
    string getEnv(string name); // get value of an environment variable
    string searchPath(string name); // search along PATH for a file name
    string getLastError(); // get description of error occurred at last
private:
    // member variables
    charbuffer strbuf; // buffer for returning string results
    intptr hdl1;       // handle of DLL
    intptr execproc;   // address of execProgram function in DLL
    intptr getenvproc; // address of getEnv function in DLL
    intptr searchproc; // address of searchPath function in DLL
    intptr geterrproc; // address of getLastError function in DLL
}

/**
Constructor loads library, assigns function pointers and creates the result
buffer.
*/
System::System()
{
    hdl1=LoadLibrary("system.exe");
    // We need not check hdl1 value since GetProcAddress returns null on
    // invalid handle.
    execproc = GetProcAddress(hdl1,"execProgram");
    getenvproc = GetProcAddress(hdl1,"getEnv");
    searchproc = GetProcAddress(hdl1,"searchPath");
    geterrproc = GetProcAddress(hdl1,"getLastError");
    strbuf=newcbuffer(400);
}

/**
Destructor frees releases the library.
*/
System::~System()
{
    FreeLibrary(hdl1);
}

/**
Method encapsulates call to execProgram library function.
@param cmd name of program to execute (string)
@param cmdArgs command line arguments for program (string)
@return return code of execution (or -1 on error)
*/
int System::exec(string cmd, string cmdArgs="")
{
    return CallLibFunc(execproc,addr(cmd),addr(cmdArgs));
}

/**
Method encapsulates call to getEnv library function.
@param name name of environment variable to get value from (string)
@return value of environment variable (empty string if not found)
@note If length of result exceeds 400 characters it is truncated.
*/
string System::getEnv(string name)
{
    strbuf[0] = 0;
    CallLibFunc(getenvproc,addr(name),400,addr(strbuf));
}

```

```

    return string(strbuf);
}

/**
Method encapsulates call to searchPath library function.
@param name name of file to search for in PATH
@return full file path if found, otherwise empty string
*/
string System::searchPath(string name)
{
    strbuf[0] = 0;
    CallLibFunc(searchproc, addr(name), 400, addr(strbuf));
    return string(strbuf);
}

/**
Method encapsulates call to GetLastError library function.
@return description of last error occurred in DLL
@note Currently error description is supported only in C++ version
of system library.
*/
string System::getLastError()
{
    strbuf[0] = 0;
    CallLibFunc(geterrproc, 400, addr(strbuf));
    return string(strbuf);
}

```

Beispiel 95 Kapselung einer Funktionsbibliothek als B++-Klasse

Den wichtigsten Teil ihrer Arbeit erledigt diese Klasse im Konstruktor bzw. Destruktor.

Der Konstruktor übernimmt das Laden der Bibliothek, ermittelt die Adressen der benötigten DLL-Funktionen und weist sie an entsprechende Member-Variablen zu. Außerdem legt er einen internen Zeichenpuffer an, der für die Rückgabe von Ergebnissen der Funktionsaufrufe benötigt wird.

Der Destruktor sorgt bei der Zerstörung einer Objektinstanz der Klasse automatisch für die Freigabe der Bibliothek.

Die bereitgestellten Methoden schließlich sorgen für die korrekte Ausführung der `CallLibFunc`-Aufrufe.

Mit der Übersetzung der Klasse in Bytecode entsteht ein Bibliotheksmodul wie in Abschnitt 15.1 beschrieben. Entsprechend einfacher und übersichtlicher wird nun die Benutzung der Bibliothek im Anwendungsprogramm:

```

// import precompiled System class
#include "system.bpm"

void main()
{
    // create instance of System class
    System sys;
    // now execute system library directly
    // a message should be printed
    int result = sys->exec("system.exe", "");
    if (result < 0)
    {
        // on error print error message
        print("Error:", sys->getLastError(), "\n");
    }
    // ... and print result
    print("Execution result: ", result, "\n");
    // get value of PATH environment variable
    string s = sys->getEnv("PATH");
    // ... and print it
    print("PATH (", strlen(s), "): ", s, "\n");
    // search for command.com and print its path
    print("command.com: ", sys->searchPath("command.com"), "\n");
}

```

In diesem Beispiel werden alle Funktionen der System-DLL aufgerufen. Der zusätzliche Aufwand reduziert sich auf das Anlegen Variablen `sys`.

15.3.2 B++-spezifische Bibliotheken

Da B++-spezifische Bibliotheken die von ihnen exportierten Klassen bzw. Funktionen in die Symboltabellen der B++-VM eintragen, gestaltet sich ihre Benutzung bedeutend einfacher als die allgemeiner Funktionsbibliotheken.

Eine solche Bibliothek wird der Verarbeitungsanweisung

```
#import "dateiname.ext"
```

(vgl. Abschnitt 4.5.1) in ein Programm (bzw. Modul) aufgenommen.

Das Laden der Bibliothek erfolgt implizit während der Übersetzung oder beim Laden eines vor-kompilierten Moduls. Dabei vollzieht sich intern folgender Ablauf:

- Lokalisieren und Laden der Bibliothek in den Speicher,
- Lokalisieren der Funktion `RegisterNativeModule` in der geladenen Bibliothek,
- Aufruf dieser Funktion mit der VM als Argument.

Die Implementierung von `RegisterNativeModule` trägt dabei die Klassen und Funktionen der Bibliothek in die Symboltabellen der VM ein (vgl. 15.2.2). Danach stehen die Symbole im B++-Programm Verfügung.

Aus Anwendersicht (B++-Programmierersicht) verhalten sich die Bibliotheksfunktionen (bzw. Methoden von Klassen) wie die vordefinierten Funktionen von B++. Dies betrifft auch die Aussage, dass sie keine statische Typisierung im eigentlichen Sinne, d. h. vom B++-Compiler oder dem Laufzeitsystem überprüfbar, unterstützen. Selbstverständlich können Bibliotheksfunktionen in ihrer Implementierung natürlich Typprüfungen und/oder -konvertierungen vornehmen.

Von B++-Bibliotheken exportierte Klassen sind aus B++-Sicht ‚ganz normale‘ Klassen. Insbesondere können sie als Basisklassen für in B++ implementierte Klassen verwendet werden⁸².

Hinweis:

B++ kennt keine Standardfunktion, die es – analog zur Funktion `loadmodule` (siehe 11.9) – gestattet, dynamische B++-Bibliotheken zur Laufzeit zu laden. Wird diese Funktionalität benötigt, lässt sie sich aber leicht über ein Hilfsmodul bereitstellen, das lediglich die `#import`-Anweisung für die Bibliothek enthält und dann seinerseits mit `loadmodule` geladen werden kann.

⁸²Theoretisch geht es auch umgekehrt: Es ist technisch durchaus möglich, in einer Bibliothek eine ‚native‘ Klasse zu implementieren, die von einer in B++ implementierten erbt. Ob es hierfür einen sinnvollen Anwendungsfall gibt, ist allerdings eher fraglich.

16 Einbetten von B++

Ein typisches Anwendungsgebiet einer Skriptsprache ist ihre Verwendung innerhalb einer Applikation zur benutzerdefinierten Steuerung und/oder Erweiterung von deren Funktionalität, also ihre Einbettung in die betreffende Applikation.

B++ bietet zu diesem Zweck zwei Schnittstellen:

- eine C++-Schnittstelle zur Einbindung in Anwendungen, die selbst in C++ implementiert sind, sowie
- eine C-Schnittstelle für die Verwendung in anderen Umgebungen, die eine Teilmenge der C++-Schnittstelle als Satz von Funktionen bereitstellt.

In der B++-Distribution werden beide Varianten verwendet. Die Kommandozeilenversionen sind gewissermaßen Minimalanwendungen, die die B++-VM über die C++-Schnittstelle einbinden. Die in C# implementierte B++-IDE verwendet dagegen die C-Schnittstelle. Die zugehörigen Quelltexte können als Referenzimplementierungen und Beispiele für die Einbindung von B++ angesehen werden.

Für die Steuerung (Automatisierung) einer konkreten Applikation mit B++ kann die betreffende Applikation spezifische Funktionen und/oder Klassen für B++ bereitstellen, die in der in Abschnitt 15.2.2 beschriebenen Form implementiert werden. Sofern diese spezifische Funktionalität nicht in einer separaten Bibliothek (Dll) untergebracht wird, muss die Applikation für die Registrierung in der B++-VM, also u. a. den Aufruf von `registerNativeFunction` bzw. `registerNativeClass` sorgen.

17 Quellen

- [1] Bob-Source <http://www.atari-portfolio.co.uk/library/language/bob.zip>
- [2] Betz, D. M.: Bob: A Tiny Object-Oriented Language. In: Dr.Dobbs Journal, Sep. 1991, S.26ff.
<http://www.drdobbs.com/open-source/bob-a-tiny-object-oriented-language/184409401#>
- [3] Neue Bob-Sourcen von D. M. Betz: XLISP HomePage, <http://www.mv.com/ipusers/xlisper/bob.zip>
- [4] Ebert, R.-E.: DOS-DLL – Dynamische Bibliotheken für MS-DOS. Berlin 2007
http://www.kiezsoft.de/download/dosdll_manual_de.pdf
- [5] van Heesch, Dimitri: Doxygen – Generate documentation from source code.
<http://www.doxygen.org> bzw. <http://www.stack.nl/~dimitri/doxygen/index.html>
- [6] Demichelis, Alberto: T-Rex a tiny regular expression library.
<http://tiny-rex.sourceforge.net>
- [7] ECMAScript regular expression pattern syntax.
<http://www.cplusplus.com/reference/regex/ECMAScript/>