B++
Programmierhandbuch

Oktober 2026

B++ Programmierhandbuch

Oktober 2026

© 2026 Ralf-Erik Ebert, Frankfurt (Oder)

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 19

4.3.5 Erweiterte Literale 19

4.3.6 Literale Ausdrücke 20

4.3.7 Benannte Literale 20

4.3.8 Vordefinierte Literale 21

4.4 Definitionen, Anweisungen und Blöcke 23

4.5 Verarbeitungsanweisungen 25

4.5.1 Die #include-Anweisung 25

4.5.2 Die #use-Anweisung 26

4.5.3 Die #import-Anweisung 27

4.5.4 Die Anweisungen #deflit, #literal und #undeflit 27

4.5.5 Die Anweisungen #define und #undef 28

4.5.6 Die #defvar-Anweisung 29

4.5.7 Die Anweisungen #ifdef, #ifndef, #if, #elif, #else und #endif 29

4.5.8 Die Anweisungen #error und #warning 30

4.5.9 Die #pragma-Anweisung 30

4.5.10 Kennzeichnen literaler Ausdrücke mit #() 31

4.5.11 Die #endsrc-Anweisung 32

5 VARIABLEN, KONSTANTEN UND DATENTYPEN 33

5.1 Variablen 33

5.2 Konstanten 34

5.3 Datentypen 34

5.3.1 Werttypen 36

5.3.2 Referenztypen 39

5.4 Statisch typisierte Variablen 43

5.5 Initialisierung von Variablen und Konstanten 44

5.6 Implizite und explizite Typkonvertierungen 46

5.7 Vordefinierte Variablen und Konstanten 47

6 OPERATOREN 48

6.1 Arithmetische Operatoren 48

6.2 Vergleichsoperatoren 49

6.3 Logische Operatoren 50

6.4 Bitoperatoren 50

6.5 Zuweisungsoperatoren 51

6.6 Inkrement- und Dekrement-Operatoren 52

6.7 Sonstige Operatoren ... 53
6.8 *Rangfolge* der Operatoren ... 56
7 SCHLÜSSELWÖRTER UND RESERVIERTE BEZEICHNER ... 57
8 STEUERSTRUKTUREN ... 60
8.1 Bedingungen (if-else) ... 60
8.2 Fallunterscheidungen (switch-case) ... 60
8.3 Wiederholungen (Schleifen) ... 61
8.3.1 Die while-Anweisung ... 61
8.3.2 Die do-while-Anweisung ... 61
8.3.3 Die for-Anweisung ... 62
8.3.4 Geschachtelte Schleifen ... 63
8.3.5 Die Anweisungen break und continue ... 63
8.4 Ausnahmebehandlung (try – catch – throw) ... 64
9 FUNKTIONEN ... 65
9.1 Funktionsdefinition ... 65
9.2 Aufruf von Funktionen ... 67
9.3 Emulation von Referenzparametern ... 67
9.4 Variable Parameterlisten und Vorgabeparameter ... 68
9.5 Statische Variablen in Funktionen ... 69
9.6 Funktionszeiger ... 70
9.7 Überladen von Funktionen ... 70
9.8 Anonyme Funktionen und Funktionsliterale ... 71
9.9 Coroutinen ... 72
10 OBJEKTORIENTIERTE PROGRAMMIERUNG ... 75
10.1 Aufbau einer Klasse ... 75
10.2 Inline-Definition von Klassen ... 78
10.3 Konstruktoren und Destruktoren ... 78
10.4 Zugriff auf Elementfunktionen und -variablen innerhalb einer Klasse ... 81
10.5 Virtuelle Methoden ... 82
10.6 Erweiterte Syntax des Operators -> ... 84
10.7 Methodenzeiger und Methodenliterale ... 85
10.8 Lokale Klassen ... 86
10.9 Partielle Klassen ... 88
10.10 Ersetzen von Member-Funktionen (Redefinition) ... 90
10.11 Definieren von Operatoren ... 91
10.11.1 Definieren des Funktionsoperators ... 91
10.11.2 Definieren des Indexoperators ... 92
10.11.3 Definieren des Punktoperators ... 94
10.11.4 Definieren der Inkrement- und Dekrement-Operatoren ... 95
10.11.5 Definieren des Zuweisungsoperators ... 96
10.11.6 Definieren des Dereferenzierungsoperators ... 96
10.11.7 Definieren der weiteren unären Operatoren ... 97
10.11.8 Indirekte Definition von Vergleichsoperatoren mit den Methoden compare bzw. equals ... 97
10.11.9 Binäre Infix-Operatoren ... 98
10.11.10 Operatoren zur Typumwandlung ... 101
11 NAMENSRÄUME ... 102
12 SYNCHRONE ZEITGEBER UND ANGEHÄNGTE FUNKTIONSERWEITERUNGEN ... 103
12.1 Synchrone Zeitgeber ... 103
12.2 Angehängte Funktionserweiterungen ... 103
13 VORDEFINIERTE FUNKTIONEN ... 105
13.1 Standardfunktionen für Container-Datentypen ... 105
13.1.1 Container-Konstruktorfunktionen ... 105

13.1.2 Allgemeine Containerfunktionen....109
13.1.3 Vektor-Funktionen....121
13.1.4 Dictionary-Funktionen....124
13.2 Typkonvertierungen und RTTI....126
13.2.1 Funktionen zur Typkonvertierung....126
13.2.2 RTTI-Funktionen....132
13.3 Ein-/Ausgabe....140
13.4 Zeichenkettenfunktionen....151
13.5 Datum und Uhrzeit....157
13.6 Mathematische Funktionen....161
13.7 Systemfunktionen....164
13.7.1 Funktionen zur Prozesssteuerung....174
13.7.2 MS-DOS-Funktionen....179
13.7.3 Windows-Funktionen....182
13.8 Callbacks....184
13.9 MS-DOS-Grafikfunktionen....187
13.10 Sonstige Funktionen....189
14 REGULÄRE AUSDRÜCKE....200
14.1 Pattern-Syntax....200
14.2 Die vordefinierte Klasse RegEx....203
14.2.1 Methoden....204
14.2.2 Operatoren....212
15 EMULATION VON C-STRUKTUREN....215
15.1 Einfache Strukturen....215
15.2 Gepackte Strukturen....217
15.3 Unions....218
15.4 Anonyme Strukturen und Unions....219
15.5 Felder variabler Länge in Strukturen....219
15.6 Makros zur vereinfachten Definition von Strukturen und C-Arrays....221
16 SPEICHERVERWALTUNG....222
16.1 Weak References und die unären Operatoren & bzw. *....223
16.2 Explizite Speicherverwaltung....228
17 INTERNE DATENSTRUKTUREN....229
17.1 Globale Symboltabellen und Typen von Symbolen....229
17.2 Struktur von Klassen und Objekten....229
17.3 Struktur von Funktionen und Aufbau des Bytecodes der VM....230
17.4 Organisation des Stacks der VM....237
18 ERWEITERN VON B++....239
18.1 B++-Bibliotheksmodule....239
18.2 Erweitern von B++ mit dynamischen Bibliotheken....241
18.2.1 Allgemeine Funktionsbibliotheken....241
18.2.2 B++-spezifische Bibliotheken....241
18.2.3 Dynamische Bibliotheken für MS-DOS....249
18.3 Verwenden dynamischer Bibliotheken....258
18.3.1 Allgemeine Funktionsbibliotheken....258
18.3.2 B++-spezifische Bibliotheken....262
19 EINBETTEN VON B++....263
20 QUELLEN....264

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 gepflegt[1].

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“ Module[2]
- 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 gilt ein besonderer Dank 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 Plattform[3].

Wie für Skriptsprachen üblich, erlaubt es B++, mit *dynamisch typisierten* Variablen zu arbeiten. Daneben steht auch eine *statische Typisierung* (siehe Abschnitt 5.4) 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 insgesamt 36 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 Emulation[4]
- explizit verwendbare Integer-Datentypen von 8 bis 64 Bit Breite, jeweils mit und ohne Vorzeichen, wobei 32 Bit die Standardgröße ist[5]
- integrierte Datentypen für Zeichenketten, Puffer, Vektoren (dynamische Arrays) und Dictionarys (assoziative Container)
- ganzzahlige numerische Literale in dezimaler, binärer, oktaler oder hexadezimaler Schreibweise
- Literaldarstellungen für Vektoren und Dictionarys
- Funktions- und Methodenzeiger
- anonyme Funktionen und Methoden
- asymmetrische Coroutinen
- 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)

[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.

- Zugriff auf Systemfunktionen
- integrierte Serialisierung beliebiger Daten
- Verwendung vorkompilierter Bytecode-Bibliotheken
- dynamisches Laden von Bibliotheken zur Laufzeit
- wahlweise explizite Speicherverwaltung oder automatische Speicherverwaltung mit Referenzzählung
- optional statische Typisierung

Hinzu kommen eine Vielzahl vordefinierter Funktionen (vgl. Kapitel 13), welche die Aufgaben einer Standardbibliothek erfüllen, sowie vordefinierte Klassen zur Unterstützung regulärer Ausdrücke (Kapitel 14) und zur Emulation von C-Strukturen (Kapitel 15).

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.

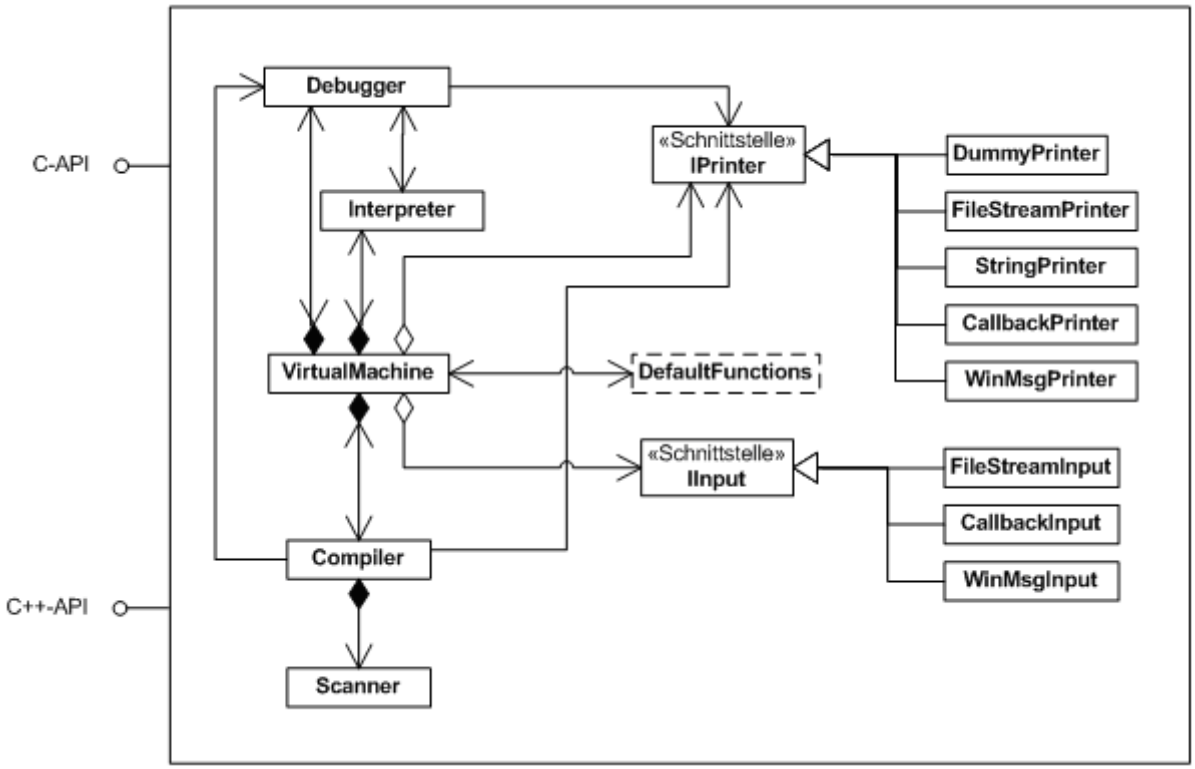


Abbildung 1 Grobstruktur von B++

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 Codeerzeugung[6]. 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

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

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.

2.3 *Die B++ Distribution*

B++ ist derzeit für folgende Plattformen verfügbar:

- MS-DOS ab Version 3.0[7]
- Windows-Desktop 32 Bit (XP, Vista, Win7/8/10/11)
- Windows-Desktop 64 Bit (Win 7/8/10/11)
- Linux (nur als Sourcecode)

Zur vollständigen Distribution gehören

- je ein Installer (Setup) für die Windows-Desktop-Versionen
- ein ZIP-Archiv mit den portablen Windows-Desktop-Versionen (32 und 64 Bit)
- 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 2022 (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

[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

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.

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-Modul[10] 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

[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.

- -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 „.bpp“ 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.

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 13.10).
Die Liste der Benutzerargumente wird beim Aufruf von der Kommandozeile auch der Eintrittsfunktion `main` übergeben (nach dem Programmnamen), so dass mit Hilfe der Funktionen `argcnt()`, `arg(i)` bzw. `argv()` ebenfalls ein Zugriff auf diese Argumente möglich ist[12].

Im Gegensatz zu BOB+ verwendet B++ einen dynamischen Stack, dessen Größe nur durch den verfügbaren Speicherplatz begrenzt ist. Eine explizite Festlegung der Größe[13] 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 Suchpfad[14].

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.

[11]Unter Unix/Linux wird kein „.exe“ angehängt, der Standard-Ausgabename ist also einfach „out“.
[12]Beim Aufruf eines Runtime-Moduls bzw. einer direkt ausführbaren Datei (.exe) wird die vollständige Kommandozeile an *`main`* übergeben.
[13] Diese Möglichkeit gibt es für BOB+.
[14]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 normalerweise[15] die Funktion `main()`. Neben reinen Funktionen können auf der obersten Strukturebene zusätzlich Verarbeitungsanweisungen (siehe Abschnitt 4.5), Variablendefinitionen (vgl. Kapitel 5) und Klassendeklarationen/-definitionen (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. Allerdings können lokal definierte Funktionen mit Hilfe anonymer Funktionen (vgl. Abschnitt 9.8) nachgebildet werden, die einer lokalen Variablen zugeordnet sind.

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 Funktionen[16] `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

[15] Dies gilt zumindest für die Kommandozeilenversion. Eine Anwendung, die die virtuelle Maschine von B++ integriert, kann prinzipiell jede beliebige Funktion als Einsprungpunkt aufrufen.
[16] siehe hierzu Kapitel 13

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 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) zurückgegeben[17]. 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 Rückgabewertes[18]. 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)
      putc(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 5.4.

[17] Das gilt nur für die Kommandozeilenversion. Eine Anwendung, die B++ integriert, kann mit dem Wert natürlich machen, was sie will.
[18] 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 4.5.11.

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-Symbol[19] ($). 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 5.3). Hinzu kommen in B++ erweiterte Formen zur Initialisierung von Vektoren und Dictionarys (siehe 5.3.2) sowie Funktionsliterale (4.3.5) und literale Ausdrücke (4.3.6).

[19] 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 255[20], 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)
`\0`	NUL-Byte	`00`
`\a`	Alarmton	`07`
`\b`	Backspace	`08`
`\f`	Seitenvorschub	`0C`
`\n`	Zeilenvorschub	`10`
`\r`	Wagenrücklauf	`0D`
`\t`	Tabulator horizontal	`09`
`\v`	Tabulator vertikal	`0B`
`\\`	Backslash	`5C`
`\'`	Apostroph	`27`
`\"`	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`' geschrieben werden.

Darüber hinaus werden analog zu C++11 auch die Escape-Sequenzen '`\uXXXX`' (UTF-16-Zeichen) bzw. '`\UXXXXXXXX`' (UTF-32-Zeichen) interpretiert, wobei `XXXX` für eine Folge von Hexadezimalziffern steht[21].

Zeichenliterale (bzw. Zeichentypen) sind integrale Typen und können wie ganze Zahlen behandelt werden, 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[22], die in Anführungszeichen eingeschlossen sind. Kommt das

[20] In der Unicode-Version von B++ ist der größtmögliche Zeichencode FFFFH.
[21] Bis zur Version 1.4 von B++ wurden die beiden Sequenzen wie die \xXXXX-Sequenz behandelt, also einfach als hexadezimaler Zahlenwert, der als Zeichencode interpretiert wurde. Ab Version 1.5 ergeben sich aus den Sequenzen die Typen `char16_t` bzw. `char32_t`.
[22] Die Zeichen in Zeichenketten sind immer vom internen Zeichentyp, also `char` für Singlebyte-Builds bzw. `wchar_t` für Unicode-Builds. Unicode-Sequenzen (\u bzw.\U) werden deshalb implizit in den internen Zeichensatz konvertiert.

Anführungszeichen selbst in einer Zeichenkette vor, so muss es als Escape-Code angegeben werden. Zeichenkettenliterale können beliebig lang sein.

Beispiele für Zeichenkettenliterale:

```
“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 Zeichenkettenliterale direkt verkettet werden. Dies erlaubt das einfache Verteilen langer Zeichenkettenliterale auf mehrere Zeilen.

Beispiel:

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

Anders als in C/C++ sind Zeilenumbrüche innerhalb von Zeichenkettenliteralen 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 Literale

Ganzzahlige Literale sind Darstellungen ganzer Zahlen in dezimaler, binärer, oktaler oder hexadezimaler Schreibweise, denen optional ein Vorzeichen (+ oder -)[23] 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 Dezimalzahlen[24]. Standardmäßig werden ganzzahlige Literale 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 $+2^{63}-1$.

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

[24] Eine ganze Dezimalzahl kann nicht mit der Ziffer 0 beginnen!

Wie in C++ können dem Literal die Suffixe U und L bzw. LL[25] (auch kombiniert, Groß- bzw. Kleinschreibung wird nicht berücksichtigt) angehängt werden, um die interne Darstellung als vorzeichenlose Konstante 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
0x7FFFFFFUL       // 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

Erweiterte Suffixe:

Mit Version 1.3 von B++ wurde die Interpretation von Suffixen ganzzahliger Literale modifiziert bzw. erweitert. Die Zielstellung besteht darin, den Datentyp des resultierenden Wertes exakt festlegen zu können.

Im Einzelnen gelten folgende Regeln:

- Das Suffix S (oder s) erzwingt einen vorzeichenbehafteten Wert. Es kann nicht mit dem Suffix U kombiniert werden.
- Dem Suffix L kann eine Ziffer 0, 1, 2, 4 oder 8 nachgestellt sein. Die Ziffer 0 legt fest, dass der ‚kleinste' Datentyp verwendet werden soll, der den Wert darstellen kann; die übrigen Ziffern legen die geforderte Datentypgröße in Bytes fest.
- Ist kein Suffix L angegeben, ist das Ergebnis in Abhängigkeit vom Wert 32 oder 64 Bit breit.
- Ist kein Suffix U oder S angegeben, wird der vorzeichenbehaftete Typ bevorzugt, das bedeutet, der vorzeichenbehaftete Typ wird verwendet, falls der Betrag des Wertes damit darstellbar ist, andernfalls der vorzeichenlose Typ.
- Ist die Datentypgröße explizit angegeben, wird garantiert, dass der Wert mit dem resultierenden Typ darstellbar ist.
- Ist das Suffix S angegeben und der Wert in Dezimaldarstellung notiert, wird garantiert, dass der Betrag des Wertes im positiven Wertebereich des resultierenden Typs darstellbar ist[26].
- Verletzungen der genannten Garantien führen zu einem Fehler bei der Übersetzung.

Beispiel 4 demonstriert einige Varianten der Notation mit erweiterten Suffixen:

[25] Weil der Typ `long` in B++ generell 64 Bit breit ist, ist die Bedeutung der Suffixe L und LL identisch.
[26] Die Regel garantiert, dass der negierte Wert ebenfalls im Wertebereich liegt.

```
17L1        // 8-Bit-Integer in Dezimaldarstellung mit Vorzeichen (implizit)
128L1       // 8-Bit-Integer in Dezimaldarstellung ohne Vorzeichen (implizit)
17SL1       // 8-Bit-Integer in Dezimaldarstellung mit Vorzeichen (explizit)
17UL1       // 8-Bit-Integer in Dezimaldarstellung ohne Vorzeichen (explizit)
128SL1      // --> Fehler: 128 nicht im Bereich 0..127 darstellbar
0x80SL1     // 8-Bit-Integer hexadezimal mit Vorzeichen (explizit)
0x8000SL0   // 32-Bit-Integer mit Vorzeichen
0x8000L0    // 16-Bit-Integer ohne Vorzeichen
```

Beispiel 4 Notation ganzzahliger numerischer Literale mit erweiterten Suffixen

Hinweis:

Die erweiterten Suffixregeln sind weitestgehend kompatibel zu den bisherigen. Werden die Erweiterungen nicht benutzt, entspricht auch das Resultat dem bisherigen – mit einer Ausnahme: Überschreitet der Wert eines Literals ohne Angabe des Suffixes U den positiven Wertebereich des Typs `long`, nicht aber den von `ulong`, so wird kein Fehler gemeldet, sondern stillschweigend ein Ergebnis vom Typ `ulong` erzeugt.

4.3.4 Gleitkommaliterale

Gleitkommaliterale sind Darstellungen reeller[27] 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 $-1.7*10^{308}$ und $1.7*10^{308}$ liegt. Mit dem Suffix F (bzw. f) wird explizit die Verwendung einfacher Genauigkeit (4 Byte, Typ `float`, Wertebereich zwischen $-3.4*10^{38}$ und $3.4*10^{38}$ bei 7 Stellen der Mantisse) festgelegt.

Beispiele für Gleitkommaliterale:

```
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 5 Notation von Gleitkommaliteralen

4.3.5 Erweiterte Literale

Variablen der Container-Datentypen `buffer`, `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 5.3.2) 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 9.8.

[27] Wegen 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 während der Übersetzung 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. 4.5.7) zur Auswertung von Bedingungen von Bedeutung.

Mit Version 1.7 wurde das Konzept der literalen Ausdrücke dahingehend erweitert, dass ihre Auswertung entweder (wie bisher und als Standard) direkt vom Compiler vorgenommen wird oder zur Auswertung eine temporäre Übersetzung in Bytecode und dessen Interpretation erfolgt.
Letzteres erlaubt es, für Operanden literaler Ausdrücke alle im Kontext der Übersetzung bereits vorhandenen globalen und statischen Symbole zu verwenden. Dies betrifft globale Variablen, Konstanten und Funktionen, bereits definierte Klassen, statische Elemente von Klassen oder statische lokale Variablen bzw. Konstanten in Funktionen.
Der Preis für diese Erweiterung ist eine Verlängerung der Übersetzungszeit: Für jeden literalen Ausdruck wird dann zunächst Bytecode in Form einer temporären Funktion generiert und anschließend interpretiert.
Weil die Erweiterung gewöhnlich nur in seltenen Fällen benötigt wird, ist sie standardmäßig deaktiviert. Sie lässt sich mit Hilfe der Verarbeitungsanweisung `#pragma compexpr` (4.5.9) ein- und ausschalten.

Explizite literale Ausdrücke:

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 oder Konstanten verwendet werden – auch dann, wenn ihre Operanden Literale (und damit konstant) sind.

Mit Hilfe des Schlüsselwortes `litexp`[28] (vgl. Kapitel 7) kann ein Ausdruck `<expr>` in der Form

```
litexp(<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.
Ab Version 1.7 kann die Kennzeichnung stattdessen auch in der Form

```
compexp(<expr>)
```

erfolgen, wobei das Schlüsselword `compexp` die erweiterte Form der Interpretation erzwingt.

Die Schlüsselwörter `litexp` (für die einfache Form) bzw. `compexp` (für die erweiterte) überschreiben für den jeweiligen Ausdruck das Standardverhalten zur Interpretation literaler Ausdrücke (`#pragma compexpr`). Sie können auch innerhalb eines Ausdrucks gemischt werden, um die Interpretation von Teilen eines komplexeren literalen Ausdrucks zu steuern.

4.3.7 Benannte Literale

Literalen 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.

[28] Das Schlüsselwort existiert ab Version 1.3. Zuvor war die Kennzeichnung nur über die Verarbeitungsanweisung `#(<expr>)`, siehe 4.5.10, möglich.

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

Für den Datentyp `bool` existieren die vordefinierten Literale `true` und `false` zur Darstellung dessen möglicher Werte[29].

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 sind vom Typ `int` haben den Wert 1 (bzw. *`true`*), wenn beim Kompilieren die jeweilige Gegebenheit zutrifft, und 0 (bzw. *`false`*), wenn nicht.

Das vordefinierte Literal `__bplusplus`[30] ist vom Typ `uint` und beinhaltet das 100-fache der Versionsnummer des Compilers bei der Übersetzung, also z.B. 130 für die Version 1.30.

Ebenfalls vordefiniert sind `PTRSIZE` (Typ `uint`), das die Größe einer Adresse des bei der Übersetzung verwendeten Systems in Bytes angibt, und `NPOS` (ab Version 1.4 und als `uint(-1)` definiert) zur Kennzeichnung eines ungültigen Indexwertes.

Ab Version 1.5 gibt es außerdem die vordefinierten Literale `CHARSIZE` und `WCHARSIZE` (beide `uint`), die die Größe der internen Zeichendarstellung bzw. die Typgröße von `wchar_t` angeben.

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 13.6) 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 5.3) 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 13.2) 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 5.4.

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 13.7) , `SEEK_SET` (0), `SEEK_CUR` (1) und `SEEK_END` (2) für die Ursprungsangabe der Funktion `fseek` (siehe Abschnitt 13.3) sowie `P_WAIT` (0), `P_NOWAIT` (1), `P_OVERLAY` (2), `P_NOWAITO` (3) und `P_DETACH` (4) für die Modi der Funktion `spawn` (siehe Abschnitt 13.7.1).

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.

---

[29] Bis zur Version 1.3 von B++ gab es diese Literale ebenfalls, sie wurden jedoch einfach auf die Integer-Werte 1 (true) bzw. 0 (false) abgebildet.

[30] Das Literal ist ab Version 1.3 definiert.

Literal	Typ-ID	Datentyp
BVT_NULL	0	**null**
BVT_VOID	1	kein Wert (Ergebnistyp für Funktionen)
BVT_AUTO	2	uninitialisierte **auto**-Variable
BVT_NIL	3	**null** (Ergebnistyp für Funktionen)
BVT_SBYTE	64	**sbyte**
BVT_BOOL	65	**bool**
BVT_CHAR	66	**char**
BVT_BYTE	68	**byte**
BVT_UCHAR	70	**uchar**
BVT_U8CHAR	71	**u8char**
BVT_SHORT	72	**short**
BVT_USHORT	76	**ushort**
BVT_CHAR16	78	**char16_t**
BVT_WCHAR	79	**wchar_t**
BVT_INT	80	**int**
BVT_INTPTR	81	**intptr**
BVT_UINT	84	**uint**
BVT_UINTPTR	85	**uintptr**
BVT_CHAR32	86	**char32_t**
BVT_LONG	88	**long**
BVT_ULONG	92	**ulong**
BVT_FLOAT	112	**float**
BVT_DOUBLE	120	**double**
BVT_FILE	128	**FILE**
BVT_CODE	129	**function** (nativ)
BVT_VAR	130	Variable/Element in dictionary
BVT_OBJECT	160	**object**
BVT_STRONGOBJECT	161	Objektvariable mit fester Klasse
BVT_CLASS	168	**Class**
BVT_STRING	176	**string**
BVT_BUFFER	177	**buffer**
BVT_CHARBUFFER	178	**charbuffer**
BVT_VECTOR	180	**vector**
BVT_BYTECODE	181	**function** (B++)
BVT_LOCALVARINFO	182	Variableninformation (intern)
BVT_DICTIONARY	184	**dictionary**
BVT_OBJECTREF	224	Referenz auf object
BVT_CLASSREF	232	Referenz auf Class
BVT_STRINGREF	240	Referenz auf string
BVT_BUFFERREF	241	Referenz auf buffer
BVT_CHARBUFFERREF	242	Referenz auf charbuffer
BVT_VECTORREF	244	Referenz auf vector
BVT_BYTECODEREF	246	Referenz auf function (B++)
BVT_DICTIONARYREF	248	Referenz auf dictionary

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 (`;`) abgeschlossen[31] 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:
```

[31] Das ist ein feiner aber wichtiger Unterschied zu Sprachen wie z. B. PASCAL, in denen 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.

```
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 };
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 6 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 große Ähnlichkeit zu Präprozessorbefehlen in C/C++ und werden auch so notiert. Insbesondere bedeutet dies, dass sie ***nicht*** mit einem Semikolon abgeschlossen werden. Wie Präprozessorbefehle in C/C++ können auch Verarbeitungsanweisungen an beliebiger Stelle in einem Quelltext notiert werden[32].

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>`.

[32]Dies gilt ab der Version 1.3 von B++. Zuvor durften Verarbeitungsanweisungen nur auf der globalen Ebene oder innerhalb einer Klassendeklaration bzw. innerhalb des äußersten Blocks einer Funktionsdefinition stehen.

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 7 Verwendung der #use-Anweisung

Hinweis:

Wenn der als Argument verwendete Dateiname ein Sternchen (*) enthält, wird dieses Zeichen durch eine Build-spezifische Kennung ersetzt, die folgende Bestandteile enthalten kann:

- u : bezeichnet eine Unicode-Version
- d : bezeichnet eine Debug-Version
- _64 : bezeichnet eine 64-Bit-Version

Dies erlaubt die Einbindung unterschiedlicher Bytecode-Module in Abhängigkeit von der zur Ausführung verwendeten Build-Variante der Laufzeitumgebung[33].

4.5.3 Die `#import`-Anweisung

Die `#import`-Anweisung erlaubt das Einbinden dynamischer B++-Erweiterungsbibliotheken (DLLs; siehe Abschnitt 18.2.2) 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 Suchpfad[34].

Hinweis:

Wenn der als Argument verwendete Dateiname ein Sternchen (*) enthält, wird dieses Zeichen durch eine Build-spezifische Kennung ersetzt, die folgende Bestandteile enthalten kann:

- u : bezeichnet eine Unicode-Version
- d : bezeichnet eine Debug-Version
- _64 : bezeichnet eine 64-Bit-Version

Dies erlaubt es, Bytecode-Module zu erzeugen, die bei der Ausführung auf einer anderen Zielplattform ggf. andere Erweiterungsbibliotheken einbinden.

4.5.4 Die Anweisungen `#deflit`, `#literal und #undeflit`

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

Mit

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

`#literal <identifier> <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.

Mit `#undeflit <identifier>` kann eine bestehende Definition eines benannten Literals aufgehoben werden.

[33]Ab Version 1.7. Im Normalfall ist eine solche Unterscheidung nicht nötig, sie kann aber z.B. erforderlich sein, wenn die Bibliothek plattformspezifische Implementierungen enthält.
[34]Unter Windows werden die spezifischen Regeln zur Suche nach DLLs verwendet. Demnach wird auch im Verzeichnis des aufrufenden Programms sowie im Windows-Verzeichnis gesucht.

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 8 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 Funktionsmakros in C vergleichbar. Der wesentliche Unterschied besteht darin, dass es sich hier um „echte“ (anonyme) Funktionen handelt, nicht um Makros, die erst bei ihrer Verwendung expandiert werden.

4.5.5 Die Anweisungen `#define` und `#undef`

Bis zur Version 1.4 war `#define` ein Alias für `#literal` und `#undef` ein Alias für `#undeflit`. Mit Version 1.5 entsprechen die Anweisungen weitestgehend den gleichnamigen Präprozessorbefehlen in C/C++, d. h., sie dienen der Definition von Makros.
Insbesondere werden dabei auch variadische Makros, also solche mit einer variablen Anzahl von Argumenten unterstützt (vgl. [6] und [9]). Ebenfalls wird die gcc-spezifische Erweiterung unterstützt, die die Notation von `##` vor `__VA_ARGS__` erlaubt, um ein vor einer variablen Argumentliste stehendes Komma zu unterdrücken, falls die variable Argumentliste leer ist.
Darüber hinaus kann der Verkettungsoperator `##` auch nach `__VA_ARGS__` notiert werden. In diesem Fall unterdrückt er ein ggf. nach einer leeren variablen Argumentliste stehendes Komma.
Mit Version 1.7 ist die Unterstützung von `__VA_OPT__(x)` hinzugekommen (expandiert x, falls `__VA_ARGS__` nicht leer. Als Umkehrung gibt es außerdem `__VA_DEF__(x)`, bei dem x expandiert wird, falls `__VA_ARGS__` leer ist.

Hinweise zur Verarbeitung von Makros:

Weil B++ keinen echten Präprozessor besitzt, müssen sowohl die Makronamen als auch die formalen Parameter immer Identifier im Sinne von B++ sein. Das Ersetzen von Schlüsselwörtern ist somit nicht möglich. Die Makros selbst werden intern nicht als Text, sondern als Tokenliste abgelegt. Entsprechend erfolgt die Ersetzung ebenfalls auf Basis der vorhandenen Tokenliste. Das bedeutet, dass Operationen, die bei der lexikalischen Analyse durchgeführt werden, bei der Makroexpansion nicht möglich sind – etwa die Erkennung von numerischen Literalen (vgl. 4.3.3 bzw. 4.3.4).

Bei der Expansion von Makros wird zunächst die Ersetzung der ggf. vorhandenen Parameter über den gesamten Makroinhalt vorgenommen – einschließlich der oben beschriebenen Behandlung variabler Argumentlisten.

Im nächsten Schritt werden verbliebene Verkettungsoperatoren (`##`) behandelt. Für die Verkettung sind folgende Token-Kombinationen (links `##` rechts) möglich:

- Identifier oder Schlüsselwort `##` Identifier, Schlüsselwort oder ganze Dezimalzahl
  → Ergebnis ist Identifier oder Schlüsselwort
- Sonderzeichen `##` Sonderzeichen, falls die Kombination ein Token ergibt
  → Ergebnis ist das entsprechende (Operator-)Token
- String mit beliebigem anderen Token (links oder rechts von `##`)
  → Ergebnis ist ein String

Danach werden Stringify-Operatoren (`#`) behandelt. Als Argumente (Folgetoken) sind hier beliebige Token mit Ausnahme der Bezeichner von Verarbeitungsanweisungen[35] möglich.

Im letzten Schritt werden direkt aufeinander folgende Zeichenketten zusammengefasst, um so die sonst vom Scanner realisierte Verkettung durchzuführen (vgl. 4.3.2).

4.5.6 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
```

Hinweise:

- 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.
- Mit Version 1.4 wurde das Verhalten der #defvar-Anweisung bei schon vorhandener Variable geändert: Sie wird nun *ignoriert*, während zuvor die bestehende Definition *überschrieben* wurde.

4.5.7 Die Anweisungen #ifdef, #ifndef, #if, #elif, #else und #endif

Seit 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.

[35] Aus ihnen dürfen keine Strings gebildet werden, weil sonst die Verwendung von Verarbeitungsanweisungen innerhalb von Makros (unterstützt ab Version 1.7) nicht möglich wäre.

Anweisung	Bedeutung
`#ifdef` *name*	Übersetzung, falls ein benanntes Literal (vgl. 4.3.7) oder ein Makro (vgl. 4.5.5) *name* definiert ist
`#ifndef` *literalname*	Übersetzung, falls kein benanntes Literal oder ein Makro *name* definiert ist
`#if` *literal_expression*	Übersetzung, falls *literal_expression* den Wert *true* ergibt
`#elif` *literal_expression*	bedingter Alternativzweig; Übersetzung, falls *literal_expression* den Wert *true* ergibt
`#else`	unbedingter Alternativzweig
`#endif`	Ende des bedingten Übersetzungsabschnitts

Tabelle 3 Übersicht der Anweisungen zur bedingten Übersetzung

Darin meint *literal_expression* einen literalen Ausdruck (vgl. 4.3.6), der auch expandierte Makros als Bestandteile enthalten kann.

Innerhalb eines literalen Ausdrucks kann wiederum der Pseudo-Operator

`defined(`*literalname*`)`

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

4.5.8 Die Anweisungen `#error` und `#warning`

Die Anweisungen werden zur Ausgabe eines Meldungstextes mit Bezug zur jeweiligen Code-Zeile während der Übersetzung verwendet.

`#error <message>` bzw.

`#warning <message>`,

wobei `<message>` ein literaler Ausdruck ist, der eine Zeichenkette ergibt.

Während `#warning` lediglich eine Meldung mit Zeilennummer und Position ausgibt, bricht `#error` die Übersetzung ab.

4.5.9 Die `#pragma`-Anweisung

Die `#pragma`-Anweisung dient der Steuerung des Compilers während der Übersetzung.

Sie hat die allgemeine Form

`#pragma` *<function> <parameter>*,

wobei folgende Funktionen (Bezeichner für <function>) unterstützt werden:

Funktion	Parameter	Beschreibung
`cdefine`	`on`, `off`, `push` oder `pop`	Legt fest, ob `#define` und `#undef` nach C-Standard verwendet werden (`on`, Standardeinstellung) oder einfach Aliase für `#deflit` bzw. `#undeflit` sind (`off`). Letzteres dient vor allem der Kompatibilität zu früheren Versionen. Mit `push` kann der aktuelle Status auf einem Stack gespeichert und mit `pop` wiederhergestellt werden. Wird `pop` ohne vorheriges `push` (bei leerem Stack) aufgerufen, führt dies zu einem Compilerfehler.
`compexpr`[36]	`on`, `off`, `push` oder `pop`	Legt fest, ob literale Ausdrücke (vgl. 4.3.6) direkt vom Compiler ausgewertet (`off`, Standardeinstellung) oder temporär in Bytecode übersetzt und interpretiert werden (`on`). Mit `push` kann der aktuelle Status auf einem Stack gespeichert und mit `pop` wiederhergestellt werden. Wird `pop` ohne vorheriges `push` (bei leerem Stack) aufgerufen, führt dies zu einem Compilerfehler.
`warning`[37]	Ausgabetext (literaler Ausdruck)	Gibt einen Text während der Übersetzung mit dem Präfix „Warning:“ und der nachfolgenden Codezeile aus. Dies entspricht dem Verhalten von `#warning`.
`message`	Ausgabetext (literaler Ausdruck)	Gibt einen Text während der Übersetzung aus.
`codepage`	Zahl oder Zeichenkette (literaler Ausdruck)	Legt die Kodierung der aktuellen Quelldatei fest. Wird als Parameter eine ganze Zahl angegeben, wird dies als Codepage-Nummer (nach Windows-Standard) verwendet. Andernfalls wird versucht, den Parameter als Kodierungsbezeichner zu interpretieren (z.B. „iso-8859-1“, „Latin-1“ oder „UTF-8“). Standardmäßig wird der Standard-Zeichensatz Betriebssystems angenommen.

Tabelle 4 Funktionen der `#pragma`-Anweisung

4.5.10 Kennzeichnen literaler Ausdrücke mit `#()`

Die Verarbeitungsanweisung war bis zur Version 1.2 von B++ erforderlich, um literale Ausdrücke explizit zu kennzeichnen (siehe 4.3.6). Seit Version 1.3 wird an ihrer Stelle die Verwendung des Schlüsselworts `litexp` bevorzugt, sie wird jedoch aus Gründen der Rückwärtskompatibilität weiter unterstützt und kann in der Form

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

zur Kennzeichnung von `<expr>` als literalen Ausdruck verwendet werden. Dabei darf *zwischen dem Hash-Symbol (#) und der öffnenden Klammer kein Leerzeichen* stehen.

[36] Ab Version 1.7
[37] Ab Version 1.9

4.5.11 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 9 Script-Einbettung von B++-Quellen

Das Programm[38] 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.

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

5 Variablen, Konstanten 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. 4.5.6).

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 10 Verwenden von lokalen Variablen

Bei der Entwicklung neuer Programme ist die Variablendeklaration/-definition mit `var`[39] (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 *Konstanten*

Konstanten wurden mit Version 1.5 von B++ eingeführt.

Technisch sind Konstanten Variablen mit der Restriktion, dass nach ihrer Initialisierung nur noch lesend auf sie zugegriffen werden kann. Ihre Definition erfolgt analog der der Variablen, wobei dem Typ bzw. dem (dann optionalen) Schlüsselwort `var` das Schlüsselwort `const` vorangestellt wird. Wie Variablen können auch Konstanten global, als Elemente von Klassen oder temporär (innerhalb der Implementierung von Funktionen) deklariert werden, auch für Gültigkeitsbereiche und Lebensdauer gelten dieselben Regeln wie für Variablen.
Anders als Variablen müssen Konstanten stets sofort initialisiert werden.
Ein Äquivalent zur Verarbeitungsanweisung `#defvar` gibt es für Konstanten nicht, ebenso gibt es keine Möglichkeit, Konstanten in Funktionskopf zu definieren.

5.3 *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* unterscheiden[40]. Diese Unterscheidung ist für die Verwendung in zweierlei Hinsicht von Bedeutung:

- Bei der Zuweisung an eine Variable 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 11 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 13.2) erzeugen.

- **Konstanten werden bei der Zuweisung an eine Variable stets kopiert** (tiefe Kopie), unabhängig davon, ob es sich um einen Wert- oder Referenztyp handelt.

[39] 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.
[40] 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.

Beispiel:

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

Beispiel 12 Zuweisung von Konstanten

Im Unterschied zu Beispiel 11 ist *s1* in Beispiel 12 als Konstante definiert, deren Wert bei der Zuweisung an die Variable *s2* vollständig[41] kopiert wird. Folglich lautet hier die Ausgabe „Hello“, weil die Änderung des zweiten Zeichens nur die Variable *s2* betrifft.
Das Schlüsselwort `var` in der Deklaration von *s1* hätte auch weggelassen werden können.

- 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.

[41] Hier wird tatsächlich eine „tiefe“ Kopie erzeugt, d.h., sofern die zugewiesene Konstante Elemente von Referenztypen enthält, werden auch diese rekursiv kopiert, so dass das Ergebnis vollständig unabhängig vom zugewiesenen Wert ist.

5.3.1 Werttypen

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

Integrale Datentypen

Diese Typen repräsentieren ganzzahlige numerische Werte unterschiedlicher Genauigkeit. Tabelle 5 gibt hierzu eine Übersicht.

Typ	Breite (Bits)	Wertebereich	Bemerkung
`bool`	8	false, true	Wahrheitswert, intern vorzeichenloses Byte
`sbyte`	8	-128 … 127	Byte mit Vorzeichen
`char`	8	-128 … 127	Singlebyte-Zeichen mit Vorzeichen (wie `char` in C)
`byte`	8	0 … 255	Byte ohne Vorzeichen
`uchar` (`char8_t`)	8	0 … 255	Singlebyte-Zeichen ohne Vorzeichen (wie `unsigned char` in C); `char8_t` ist ein Alias auf `uchar`
`short`	16	-2^{15} .. 2^{15}-1	`WORD` mit Vorzeichen
`ushort`	16	0 … 2^{16}-1	`WORD` ohne Vorzeichen (wie `unsigned short` in C)
`char16_t`	16	0 … 2^{16}-1	16-Bit-Unicode-Zeichen
`int`	32	-2^{31} … 2^{31}-1	`DWORD` mit Vorzeichen
`uint`	32	0 … 2^{32}-1	`DWORD` ohne Vorzeichen
`char32_t`	32	0 … 2^{32}-1	32-Bit-Unicode-Zeichen
`u8char`	32	Zeichenfolge	Zeichenfolgenrepräsentation eines UTF-8-Zeichens, LSB ist erstes Zeichen
`wchar_t`	8, 16 oder 32	Wie `uchar`, `char16_t` oder `char32_t` (plattformabhängig)	Wide Character (wie `wchar_t` in C)
`intptr`	32 oder 64	wie `int` oder `long` (plattformabhängig)	Repräsentation einer Adresse (vorzeichenbehaftet)
`uintptr`	32 oder 64	wie `uint` oder `ulong` (plattformabhängig)	Repräsentation einer Adresse (vorzeichenlos)
`long`	64	-2^{63} … 2^{63}-1	wie `long long` in C
`ulong`	64	0 … 2^{64}-1	wie `unsigned long long` in C

Tabelle 5: 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. Die Modifizierer `signed` und `unsigned` können zur Syntaxkompatibilität C/C++ mit in Verbindung mit den Typen `char`, `byte`, `short`, `int` und `long` (und nach denselben Regeln wie in C/C++) verwendet werden, sind aber nicht Bestandteil einer Typbezeichnung. So entspricht z. B. eine Variablendeklaration

```
unsigned short x
```

genau der Deklaration

```
ushort x.
```

Außerdem haben die integralen Typen in B++, mit Ausnahme von `wchar_t`, `intptr` und `uintptr`, eine feste Breite, unabhängig von Compiler und Ausführungsplattform. Dies erleichtert die Übertragung von Programmen – auch in vorkompilierter Form – zwischen verschiedenen Plattformen.

Der Datentypen `intptr` und `uintptr` dienen der, gewöhnlich temporären, Aufnahme plattformspezifischer Adressen. Sie sind deshalb auf 16- und 32-Bit-Plattformen 32 Bit bzw. auf 64-Bit-Plattformen 64 Bit breit. Intern wird `intptr` wie eine Ganzzahl mit, `uintptr` wie eine Ganzzahl ohne 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).

Bis Version 1.3 kannte B++ keinen „echten“ Booleschen Datentyp, stattdessen wurden `int`-Werte verwendet, wobei der Wert Null (0) als *`false`* und alle anderen Werte als *`true`* angesehen wurden. Allerdings konnte man den reservierten Bezeichner `bool` zur Deklaration von Variablen verwenden, um die Intention besser zu verdeutlichen.
Mit Version 1.4 ist `bool` ein eigenständiger Datentyp. In numerischen Operationen werden Werte vom Typ `bool` wie vorzeichenlose Bytes behandelt.
Die Zeichentypen wurden mit Version 1.5 eingeführt, zuvor erfolgte die Darstellung einzelner Zeichen (bzw. Zeichenliterale) ebenfalls als `int`. Werte von Zeichentypen sind integrale Typen und können deshalb in numerischen Ausdrücken verwendet werden, wobei der Zeichencode den Wert repräsentiert.
Der Typ `u8char` wurde mit Version 1.9 eingeführt. Zuvor wurde hierfür der Typbezeichner `char8_t` verwendet, der jedoch nicht dem gleichnamigen Typ in C++ entsprach. Ab Version 1.9 ist `char8_t` einfach ein Alias auf `uchar`.

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`[42] 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 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.

[42] 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.

5.3.2 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. 13.1) oder die Konvertierungsfunktion `string` (siehe 13.2) 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`-Literals mit dem Operator `[]` ist nicht zulässig.
- Bei der Zuweisung eines `string`-Literals an eine Variable wird eine Kopie der Zeichenkette angelegt. Auf sie kann uneingeschränkt zugegriffen werden.

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 13.3)

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 13 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 13 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`*`=false)` erzeugt. Dabei kann mit dem Argument *`sorted`* festgelegt werden, ob die Elemente entsprechend der alphanumerischen Abfolge der Schlüssel physisch sortiert (`true`) oder in der Reihenfolge des Hinzufügens (`false`) 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.

Ab Version 1.4 von B++ kann hinter der öffnenden geschwungenen Klammer ein Ausrufezeichen (!) notiert werden, das eine physische Sortierung nach den Schlüsseln erzwingt[43] – wie das *`sorted`*-Argument der Funktion `newdict` – siehe Abschnitt 13.1.1, S. 107

[43] Standardmäßig entspricht die physische Reihenfolge der Einträge im Dictionary der Einfügereihenfolge.

Beispiel:

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

Beispiel 14 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 13.3) sowie zur Übergabe von Datenbereichen an Systemfunktionen.

Instanzen vom Typ `buffer` werden mit Hilfe der Funktion `newbuffer(size, fillByte=0)` explizit erzeugt (siehe Abschnitt 13.1). 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 numerischen Werten mit Hilfe der Konvertierungsfunktion `buffer(arg)` (vgl. Abschnitt 13.2) Puffervariablen erzeugt werden, beispielsweise zur Übergabe als Referenzparameter an Systemfunktionen.

Ab Version 1.4 von B++ gibt es zudem eine Literaldarstellung zur Erzeugung eines Puffers in der Form

bufferliteral `::= '<<'` *litexpr* `.`

Dabei ist *litexpr* ein literaler Ausdruck, der den absoluten oder relativen Pfad zu einer Datei ergibt, aus der der Pufferinhalt initialisiert wird. Dies ermöglicht es, konstante Daten (Ressourcen) direkt in das erzeugte Programm aufzunehmen, ähnlich wie das Inkludieren von Quelldateien.
Die Suche nach der durch *litexpr* bestimmten Datei erfolgt dabei nach den gleichen Regeln wie für in Anführungszeichen eingeschlossene *include*-Dateien – vgl. Abschnitt 4.5.1.

Beispiel:

```
#literal textres << "externtext.txt"
void main() {
   print(string(charbuffer(textres)));
}
```

Beispiel 15 Verwenden eines Puffers zum Einbinden einer externen Datei als Ressource

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.

Hinweise:

- Der direkte Zugriff auf Elemente eines (benannten) `buffer`-Literals (vgl. Beispiel 15) mit dem Operator `[]` ist nicht zulässig.
- Bei der Zuweisung eines `buffer`-Literals an eine Variable wird eine Kopie des gesamten Puffers angelegt. Auf sie kann uneingeschränkt zugegriffen 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 oder 32 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 13.1). 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 unterschieden[44]. 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 wird[45].

[44] Deshalb liefert die Funktion `gettypename` (siehe 13.2) auch den Typnamen CODE für vordefinierte und BYTECODE für benutzerdefinierte Funktionen.
[45] Diese Aussage scheint auf den ersten Blick nicht ganz richtig: Wird ein Funktionsliteral, das in seiner Definition mindestens eine statische Variable enthält oder eine Coroutine repräsentiert (siehe 9.9), 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.

Zulässige Operationen für den Datentyp `function` sind:

- Zuweisung an eine Variable (vgl. Abschnitt 9.6)
- Aufruf
- Verwendung als Funktionsparameter

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.

Klassen sind die einzigen benutzerdefinierten Datentypen in B++.

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 13.2)
- 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

5.4 *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 22) 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 (hierzu zählt in diesem Zusammenhang auch der Typ `bool`) 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 Dateivariable). 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 10.11.10.

Darüber hinaus gibt es zur Umwandlung von Werten in die vordefinierten Datentypen Standardfunktionen zur expliziten Typkonvertierung. Sie sind in Abschnitt 13.2 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++11[46] – bei ihrer ersten Initialisierung auf den Typ des zugewiesenen Werts festgelegt wird. Deshalb müssen sie unmittelbar bei der Definition initialisiert werden[47].

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).

5.5 *Initialisierung von Variablen und Konstanten*

Die Initialisierung von globalen Variablen, Member-Variablen in einer Klassendefinition (vgl. Abschnitt 10.1) sowie statischen Variablen in Funktionen (9.5) und Vorgabewerten für Funktionsparameter (vgl. 9.4) 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 bzw.

var *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*)

[46]Das 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.

[47]Wird eine solche Variable mit einer Objektinstanz initialisiert, so erhält sie ab Version 1.8 den vollständigen Typ des zur Initialisierung verwendeten Objekts (`BVT_STRONGOBJECT`), angelehnt an das Verhalten von C++. Zuvor wurde der Typ `object` (`BVT_OBJECT`) verwendet.

verwendet werden.
Werden Variablen der eingebauten Standardtypen auf diese Weise initialisiert, erfolgt dabei ein impliziter Aufruf der Konvertierungsfunktion in den jeweiligen Zieltyp (vgl. Abschnitt 13.2). Bei der Initialisierung von Objekten (Variablen von einem konkreten Klassentyp) wird eine Instanz der entsprechenden Klasse erzeugt und deren Konstruktor aufgerufen.

Die Initialisierung von Konstanten entspricht genau der von Variablen, die Unterscheidung wird dadurch erreicht, dass der Deklaration das Schlüsselwort `const` vorangestellt wird.

Hinweise:

Weil der Gültigkeitsbereich von Konstanten (wie auch Variablen), die auf Funktionsebene definiert sind, die Funktion ist, sollten Konstanten nicht innerhalb von Schleifen definiert werden. Dies kann beim zweiten Schleifendurchlauf zu einem Laufzeitfehler führen[48].

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. 13.3). Eine Definition

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

entspricht also genau der Form

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

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

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 vom Typ `bool` werden mit dem Wert `false` initialisiert.
- Variablen der numerischen Typen werden mit dem Zahlenwert Null initialisiert.
- Variablen des Typs `string` werden als leere Zeichenketten initialisiert.
- Variablen der Containertypen (`buffer`, `charbuffer`, `vector`, `dictionary`) werden mit einem leeren Container initialisiert.
- Variablen anderer Referenztypen (`FILE`, Objekttypen) haben generell den Initialwert `null`, sofern es sich nicht um streng typisierte lokale (temporäre) Objektvariablen in Funktionen handelt.
- 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.

Hinweis:

Eine implizite Initialisierung von Konstanten ist nicht zulässig, sie müssen stets bei der Deklaration explizit initialisiert werden.

[48] Ab Version 1.9 gibt der Compiler deshalb eine Warnung aus.

5.6 *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`, sofern nicht ausschließlich Operanden vom Typ `bool` vorkommen. Die hierbei geltenden Regeln entsprechen im Wesentlichen denen von C/C++:

- Werte der Typen `bool` (in gemischten Ausdrücken), `sbyte`, `char`, `uchar`, `byte`, `char16_t`, `short` und `ushort` werden generell nach `int` konvertiert, `wchar_t`, falls die interne Darstellung von Unicode-Zeichen nicht 32 Bit breit ist..
- Werte vom Typ `char32_t` und `char8_t` werden in `uint` konvertiert. Gleiches gilt für `wchar_t`, falls eine interne Unicode-Darstellung von 32 Bit verwendet wird (Linux-Version).
- Für die Größe der numerischen Typen gilt
  `int < uint < long < ulong < float < double`.
- Die Typen `intptr` und `uintptr` sind in ihrer Größe plattformabhängig. Auf 64-Bit-Plattformen entsprechen sie er den Typen `long` bzw. `ulong`, sonst den Typen `int` bzw. `uint`.

Die implizite Typumwandlung in numerischen Ausdrücken findet streng genommen nur für Operatoren mit zwei Operanden (Infix-Formen) statt. Dabei betrifft die Umwandlung jeweils die an einer einzelnen Operation beteiligten Operanden. So würde etwa in einem Ausdruck

```
2 * 3u - 1.7
```

zunächst der `int`-Wert 2 für die Multiplikation in den Typ `uint` und anschließend das Ergebnis in den Typ `double` umgewandelt.

Kommen in einem numerischen Ausdruck Objekte als Operanden vor, werden ggf. vorhandene Operatormethoden der Objekte (siehe Abschnitt 10.11.9) für den betreffenden arithmetischen oder Bitoperator gegenüber einer Typumwandlung bevorzugt. Existieren keine solchen Operatoren, wird nach einem Konvertierungsoperator (Abschnitt 10.11.10) in den Zieltyp gesucht.

Für alle anderen nicht numerischen Werte wird in Ausdrücken keine implizite Typumwandlung durchgeführt. Hier ist es Aufgabe des Programmierers, ggf. eine explizite Typumwandlung vorzunehmen.

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 13.2 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 10.11.10.

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.

5.7 *Vordefinierte Variablen und Konstanten*

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 10.11).

Die vordefinierte Variable `__TMDELAY__` steuert, wie oft während der Programmabarbeitung geprüft wird, ob mit installierten Zeitgebern (vgl. Abschnitt 12.1) verbundene Funktionen auszuführen sind. Der voreingestellte Standardwert ist 10.

Darüber hinaus gibt es folgende vordefinierte Konstanten (ab Version 1.5, zuvor waren es ebenfalls 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 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.
- `__PTRSIZE__`: Größe einer Adresse (eines Zeigers) der Laufzeitumgebung in Bytes (`uint`); die Variable hat den Wert 8 auf 64-Bit-Systemen, sonst den Wert 4.

Mit Hilfe dieser Konstanten 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 16 Verwendung der globalen Konstante __OSTYPE__

Beispiel 16 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. umdefiniert werden – siehe hierzu Abschnitt 10.11.

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 arithmetische Operatoren:

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

Tabelle 6 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, sofern der zweite Operand nicht von einem Zeichentyp ist. Ist der Operand von einem Zeichentyp, erfolgt eine Konvertierung in den internen Zeichensatz. In Nicht-UNICODE-Builds gibt es zudem eine Sonderbehandlung von Zeichen des Typs `u8char`. Hier erfolgt keine Umwandlung in den internen Zeichensatz. Dies ermöglicht es, UTF-8-Zeichenketten unabhängig vom internen Zeichensatz zu konstruieren.
- 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 10.11.9.

6.2 *Vergleichsoperatoren*

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

B++ kennt folgende Vergleichsoperatoren:

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

Tabelle 7 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 sein[49].
- 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 verglichen[50], 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 10.11.8)
- Werte des Typs `null` sind kleiner als alle anderen Werte.

[49] 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.
[50] Deshalb ist hier i. a. nur die Verwendung der Operatoren == und != sinnvoll.

6.3 *Logische Operatoren*

Die logischen Operatoren verknüpfen Wahrheitswerte miteinander. Bei der Interpretation von Werten, die nicht vom Typ `bool` sind, als Wahrheitswerte 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 (Werte von Referenztypen) haben den Wahrheitswert *`false`*, falls ihr Wert *`null`* ist, andernfalls *`true`*.

Folgende logische Operatoren sind in B++ definiert:

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

Tabelle 8 Logische Operatoren in B++

Hinweise:

- Der Operator `!` liefert stets ein Resultat vom Typ `bool`, also *`true`* oder *`false`*.
- Ausdrücke mit den zweiwertigen 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 `bool`-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 10.11.7.

6.4 *Bitoperatoren*

Die Bitoperatoren sind nur auf Operanden integraler Typen (siehe Abschnitt 5.3.1) direkt anwendbar. Sie realisieren Operationen auf den einzelnen Bits der Operanden. Darüber hinaus können sie für Objekttypen (Klassen) definiert werden – siehe Abschnitt 10.11.9.

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 9 Bitoperatoren in B++

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++, analog zu den arithmetischen Operatoren (siehe 6.1).

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

Tabelle 10 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 5.4.
  Der einfache Zuweisungsoperator kann für Objekttypen (Klassen) umdefiniert werden, um ein vom Standard abweichendes Verhalten zu erreichen[51] – siehe Abschnitt 10.11.5.
- 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.

[51] Die Möglichkeit des Umdefinierens des Zuweisungsoperators wurde mit Version 1.8 eingeführt.

6.6 *Inkrement- und Dekrement-Operatoren*

Die Inkrement- und Dekrement-Operatoren sind unär. Sie erhöhen (Operator `++`) bzw. vermindern (Operator `--`) ihren Operanden jeweils um den Wert 1. Der Operand muss dabei eine mit einem integralen Typ oder einem Objekttyp initialisierte Variable sein[52].

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
`++x`	Inkrement (Präfix)	`b = ++a`	`a = a+1, b = a`
`x++`	Inkrement (Postfix)	`b =  a++`	`b = a, a = a+1`
`--x`	Dekrement (Präfix)	`b = --a * 7`	`a = a-1, b = a * 7`
`x--`	Dekrement (Postfix)	`b = a-- * 7`	`b = a * 7, a = a-1`

Tabelle 11 Inkrement und Dekrement

Hinweise:

- Die Inkrement- bzw. Dekrement-Operatoren erzeugen kürzeren und schnelleren Bytecode als ihre ‚ausgeschriebenen' Entsprechungen (siehe Tabelle 11). Sie sind deshalb besonders für die Modifikation von Zählvariablen in Schleifen (vgl. Abschnitt 8) geeignet[53].
- 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 10.11.4.

[52] 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.

[53] 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 17 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 10.11.2), 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 (*Propertys*) zu definieren (siehe Abschnitt 10.11.3).

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 18 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. 10.11.1). 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 19 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 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 16 ausführlicher erläutert und sind in nachfolgender Tabelle nur der Vollständigkeit halber aufgeführt.

Operator	Bedeutung	Beispiel
`new`	Objektinstanz aus Klasse erzeugen	`a = new A()`
`delete`	Löschen einer Objektinstanz	`delete a`
`::`	Definition von Methoden, Aufruf von Klassenmethoden, Zugriff auf Elemente von Namensräumen und internen Klassen	`A::A() { ... }` `i = A::getMaxId()`
`->`	Aufruf von Instanzmethoden	`name = a->getName()`

Tabelle 12 Objektbezogene Operatoren in B++

Der Operator `::` dient darüber hinaus der expliziten Referenzierung globaler Symbole innerhalb der Implementierung von Funktionen – siehe hierzu Abschnitt 9.1.

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 16.

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 10.11.6). Die entsprechende Operatorfunktion wird dann aufgerufen, wenn der Operator mit einem ‚echten' Objekt, also nicht mit einer Weak-Referenz, verwendet wird.

6.8 *Rangfolge 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 13 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	() [] . ++ -- →
15	Bereich	::

Tabelle 13 Operatorklassifizierung und -hierarchie in B++

7 Schlüsselwörter und reservierte Bezeichner

Schlüsselwörter sind im syntaktischen Sinne *Terminalsymbole*. Das bedeutet, dass sie bereits bei der lexikalischen 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 *null* 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
`yield`	Steuerung	Rückkehr aus einer Coroutine
`try`	Ausnahmen	Einleitung eines geschützten Blocks
`catch`	Ausnahmen	Beginn der Ausnahmebehandlung
`throw`	Ausnahmen	Auslösen einer Ausnahme
`var`	Deklaration	Deklaration einer Variablen dynamischen (beliebigen) Typs
`const`	Deklaration	Deklaration einer Konstanten, kann als Modifier zusammen mit `var` oder einem Datentyp verwendet werden, bzw. Modifier einer Methodendeklaration; außerdem verwendbar als Modifizierer des Rückgabewerts einer Funktion.

Schlüsselwort	Kategorie	Bedeutung
mutable	Deklaration	Deklaration einer Member-Variablen, die auch bei konstanten Objekten geändert werden darf; kann als Modifier mit `var` oder einem Datentyp verwendet werden.
auto	Deklaration	Deklaration einer statisch typisierten Variablen mit Typfestlegung durch Initialisierung
function	Deklaration	Deklaration eines Funktionsliterals
class	Deklaration	Beginn einer Klassendeklaration
namespace	Deklaration	Beginn der Deklaration eines Namensraumes
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
litexp	Deklaration	Kennzeichnung eines literalen Ausdrucks (ab B++ 1.3)
compexp	Deklaration	Kennzeichnung eines kompilierten literalen Ausdrucks (ab B++ 1.7)
TRON	Debugging	Einschalten Trace-Modus
TROFF	Debugging	Ausschalten Trace-Modus
TRSTEP	Debugging	Einschalten Einzelschrittmodus

Tabelle 14 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.

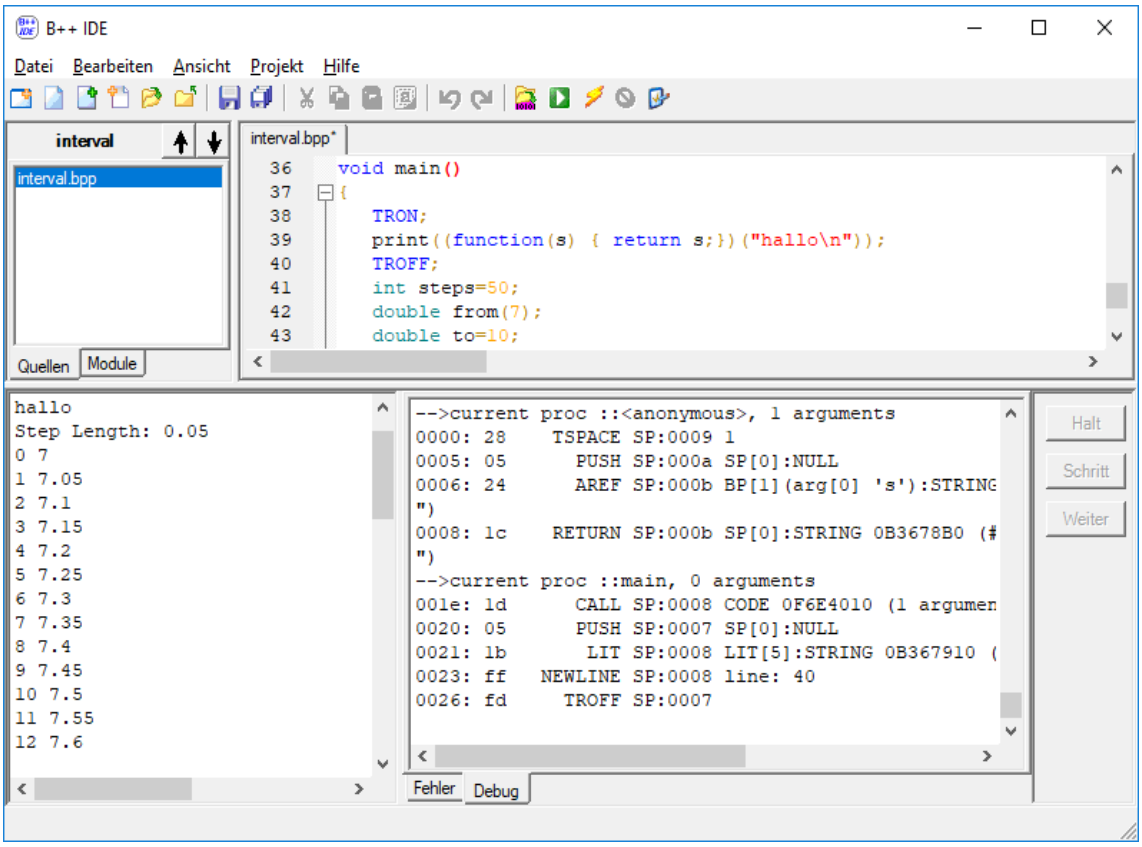


Abbildung 2 Beispiel einer Trace-Ausgabe in der B++-IDE

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

Darüber hinaus 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)
- **`__cloned`** Initialisierungsmethode, die automatisch nach dem Kopieren eines Objekts aufgerufen wird (ab B++ 1.9, siehe 10.3)
- **`defined`** Pseudo-Operator zum Test, ob ein Makro oder 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, OP_ASSIGN`**
  spezielle Methodennamen zur Definition von Operatoren (siehe 10.11)

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 5.3) und die Bezeichner der vordefinierten benannten Literale (vgl. 4.3.8) sowie der vordefinierten Variablen bzw. Konstanten (siehe 5.7).

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 vorgesehen[54] 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 20 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) ist optional. Wird er verwendet, muss er *hinter* den `case`-Zweigen notiert werden.

[54] 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 21 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 22 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 23 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 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*[55] 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 24 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.

[55] Die Ergebnisse der Ausdrücke *init* und *reinit* sind dabei ohne Bedeutung. Semantisch haben sie die Funktion von Anweisungen. Für *init* ist tatsächlich auch eine vollständige Anweisung zulässig. Somit kann z.B. an dieser Stelle direkt eine Laufvariable definiert werden.

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);
   /* intialize table */
   for (i=0; i<rows; ++i)
      for (int j=0; j<cols; ++j)
         array[i][j] = rows * i + j;
   // do anything
   // ...

}
```

Beispiel 25 Bearbeiten mehrdimensionaler Felder mit geschachtelten for-Schleifen

Eine 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
      char 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 26 Schleifensteuerung mit break und continue

8.4 *Ausnahmebehandlung (try – catch – throw)*

Das Konzept der Ausnahmebehandlung 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, "catched\n");
   throw;  // rethrow exception
}
```

Beispiel 27 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 13) sind fester Bestandteil des B++-Laufzeitsystems. Sie sind in jedem Programm verfügbar und können vom Benutzer, d. h. dem Programmierer, normalerweise[56] 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

[[`const`*] 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.

Den in *`parameterlist`* aufgeführten Namen der formalen Parameter kann optional das Schlüsselwort `var` oder ein Datentyp sowie der Modifier `const` (zur Definition eines konstanten Parameters) vorangestellt sein. Ist ein Parameter als konstant definiert, kann die betreffende Funktion auf seinen Wert nur lesend zugreifen. Wird ein konkreter Datentyp angegeben, so ist das jeweilige Argument statisch typisiert.

Vor dem Funktionsnamen kann optional ein Rückgabetyp spezifiziert werden, für den eines der Schlüsselwörter `var`, `null` bzw. `void` oder ein konkreter Typname zulässig ist. 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. Dem Rückgabetyp kann das Schlüsselwort `const` vorangestellt werden (nicht bei `null` oder `void`), um den Rückgabewert als unveränderlich zu kennzeichnen[57].

Die Definition lokaler Variablen kann nicht nur in der Liste im Funktionskopf, sondern auch (und vorzugsweise) 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 Variable 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.

---

[56] Prinzipiell ist es möglich, vordefinierte Funktionen durch benutzerdefinierte zu ersetzen und unter Nutzung von Funktionszeigern (siehe Abschnitt 9.6) sogar zu erweitern.

[57] Sinnvoll ist dies nur bei Rückgabewerten von Referenztypen, da Werttypen ohnehin stets kopiert werden und ihre Änderung so keine Konsequenzen hat.

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. 4.5.6) erfolgen.

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 – es ist dann der Wert, der gerade zuoberst auf dem Stack liegt.
Wird der Rückgabetyp einer Funktion explizit angegeben, so erfolgt nach dem Funktionsaufruf eine Prüfung des Typs des Funktionswertes. Sofern notwendig und möglich wird eine implizite Konvertierung in den geforderten Rückgabetyp vorgenommen[58].

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
uint factorial(uint n) {
   int result = 1;
   for(uint 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 28 Varianten von Funktionsdefinitionen

[58]Die implizite Konvertierung wurde mit Version 1.3 eingeführt; zuvor wurde eine exakte Übereinstimmung verlangt. Die implizite Konvertierung entspricht dem u.a. von C++ gewohnten Verhalten.

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);   // assigns return value to variable
   var g = sqrt(2.0) * 0.5; // uses function in expression
   print(queryName());      // uses function call as argument of another one
}
```

Beispiel 29 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. 9.4). 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:

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

Beispiel 30 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 *Emulation von Referenzparametern*

Die Übergabe von Parametern an Funktionen erfolgt in B++ stets als Wert („*by value*“).

Ist der übergebene Wert selbst von einem Referenztyp (vgl. 5.3.2), spielt dies keine Rolle – der referenzierte Wert kann bei Bedarf durch die jeweilige Funktion modifiziert werden, sofern das Argument nicht als `const` deklariert wurde. Parameter der Werttypen (5.3.1) können dagegen nach ‚außen‘ nicht modifiziert werden, da der Funktion eine Kopie des Wertes übergeben wird.

Ist es notwendig bzw. beabsichtigt, eine Variable von einem Werttyp, die als Parameter übergeben wird, innerhalb einer Funktion zu modifizieren, muss sie in einem Referenztyp gekapselt werden.

Ab Version 1.5 stellt B++ hierfür in der Bibliothek `boxtypes` Box-Klassen (vgl. Referenz) für alle vordefinierten Werttypen sowie eine Box-Klasse für einen beliebigen Werttyp bereit.
Die allgemeine Box-Klasse heißt `B_var`, die spezialisierten tragen den Namen des jeweiligen

Basistyps mit dem Präfix `B_` . Alle diese Klassen haben ein öffentliches Property „`val`“, das den gekapselten Wert repräsentiert, die spezialisierten Formen zudem einen Konvertierungsoperator in den gekapselten Typ.

```
#use "boxtypes.bpm"

void incInt(B_int v) {
   ++ v;
}
void main() {
    vector v = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    auto n = v.size();
    for (uint i=0;i<n;++i) {
         B_int vi(v[i]);
         incInt(vi);
         v[i] = vi.val;
     }
}
```

Beispiel 31 Übergabe von Box-Objekten zur Realisierung von Referenzparametern.

Beispiel 31 demonstriert die Verwendung des Typs `B_int` zur Emulation der Übergabe von Zahlen als Referenzparameter.

Die Bibliothek `boxtypes` kann dafür (wie im Beispiel) mittels #use in vorkompilierter Form eingebunden oder als Quelldatei `boxtypes.bpp` vor der sie verwendenden Quelldatei in den Übersetzungsprozess einbezogen werden.

Mit Version 1.8 wurde die Bibliothek dahingehend erweitert, dass Die Box-Klassen Zuweisungsoperatoren für den gekapselten Typ sowie Operatoren für die jeweils zulässigen arithmetischen Operationen definieren. Somit können die Box-Objekte weitestgehend wie die eingebauten Grundtypen verwendet werden. Einzelheiten sind der Referenzdokumentation zu entnehmen.

9.4 *Variable Parameterlisten und Vorgabeparameter*

Wie im Abschnitt 9.2 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 13.10).
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 32 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 = "";
   uint n = size(vec);
   for (uint 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 33 Optionale Funktionsargumente mit Vorgabeparametern

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

9.5 *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 34.

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

Beispiel 34 Einfache Generatorfunktion mit statischem Zähler

9.6 *Funktionszeiger*

In Abschnitt 5.3.2 wurde der Datentyp `function` erklärt, der einem Wert vom Funktionstyp zugeordnet ist. Tatsächlich lässt sich eine Funktion in gewissen Grenzen wie ein gewöhnlicher 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 Verweis 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 35 Definition und Verwendung eines Funktionszeigers

9.7 *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 36 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 36 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.

9.8 *Anonyme Funktionen und Funktionsliterale*

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 9.4) oder innerhalb der Literalkonstruktoren 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
#literal HIWORD function(x) { return (x >> 16) & 0xFFFF;}
#literal 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 37 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

oder eine Coroutine ist (vgl. 9.9).
Soll eine solche Funktion mehrfach verwendet werden (nur dann ist eine statische Variable sinnvoll), ist 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 38 Anonyme Funktion als Vergleichsfunktion beim Sortieren eines Arrays

In diesem Fall ist dies allerdings nicht notwendig 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 39 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.

9.9 *Coroutinen*

Mit Version 1.6 wurde B++ um ein einfaches Konzept *asymmetrischer Coroutinen* erweitert.

Dabei ist eine Coroutine gewissermaßen eine spezielle[59] Funktion, die dadurch entsteht, dass der Funktionsrumpf mit `yield [retvalue]` statt `return [retvalue]` verlassen wird.

Eine Coroutine wird wie eine gewöhnliche Funktion definiert und aufgerufen. Wird sie mit `yield` verlassen, wird ihr aktueller Zustand (Werte der aktuellen Parameter, lokale Variablen, Verweis auf den nachfolgenden Befehl) zwischengespeichert. Beim nächsten Aufruf wird dann direkt zur gespeicherten Befehlsposition gesprungen, wobei zuvor die lokalen Variablen und ggf. die Argumentwerte restauriert werden. Letztere werden nur dann restauriert, wenn für sie beim erneuten Aufruf keine Werte festgelegt wurden – die gespeicherten Werte werden also wie Vorgabewerte für optionale Argumente behandelt und ersetzen ggf. die ursprünglichen Vorgabewerte.

[59] Häufig werden Coroutinen als Verallgemeinerung einer Funktion oder Prozedur beschrieben, was konzeptionell auch stimmt. Das „speziell“ kommt hier aus Sicht der Implementierung zustande.

Beispiel 40 veranschaulicht das Verhalten einer Coroutine.

```
int cofunc(int i=0) {
   int j = i;
   while (j>0) {
      print (__func__," ",i,"\n");
      yield j--;
   }
   return j;
}
int main() {
   for (int j=0;j=cofunc(j+10);)
      print(j,"\n");
    return 0;
}

/* prints following:
cofunc 10
10
cofunc 20
9
cofunc 19
8
cofunc 18
7
cofunc 17
6
cofunc 16
5
cofunc 15
4
cofunc 14
3
cofunc 13
2
cofunc 12
1
*/
```

Beispiel 40: Rückwärtszähler mit Coroutine

Hier wird ein einfacher Rückwärtszähler implementiert.
Die Coroutine `cofunc` bekommt einen Startwert als Parameter übergeben, der anfangs der lokalen (temporären) Variablen *j* zugewiesen wird.
In der `while`-Schleife werden Funktionsname und Aufrufparameter ausgegeben. Anschließend erfolgt die Rückkehr zum Aufrufer mittels `yield` mit dem um 1 verminderten Wert von *j*. Ist *j==0* erreicht, erfolgt die Rückkehr mit `return`.

Die `main`-Funktion ruft `cofunc` zehnmal mit jeweils inkrementiertem Argument auf.
Wie man an der Ausgabe erkennt, wird in den wiederholten Aufrufen das neu übergebene Argument verwendet, jedoch der zwischengespeicherte Wert der Variablen *j*. Der Einsprung in `cofunc` erfolgt beim wiederholten Aufruf hinter der `yield`-Anweisung, hier also am Ende der `while`-Schleife.

Hinweise:

- Coroutinen können – wie alle Funktionen – auch als Elementfunktionen von Objekten eingesetzt und/oder als Funktionsliteral (siehe 9.8) definiert werden.
- Da eine Coroutine nur einen Platz zur Zwischenspeicherung ihres Zustands besitzt (dieser Speicherplatz verhält sich wie eine versteckte statische Variable, vgl. 9.5) , dürfen Coroutinen ***nicht rekursiv*** aufgerufen werden und sich auch ***nicht gegenseitig aufrufen***.
  Man kann diese Restriktion dadurch umgehen, dass man vor jedem rekursiven Aufruf eine

neue Instanz der Coroutine erzeugt. Dies ist entweder mit Hilfe der Funktion `clone` (siehe 13.10) oder dadurch möglich, die Coroutine als Funktionsliteral zu definieren und dieses Literal vor dem Aufruf an eine Variable zuzuweisen.

- Wird eine Coroutine mit benannten Parametern ohne Vorgabewerte definiert, müssen auch beim wiederholten Aufruf die entsprechenden Parameter angegeben werden, andernfalls tritt ein Compiler-Fehler auf. Dies liegt daran, dass sich für den Compiler der Aufruf einer Coroutine nicht von dem einer gewöhnlichen Funktion unterscheidet und eine Coroutine in B++ keinen eigenständigen Datentyp darstellt.
- Wird eine Coroutine mit `return` (auch implizit am Ende des Rumpfes) verlassen, verliert sie temporär ihre Coroutinen-Eigenschaft, d. h., es wird kein Aufrufzustand zwischengespeichert und ein ggf. erneuter Aufruf entspricht dem einer gewöhnlichen Funktion.
- Der Aufruf eine Coroutine benötigt im Allgemeinen deutlich mehr Zeit als der einer gewöhnlichen Funktion, da bei jedem Aufruf der Zustand gesichert und wiederhergestellt werden muss. Deshalb sollten Coroutinen nur dort verwendet werden, wo sie tatsächlich von Nutzen sind, d.h. zu einer Vereinfachung des Codes führen.

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[60] 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 41 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 Basisklasse ist.

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

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

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

Beispiel 41 Definition von Klassen

[60] Nicht statische Member-Funktionen bzw. -Variablen werden in diesem Text synonym auch als *Elementfunktionen* bzw. *Elementvariablen* bezeichnet.

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.[61]

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 können[62]:

```
class MyDerivedClass : MyBaseClass

{
   string _name;
   var _value;
}
```

Beispiel 42 Minimale Klassendeklaration

Diese Eigenschaft erlaubt es, Klassen bei Bedarf nachträglich um Methoden zu ergänzen (siehe hierzu auch 10.10).

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 Standardargumentwerte bei der Implementierung im Methodenkopf angegeben werden – siehe Beispiel 41.

Konstanten und konstante Methoden

In einer Klasse definierte Konstanten sind implizit statisch, d.h. auf Klassenebene definiert; der Modifier `static` bei der Deklaration ist demnach optional und dient lediglich der Syntaxkompatibilität zu C++. Wie statische Variablen einer Klasse müssen sie in der Klassendeklaration aufgeführt werden. Ihre Initialisierung erfolgt immer zur Übersetzungszeit, woraus sich ergibt, dass eine Elementkonstante stets mit einem Literal oder einem literalen Ausdruck (vgl. 4.3.6) initialisiert werden muss.

[61] Dies entspricht den Regeln von BOB bzw. BOB+, die keine Zugriffsspezifizierer kennen.
[62] Normalerweise 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`).

Konstante Methoden werden wie in C++ mit einem der Argumentliste nachgestellten `const` deklariert. Ebenfalls wie in C++ müssen konstante Methoden stets echte Elementfunktionen (also nicht `static`) sein.
Innerhalb einer solchen Methode wird der `this`-Verweis als konstant angesehen, es können also normalerweise keine Elementvariablen geändert werden.
Eine Ausnahme bilden hierbei wie in C++ Elementvariablen, die mit dem Modifizierer `mutable` deklariert sind.

Hinweis:

Der Aufruf nicht als `const` deklarierter Funktionen ist – anders als in C++ – auch von `const`-Objekten aus möglich. Fehler werden erst bei der Modifikation nicht als `mutable` gekennzeichneter Elementvariablen erzeugt. Dieses Verhalten ist notwendig, um bestehende Methoden aus älteren Versionen (ohne `const` und `mutable`) sowie „native“ Methoden weiter verwenden zu können.

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`).

Objekte werden immer dynamisch erzeugt

Alle Instanzen von Klassen (Objekte) haben den internen Typ `object` und sind damit Referenztypen (siehe 5.3.2). Der einzige Weg sie zu erzeugen führt über den Operator `new`:

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

Dies gilt technisch gesehen auch für statisch typiserte 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 steht[63].

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.

[63] 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.

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 43 zeigt eine entsprechende Abwandlung von Beispiel 41, wobei hier zusätzlich auch Zugriffsspezifizierer verwendet werden.

```
// 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() const { return _ID; }
}

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

Beispiel 43 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 Elementfunktion 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 9.7), 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 44 Konstruktoraufruf in B++

Hinweise:

- Ein Konstruktor lässt sich in B++ bei Bedarf auch wie eine gewöhnliche Elementfunktion aufrufen.
- Von einer Klasse, die keinen eigenen Konstruktor besitzt, können keine Instanzen erzeugt werden. Diese Eigenschaft lässt sich zur Definition *abstrakter Klassen* nutzen[64].

Eine *Sonderform eines Konstruktors* stellt die Methode `__cloned` dar, die mit Version 1.9 eingeführt wurde.
Wenn ein Objekt vollständig kopiert wird, was implizit bei der Zuweisung eines konstanten Objekts an eine Variable oder explizit durch Verwendung der Funktion `clone` (vgl. Abschnitt 13.7) geschieht, so werden normalerweise einfach alle Member-Variablen kopiert. Ein ‚echter' Konstruktoraufruf erfolgt hierbei nicht.
In manchen Fällen kann es erforderlich sein, das kopierte Objekt zusätzlich zu initialisieren. Etwa würden Kopien von Objekten , die von `MyBaseClass` in Beispiel 43 abstammen dieselbe ID besitzen, wie das kopierte Objekt, es gäbe also keine eindeutigen Identifikatoren für solche Objekte mehr.
Dieses Problem lässt sich durch die Bereitstellung einer Methode `__cloned` beheben, die, sofern vorhanden, unmittelbar nach dem Kopieren auf der Kopie aufgerufen wird.

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

Die Klasse MyBaseClass aus Beispiel 43 wird dazu wie folgt modifiziert:

```
// 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() const { return _ID; }
   void __cloned() { _ID = createID(); }
}
```

Beispiel 45 Implementierung der `__cloned`-Methode

Die Methode `__cloned` ist also eine parameterlose Elementfunktion. In Beispiel 45 wird sie verwendet, der Objektkopie eine eigene Identität zuzuweisen.

Hinweis:

Abgeleitete Klassen können die Methode `__cloned` überschreiben. Ein automatischer Aufruf einer geerbten Methode einer Vorfahrklasse erfolgt dann – analog zum Konstruktor – nicht, er kann aber in der Implementierung explizit durchgeführt werden.

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 Destruktoren aus C++. Konkret heißt dies:

- Ein Destruktor wird als parameterlose Elementfunktion 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 Destruktoren aller Basisklassen aufgerufen, d. h., der Destruktor-Code der abgeleiteten Klasse wird immer *vor* dem der Basisklasse ausgeführt. Klassen ohne Destruktor werden dabei innerhalb der Klassenhierarchie übersprungen.

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 46 Destruktoraufrufe in B++

Anmerkungen:

- In Beispiel 46 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 Elementfunktion 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 aufrufen[65].

10.4 *Zugriff auf Elementfunktionen und -variablen innerhalb einer Klasse*

Enthält die Implementierung einer Elementfunktion 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 Elementfunktion 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 47 zeigt.

[65] Achtung: Bis zur Version 1.8 von B++ werden auch hierbei die die Destruktoren der Basisklassen mit aufgerufen! Ab Version 1.9 gilt dies nicht mehr.

```
// 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 47 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 statischen[66] – *virtuell* sind. Betrachten wir dazu nochmals Beispiel 46 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 48 Virtueller Aufruf einer Methode

Hier werden ein und derselben Variablen (`obj`) nacheinander Instanzen der Klassen `Base` bzw. `Derived` zugewiesen[67] 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 Vorfahrmethode aufzurufen, wird dem Funktionsnamen das Präfix **BC_** (für BaseClass) vorangestellt.*

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

[67] 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.

```
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 49 Expliziter Aufruf von Methoden einer Basisklasse

Hinweise:

- Der Aufruf der geerbten Vorfahrmethode muss über den `this`-Zeiger erfolgen (siehe Beispiel 49). 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.

Beispiel 50 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 50 Methodenaufrufe 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 (Elementfunktionen) 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

objref`->(`*classname*`::`*methodname*`)()`

Dabei ist *classname* der Name der eigenen Klasse von *objref* oder einer ihrer Basisklassen, während *methodname* die aufzurufende Methode kennzeichnet[68].

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 Methodenliterale*

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 9.6 – 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 51 Methodenaufruf über Methodenzeiger

Der Zugriff auf eine Methode zur Zuweisung an eine Variable erfolgt syntaktisch wie der 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.

[68] Statt `classname::methodname` kann auch ein anderer Ausdruck stehen, der als Ergebnis eine Referenz auf eine zu *objref* gehörige Elementfunktion liefert.

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

```
function '['classid']' [[const] returntype] '('paramlist
[;localvarlist]')' [const] body
```

verwendet wird (vgl. 9.8).

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 52.

```
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 52 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 *Lokale Klassen*

Ab Version 1.5 erlaubt B++ die „lokale“ Definition von Klassen, d. h. die Definition von Klassen als Element einer anderen, übergeordneten Klasse. Typische Anwendungsfälle hierfür sind die Definition von Iteratoren und Implementierungsdetails benutzerdefinierter Container-Klassen.sowie die Kapselung sonstiger Hilfsklassen zur Implementierung.

Beispiel 53 demonstriert die Verwendung lokaler Klassen anhand der Implementierung einer einfachen Listenklasse mit Vorwärtsiterator und einem internen Elementtyp.

```
class SimpleList
{
   class Element
   {
    public:
       var _value = null;
      Element _next = null;
      Element() {}
   }
   class iterator
   {
   private:
      Element _el = null;
      SimpleList _list = null;
   public:
      iterator(SimpleList list, Element el) {
         _el = el;
         _list = list;
      }
      bool equals(iterator other) {
         if (!other) return false;
         return (addr(_list) == addr(other._list) &&
                 addr(other._el) == addr(_el));
      }
      Element operator*() { return _el; }
      iterator operator++() { if (_el) _el = _el._next; return this; }
      iterator operator++(int) {
         auto res = new iterator(_list,_el);
         return ++res;
      }
      iterator next() { iterator res = clone(this}; return ++res; }
   }
private:
   Element _start = null;
public:
   SimpleList() {};
   iterator begin() { return new iterator(this,_start);}
   iterator end() { return new iterator(this,null); }
   iterator insert(var value, iterator pos = null) {
      auto e = new Element();
      e._value = value;
      auto iter = begin();
      if (!pos || pos == iter) {
         e._next = _start;
         _start = e;
         return begin();
      }
      auto end = end();
      for (auto it=iter; it != end(); ++it;) {
          auto nxt = it->next();
          if (nxt == pos || nxt == end) {
              auto before = *it;
              e._next = before._next;
              before._next = e;
              return nxt;
          }
      }
      return end;
   }
```

```
   iterator erase(iterator pos) {
      auto iter = begin();
      auto end = end();
      if (!_start) return end;
      if (!pos || pos == iter) {
        _start = _start.next;
        return begin();
      }
      for (auto it = iter; it != end; ++it) {
         auto nxt = it->next();
         if (nxt == pos || nxt == end) {
            auto before = *it;
            if (*nxt) {
               before._next = (*nxt)._next;
               (*nxt)._next = null;
            }
            else before._next = null;
            return ++it;
         }
      }
      return end;
   }
}
```

Beispiel 53 Einfach verkettete Liste mit Iterator

Lokale Klassen sind im Gegensatz zu Methoden und Elementvariablen stets öffentlich (`public`).

Der Zugriff auf lokale Klassen durch den Clientcode erfolgt mit dem Bereichsauflösungsoperator (`::`).

10.9 *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` aufruft. 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 definiert[69] – 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 54 Verwenden Partieller Klassen

Übersetzung und Aufruf erfolgen analog:

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

[69] Die Deklaration von doAction hätte man auch weglassen können.

10.10 *Ersetzen von Member-Funktionen (Redefinition)*

So, wie man bestehende gewöhnliche Funktionen durch einfache Neudefinition ersetzen kann (siehe 9.7), 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 Class1::doAction called\n");
}
```

Beispiel 55 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.

10.11 *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 alle anderen Methoden geben auch Operatorfunktionen einen Wert zurück. Deshalb kann eine Operatordefinition optional mit einem Rückgabetyp eingeleitet sein.
- Anders als in C++ können in B++ keine Operatoren als globale Funktionen definiert werden, sondern ausschließlich als (nicht statische) Elementfunktionen von Klassen.
- Wie alle anderen Methoden können Operatorfunktionen als `const` deklariert werden, wenn sie keine Eigenschaften des aufrufenden Objekts ändern (Ausnahme: als `mutable` deklarierte Elementvariablen). Die Argumente von Operatorfunktionen können ebenfalls als `const` deklariert werden.

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

10.11.1 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 von Funktionen gelten. Insbesondere können auch Argumentlisten variabler Länge oder Vorgabeparameter verwendet werden – vgl. Abschnitt 9.4.

Beispiel 56 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 56 Definition des Funktionsoperators für eine Klasse

10.11.2 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 Elementfunktionen – eine für den Lesezugriff und eine für den Schreibzugriff – erforderlich[70]. 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*`)` *[*`const`*]* oder

[returntype] `operator[] (`*index*`)`*[*`const`*]*

für den Lesezugriff bzw.

[returntype] `OP_VSET(`*index*`,` *value*`)` oder

[returntype] `operator[]=(`*index*`,` *value*`)` oder

[returntype] `operator[var](`*index*`,` *value*`)`[71]

[70] Wird nur eine der Funktionen implementiert, funktioniert der Operator nur in einer Richtung.
[71] 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 57 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) const;
   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) const { 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 57 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.

10.11.3 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 Propertys (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*`)` *[*`const`*]* oder

[returntype] `operator.(`*key*`)`*[*`const`*]*

für den Lesezugriff bzw.

[returntype] `OP_PSET(`*key*`,` *value*`)` oder

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

für den Schreibzugriff.

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

Beispiel 58 Verwendet wieder eine Klasse für einen Punkt (diesmal einen zweidimensionalen) zur Darstellung einer möglichen Implementierung[72].

```
class Point2d
{
private:
   double _x;
   double _y;
public:
   Point2d(double x=0, double y=0) { _x=x; _y=y; }
   double get_x() const { return _x; }
   void set_x(double val) { _x = val; }
   double get_y() const { return _y; }
   void set_y(double val) { _y = val; }
   double operator . (string key) const { 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 58 Implementierung des Punktoperators

Hinweis:

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

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.

10.11.4 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.

`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 59 demonstriert die Definition der Operatoren anhand des Ausschnitts einer Klasse für Rationale Zahlen.

```
class Rational
{
private:
   int _num;    // numerator
   uint _denom; // denominator
public:
   Rational(int n=0, uint d=1) { _num=n; _denom=d; }
   Rational operator++() { _num += _denom; return this; }
   Rational operator++(int) {
      return new Rational(_num+_denom,_denom);
   }
   Rational operator--() { _num -= _denom; return this; }
   Rational operator--(int) {
      Rational r(_num,_denom);
      return --r;
   }
   operator string() { return newstring("%d/%u",_num,_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 59 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.

10.11.5 Definieren des Zuweisungsoperators

Für Werte von Referenztypen, wozu auch Objekte gehören, wird im Normalfall einfach eine Referenz an die Zielvariable zugewiesen.

In bestimmten Fällen ist es jedoch wünschenswert, Zuweisungen von Werten anderer Typen an ein Objekt durchführen zu können, in Ergänzung zur Initialisierung im Konstruktor bzw. in Umkehrung von Konvertierungsoperatoren.
Deshalb kann der Zuweisungsoperator ab Version 1.8 umdefiniert werden, wobei das Verhalten dem von C++ ähnelt.

Die Definition des Operators erfolgt in der Form

```
[returntype] OP_ASSIGN([const] value) bzw.
[returntype] operator=([const] value) .
```

Dabei ist *value* der zuzuweisende Wert, dem auch eine Typspezifikation vorangestellt sein kann.

Ist ein benutzerdefinierter Zuweisungsoperator vorhanden, wird er bei jeder Zuweisung an das betreffende Objekt aufgerufen und hat die Möglichkeit, die Zuweisung selbst zu behandeln oder – durch Rückgabe des Wertes *null* – die Standardbehandlung (Einfache Zuweisung an die Zielvariable) zu aktivieren.

Hinweise:

- Die Zuweisung des Wertes *null* oder eines Objekts desselben bzw. eines davon abgeleiteten Typs stellt einen Sonderfall dar. Hier wird der benutzerdefinierte Zuweisungsoperator nicht aufgerufen. Diese Ausnahme ist erforderlich, um die Zuweisung von *null* an die zugehörige Objektvariable und damit ihre Freigabe bzw. das Ersetzen einer Objektinstanz zu ermöglichen.
- Ein benutzerdefinierter Zuweisungsoperator sollte als Ergebnis das Objekt selbst oder den zugewiesenen Wert zurückgeben, um das Standardverhalten des Zuweisungsoperators für einfache Datentypen (Werttypen) nachzubilden.

10.11.6 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 16). Darüber hinaus kann er für Objekte als Präfix-Operator wie folgt explizit definiert werden:

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

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 – vgl. Beispiel 53.

10.11.7 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 15 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 59 könnte etwa so aussehen:

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

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

10.11.8 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([const] [typename] other) [const]
```

und/oder

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

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 postiven 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 Vergleichsergeb-

nis 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 60 erweitert die Klasse `Rational` aus Beispiel 59 um eine `compare`-Methode.

```
class Rational
{
   int _num; // numerator
   uint _denom; // denominator
   Rational(int n=0, uint d=1) { _num=n; _denom=d; }
   // other functions ...
   operator string() const { return newstring("%d/%u",_num,_denom); }
   operator double() const { return ::double(_num) / _denom; }
   int compare(const var other) const {
      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 60 Implementierung einer `compare`-Methode

In dem Beispiel wird auch eine Typkonvertierung nach `double` verwendet (siehe 10.11.10). Zu beachten ist dabei der explizite Aufruf der globalen Konvertierungsfunktion `double` mit Hilfe des Bereichsauflösungsoperators (`::`), wodurch ein rekursiver Aufruf der Elementfunktion `double` verhindert wird.

10.11.9 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 16), 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(*arg*)	OP_ADD_R(*arg*)	operator *[L|R]* + (*arg*)
-	OP_SUB(*arg*)	OP_SUB_R(*arg*)	operator *[L|R]* - (*arg*)
*	OP_MUL(*arg*)	OP_MUL_R(*arg*)	operator *[L|R]* * (*arg*)
/	OP_DIV(*arg*)	OP_DIV_R(*arg*)	operator *[L|R]* / (*arg*)
%	OP_REM(*arg*)	OP_REM_R(*arg*)	operator *[L|R]* % (*arg*)
Bitweise Operatoren			
|	OP_BOR(*arg*)	OP_BOR_R(*arg*)	operator *[L|R]* | (*arg*)
&	OP_BAND(*arg*)	OP_BAND_R(*arg*)	operator *[L|R]* & (*arg*)
^	OP_XOR(*arg*)	OP_XOR_R(*arg*)	operator *[L|R]* ^ (*arg*)
Verschiebeoperatoren			
<<	OP_SHL(*arg*)	OP_SHL_R(*arg*)	operator *[L|R]* << (*arg*)
>>	OP_SHR(*arg*)	OP_SHR_R(*arg*)	operator *[L|R]* >> (*arg*)

Tabelle 16 Definierbare Infix-Operatoren in B++

Das Prinzip der Definition ist für alle diese Operatoren gleich und ist in Beispiel 61 für den Additionsoperator veranschaulicht. Hier wird die Klasse `Point` aus 10.11.2 modifiziert[73] 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() const { return _coords };  // get coordinates
   Point(uint dim);                       // ctor
   Point operator+(const var other) const;          // operator + (left)
   Point operator R +(const var other) const;       // operator + (right)
}

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

Point Point::operator+(const var other) const
{
   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);
```

[73]Die Indexoperatoren sind 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+(const var other) const
{
   return OP_ADD(other);
}
```

Beispiel 61 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 61). 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.

10.11.10 Operatoren zur Typumwandlung

Wie schon in Abschnitt 5.6 erwähnt, können für Objekte (Klassen) Elementfunktionen zur Umwandlung in andere Datentypen definiert werden.
Syntaktisch sind dafür zwei Varianten zulässig:

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

oder

```
operator type().[const]
```

Die erste Form ist eine normale Methodendefinition, wobei der Methodenname (*type*) dem Rückgabetyp entspricht. Optional kann der Rückgabetyp zusätzlich angegeben werden[74]. Die zweite Form entspricht der C++-Notation für Konvertierungsoperatoren.

Konvertierungsmethoden haben grundsätzlich keine Argumente.

In Beispiel 60 (Abschnitt 10.11.8) werden für die Klasse `Rational` Konvertierungsoperatoren in die Typen `double` und `string` definiert.

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([const] [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.

Hinweise:

- 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.
- Eine Klasse, die eine Konvertierung nach `char8_t` definiert, sollte stets auch eine Konvertierung nach `uchar` definieren, da `char8_t` intern ein Alias auf `uchar` ist.

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

11 Namensräume

Das Konzept von Namensräumen, das der Gliederung von Bezeichnern nach Verwendungszweck und der Vermeidung von Bezeichnerkonflikten dient, wird in verschiedenen Programmiersprachen, u.a. C++ und Java, verwendet.

In B++ werden Namensräume ab Version 1.5 unterstützt und syntaktisch analog zu C++ in der Form

```
namespace identifier { … }
```

definiert.

Aus technischer Sicht sind Namensräume in B++ Klassen mit folgenden Einschränkungen:

- Alle Elemente (Variablen, Funktionen) sind Klassenelemente (`static`).
- Es existieren keine Konstruktoren/Destruktoren; somit können keine Instanzen erzeugt werden.
- Alle Elemente sind implizit öffentlich (`public`).
- Es gibt keine Ableitung (Vererbung) von Namensräumen.

Namensräume können untergeordnete Namensräume, Klassendefinitionen, Variablen und Konstanten enthalten. Sie können jedoch nicht Bestandteil einer echten Klasse sein; die Definition eines Namensraumes als untergeordneter Bestandteil einer Klasse ist also (im Gegensatz zu echten Klassen) nicht zulässig.

Die Referenzierung von Elementen eines Namensraums erfolgt wir die Referenzierung statischer Elemente einer Klasse mit Hilfe des Bereichsauflösungsoperators (`::`). Eine using-Klausel gibt es nicht, Elemente eines Namensraumes müssen somit stets explizit angesprochen werden, sofern man sich nicht im Kontext des Namensraums selbst oder eines untergeordneten Namensraums befindet.

12 Synchrone Zeitgeber und angehängte Funktionserweiterungen

Mit Version 2.0 wurden zwei Konzepte eingeführt, Funktionsaufrufe außerhalb des regulären Programmablaufs auszuführen. Solche Funktionsaufrufe können die Programmlogik vereinfachen und so zu verkürzten und schnelleren Programmen beitragen.

12.1 *Synchrone Zeitgeber*

Synchrone Zeitgeber erlauben die einfache oder zyklische Ausführung einer Funktion nach Ablauf einer festgelegten Zeitspanne, wobei hierfür der normale Programmablauf unterbrochen wird.

Sie können mit Hilfe der Funktion `settimer` an beliebiger Stelle installiert und bei Bedarf mit den Funktionen `cleartimer` bzw. `cleartimers` deinstalliert werden (vgl. Abschnitt 13.10).

Die Prüfung auf abgelaufene Zeitgeber und ggf. die Ausführung der zugeordneten Funktionen erfolgt zyklisch während der Programmabarbeitung durch den Bytecode-Interpreter. Um die dadurch möglicherweise entstehende Verlangsamung der Programmabarbeitung zu reduzieren bzw. steuern zu können, lässt sich über die vordefinierte globale Variable `__TMDELAY__` festlegen, aller wie viel Bytcode-Ausführungszyklen eine solche Prüfung vorgenommen wird. Als Standardwert ist hier 10 voreingestellt; je größer der Wert ist, desto seltener werden die Zeitgeber geprüft, was einerseits eine geringere Zeitverzögerung im Ablauf, andererseits aber auch eine geringere Genauigkeit der Zeitgeber zur Folge hat. Sind keine Zeitgeber installiert, ist der Wert von `__TMDELAY__` ohne Bedeutung.

Ausnahmen in Zeitgeberfunktionen

Eine ggf. von einer Zeitgeberfunktion ausgelöste Ausnahme wird grundsätzlich nicht an das ausgeführte Programm weitergeleitet, da der Aufrufkontext der Zeitgeberfunktion unbestimmt ist. Eine solche Ausnahme wird deshalb einfach verworfen. Sofern es sich um einen zyklischen Zeitgeber handelt, wird der betreffende Zeitgeber automatisch deinstalliert, die Funktion dann also nicht erneut aufgerufen.

Hinweise:

- Synchrone Zeitgeber können sich nicht gegenseitig unterbrechen. Während eine Zeitgeberfunktion abgearbeitet wird erfolgt keine Prüfung auf weitere abgelaufene Zeitgeber.
- Es ist möglich, Coroutinen (vgl. 9.9) als Zeitgeberfunktionen zu verwenden, was allerdings nur für zyklische Zeitgeber sinnvoll ist.
- Zeitgeberfunktionen können keine Funktionswerte an das aufrufende Programm übermitteln.

12.2 *Angehängte Funktionserweiterungen*

Angehängte Funktionserweiterungen sind Funktionen, die unmittelbar nach dem Verlassen einer anderen Funktion ausgeführt werden, unabhängig davon, an welcher Stelle und mit welchem Ergebnis die jeweilige Funktion verlassen wird. Dies ermöglicht in bestimmten Konstellationen einen vereinfachten Aufbau des Funktionsablaufs.

Es gibt drei Varianten der Ausführung angehängter Funktionserweiterungen:

1. Ausführung nur, wenn die primäre Funktion keine Ausnahme ausgelöst hat,
2. Ausführung nur, wenn die primäre Funktion eine Ausnahme ausgelöst hat und
3. Ausführung unabhängig davon, ob die primäre Funktion eine Ausnahme ausgelöst hat.

Die Installation angehängter Funktionen erfolgt mit Hilfe der vordefinierten Funktionen `callsucc` (Variante 1), `callfail` (Variente 2) bzw. `callafter` (Variante 3) – siehe Abschnitt 13.10.

Da die Installation an beliebiger Stelle in der Implementierung der primären Funktion erfolgen kann, lässt sie sich selbst in Abhängigkeit von bestimmten Bedingungen realisieren.

Angehängte Funktionserweiterungen und Ausnahmen

Löst eine angehängte Funktionserweiterung eine Ausnahme aus, wird diese Ausnahme an den Aufrufer der primären Funktion weitergeleitet. Hatte die primäre Funktion selbst eine Ausnahme ausgelöst, wird diese ggf. durch die der Funktionserweiterung ersetzt.
Hat die primäre Funktion eine Ausnahme ausgelöst, die Funktionserweiterung jedoch nicht, bleibt die Ausnahme der primären Funktion erhalten. Eine angehängte Funktionserweiterung hat also keine Möglichkeit, eine Ausnahme der primären Funktion zu behandeln.

Hinweise:

- Angehängte Funktionserweiterungen haben keinen Zugriff auf lokale Variablen der primären Funktion und können auch deren Rückgabewert nicht beeinflussen.
- Es ist möglich, für eine Funktion mehrere angehängte Funktionserweiterungen zu installieren. Sie werden dann in der Reihenfolge ihrer Installation ausgeführt. Löst eine Funktionserweiterung eine Ausnahme aus, wird die Ausführung abgebrochen.
- Es ist möglich, Coroutinen (vgl. 9.9) als angehängte Funktionserweiterungen zu verwenden.

13 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* vorgenommen[75]. 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.

13.1 *Standardfunktionen für Container-Datentypen*

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

13.1.1 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 (Zahlenwert wird als Zeichencode interpretiert, Zeichentypen werden in den internen Zeichensatz umgewandelt) sowie vom Typ `buffer`, `charbuffer` oder `string` sein. Argumente anderer Typen werden ignoriert.

Rückgabewert:

Aus den Argumenten erzeugter Zeichenpuffer.

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

Beispiel:

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

Beispiel 62 Erzeugen eines Zeichenpuffers aus mehreren Bestandteilen mit `CBuf`

Funktion newbuffer

`buffer newbuffer(const 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 (`string` oder `charbuffer`) 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(const 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 oder 32 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 bzw. 32 Bit erweitert, die höherwertigen Bytes 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 Schlüsselsortierung, andernfalls in der Reihenfolge des Einfügens zurück gegeben.
- In DOS- und Windows-CE-Builds erfolgt der Zugriff auf Einträge sortierter Dictionarys schneller als auf die unsortierter.

Funktion newvector

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

Die Funktion erzeugt einen neuen Vektor mit reserviertem Speicherplatz für `capacity` Elemente. 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:

Die Kapazität eines Vektors kann nachträglich (wie auch für `buffer` und `charbuffer`) mit Hilfe der Funktion `reserve` (siehe 13.1.2) und seine Größe mit `resize` explizit geändert werden. Eine Vergrößerung ist implizit durch Einfügen von Elementen (Funktion `insert`) 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) insert(vec2,-1,j*j);
// initially empty vector
vector vec3 = newvector();
for (var k=0;k<10;++j) insert(vec3,-1,k*k);
```

Beispiel 63 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 64 Erzeugen eines Vektors aus einer heterogenen Elementliste

13.1.2 Allgemeine Containerfunktionen

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

Funktion bsearch

`uint bsearch(const var` *`cont`*`, const var` *`searchval`*`,`
`const var` *`comparer`* `= NULL)` bzw.

`uint bsearch(const var` *`cont`*`, uint startpos, uint endpos,`
`const var` *`searchval`*`, const var` *`comparer`* `= NULL)`

Die Funktion führt eine Binärsuche nach dem Wert *`searchval`* im sortierten Container *`cont`* (Variante 1) oder im angegebenen Teilbereich davon (Variante 2) durch.

Parameter:

- `cont`: zu durchsuchender Container, zulässige Typen sind `vector`, `buffer` oder `charbuffer`
- `startpos`: Startposition des Suchbereichs
- `endpos`: Endposition des Suchbereichs. Ist `endpos` größer als der Index des letzten Elements von *`cont`*, wird bis zum Ende von *`cont`* gesucht.
- `searchval`: Wert, nach dem in *`cont`* gesucht wird
- `comparer`: Vergleichsfunktion oder Funktor-Objekt der Form

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

  `comparer` wird für den Vergleich zweier Elemente aus *`cont`* aufgerufen und sollte ein negatives Ergebnis für *`left < right`*, ein positives Ergebnis für *`left > right`* oder `0` bei Gleichheit liefern.
  Ist *`comparer`* nicht angegeben, wird ein Standardvergleich der Werte durchgeführt.

Rückgabewert:

Position (Index) des ersten Auftretens von *searchval* im Container bzw. im Suchintervall [*startpos*, *endpos*].
Wird kein solches Element gefunden, ist das Ergebnis NPOS.

Hinweis:

Die Funktion geht davon aus, dass *cont* unter Berücksichtigung des Verhaltens von *comparer* aufsteigend sortiert ist. Trifft dies nicht zu, ist das Ergebnis undefiniert.

Funktion clear

```
void clear(var cont)
```

Die Funktion löscht alle Elemente *aus* *cont*.

Parameter:

- cont: Container, auf den sich die Operation bezieht. Zulässige Typen sind vector, buffer, charbuffer, string oder dictionary.

Funktion erase

```
var erase([const] var cont, uint pos = 0, uint cnt = NPOS)
```

Die Funktion löscht, beginnend bei Index *pos*, höchstens die durch *cnt* gegebene Anzahl von Elementen aus dem Container *cont*.

Parameter:

- cont: Container, aus dem Elemente entfernt werden sollen. Zulässige Typen sind vector, buffer, charbuffer oder string.
- pos: Index des ersten zu entfernenden Elements
- cnt: Anzahl der zu entfernenden Elemente. Ist *cnt* größer als die Anzahl der vorhandenen Elemente ab *pos*, so werden alle Elemente ab Index *pos* entfernt.

Rückgabewert:

Container *cont* nach der Modifikation.

Hinweis:

Ist *cont* eine Konstante, so wird vor dem Löschen eine tiefe Kopie von *cont* angelegt, andernfalls wird *cont* selbst modifiziert.

Funktion find

```
uint find(const var cont, const var searchval, uint pos = 0)
```

Die Funktion durchsucht *cont* linear nach dem Wert *searchval* und liefert die Position von dessen erstem Auftreten als Ergebnis.

Parameter:

- `cont`: zu durchsuchender Container, zulässige Typen sind `vector`, `buffer`, `charbuffer` oder `string`
- `searchval`: Wert, nach dem in *cont* gesucht wird (Elementtyp des Containers). Ist *cont* vom Typ `string`, kann *searchval* entweder ebenfalls vom Typ `string` oder von einem integralen Typ (interpretiert als Zeichencode) sein.
- `pos`: Startposition für die Suche

Rückgabewert:

Position (Index) des ersten Auftretens von *searchval* in *cont* bzw. `NPOS`, 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. 10.11.8) aufgerufen werden.

Funktion foreach

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

Die Funktion iteriert über einen Container und ruft für jedes Element einen Handler (Funktion oder Funktor-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 65 Verwenden von `foreach` zur Manipulation und Ausgabe von Containern

In Beispiel 65 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 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 insert

```
var insert ([const] var dst, uint pos, const var insertion, uint cnt = 1)
```

Die Funktion fügt *cnt*-mal *insertion* an der Indexposition *pos* in den Container *dst* ein.

Parameter:

- dst: Container, in den neue Elemente eingefügt werden sollen, zulässige Typen sind vector, buffer, charbuffer oder string.
- pos: Indexposition in *dst*, an der das Einfügen erfolgt. Ist *pos* größer als der größte aktuelle Index in *dst*, so erfolgt das Einfügen am Ende von *dst*.
- insertion: Einzufügender Wert.
  Ist *dst* vom Typ vector und wird selbst als Wert von *insertion* verwendet, wird vor dem Einfügen eine tiefe Kopie (wie beim Aufruf der Funktion clone) angelegt.
  Sind sowohl *dst* als auch *insertion* vom Typ string und hat *cnt* den Wert 1, so wird *insertion* als Zeichenkette an der Position *pos* in *dst* eingefügt.
  In allen anderen Fällen muss entweder *dst* vom Typ vector oder *insertion* von einem integralen Typ sein.
  Ist *dst* vom Typ string oder charbuffer, so wird *insertion* in den internen Zeichentyp konvertiert.
- cnt: Anzahl der Elemente, die in *dst* eingefügt werden sollen.

Rückgabewert:

Container *dst* nach der Modifikation.

Hinweis:

Ist *dst* eine Konstante, so wird vor dem Einfügen eine tiefe Kopie von *dst* angelegt, andernfalls wird *dst* selbst modifiziert.

Funktion insrange

```
var insrange([const] var dst, uint pos, const var src,
             uint startpos = 0, uint endpos = NPOS)
```

Die Funktion fügt einen Bereich eines Containers in einen anderen Container desselben Typs (oder den selben Container) an der angegebenen Position ein.

Parameter:

- dst: Container, in den Elemente eingefügt werden sollen; zulässige Typen sind vector, buffer, charbuffer oder string.
- pos: Indexposition in *dst*, an der das Einfügen erfolgt. Ist *pos* größer als der größte aktuelle Index in *dst*, so erfolgt das Einfügen am Ende von *dst*.
- src: Container, aus dem ein Bereich kopiert werden soll. Dabei muss der Typ von *src* dem von *dst* entsprechen. Wenn *src* und *dst* den selben Container bezeichnen

und ein Teil des angegebenen Bereichs hinter *pos* liegt, wird vor dem Einfügen eine temporäre (flache) Kopie von *src* angelegt.

- startpos: Startindex des Bereichs der von *src* kopiert werden soll. Ist *startpos* ungültig, d.h. größer als der letzte gültige Index in src, tut die Funktion nichts.
- endpos: Ende-Index (Index des letzten Elements) des Bereichs, der von *src* kopiert werden soll. Ist *endpos* größer als der letzte gültige Index in *src*, wird stattdessen der letzte gültige Index verwendet.
  Ist *endpos* kleiner als *startpos*, erfolgt das Einfügen in umgekehrter Reihenfolge.

Rückgabewert:

Container *dst* nach der Modifikation.

Hinweise:

- Ist *dst* eine Konstante, so wird vor dem Löschen eine tiefe Kopie von *dst* angelegt, andernfalls wird *dst* selbst modifiziert.
- Die Funktion sollte gegenüber insert bevorzugt werden, wenn mehrere Elemente aus einem Container in einen anderen (gleichartigen) übertragen werden sollen, da sie im Zielcontainer nur eine Verschiebeoperation erfordert.

Funktion lbound

```
uint lbound(const var cont, const var searchval,
            const var comparer = NULL)
```
bzw.
```
uint lbound(const var cont, uint startpos, uint endpos,
      const var searchval, const var comparer = NULL)
```

Die Funktion findet die untere Grenze für den Wert *searchval* im sortierten Container *cont* (Variante 1) oder im angegebenen Teilbereich davon (Variante 2).

Parameter:

- cont: zu durchsuchender Container, zulässige Typen sind vector, buffer oder charbuffer
- startpos: Startposition des Suchbereichs
- endpos: Endposition (letzter Index) des Suchbereichs. Ist endpos größer als der Index des letzten Elements von *cont*, wird bis zum Ende von *cont* gesucht.
- searchval: Wert, nach dem in *cont* gesucht wird
- comparer: Vergleichsfunktion oder Funktor-Objekt der Form

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

  comparer wird für den Vergleich zweier Elemente aus *cont* aufgerufen und sollte ein negatives Ergebnis für *left < right*, ein positives Ergebnis für *left > right* oder 0 bei Gleichheit liefern.
  Ist *comparer* nicht angegeben, wird ein Standardvergleich der Werte durchgeführt.

Rückgabewert:

Position (Index) des ersten Elements Container bzw. im Suchintervall [*startpos*, *endpos*], das nicht kleiner als *searchval* ist.
Wird kein solches Element gefunden, ist das Ergebnis die Anzahl der Elemente in *cont* (Variante 1) bzw. *endpos*+1 (Variante 2).

Hinweise:

- Die Funktion geht davon aus, dass *cont* unter Berücksichtigung des Verhaltens von *comparer* aufsteigend sortiert ist. Trifft dies nicht zu, ist das Ergebnis undefiniert.
- Der Standardvergleich von Elementen entspricht der Verwendung der in Abschnitt 6.2 beschriebenen Vergleichsoperatoren und realisiert eine aufsteigende Sortierung.

Funktion move

`var move ([const] var cont, uint srcpos, uint dstpos)`

Die Funktion verschiebt das Element mit dem Index *srcpos* innerhalb des Containers *cont* an die Stelle *dstpos*.

Parameter:

- `cont`: Container, auf den sich die Operation bezieht. Zulässige Typen sind `vector`, `buffer` oder `charbuffer`.
- `srcpos`: Indexposition des zu verschiebenden Elements in *cont*.
- `dstpos`: Indexposition in *const*, an die das Element verschoben werden soll.

Rückgabewert:

Container *cont* nach der Modifikation.

Beispiel:

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

Hinweise:

- Im Falle eines ungültigen Indexwertes in *srcpos* oder *dstpos* löst die Funktion eine Ausnahme vom Typ `string` aus.
- Ist *cont* eine Konstante, so wird vor der Modifikation eine tiefe Kopie von *cont* angelegt, andernfalls wird *cont* selbst modifiziert.

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 der 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 66 Sortieren eines Vektors mit `qsort`

Beispiel 66 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 reserve

```
uint reserve(var cont, uint capacity)
```

Die Funktion verändert die Kapazität eines Containers. Sie versucht, Speicher für *`capacity`* Elemente in *`cont`* bereitzustellen. Dabei werden keine Daten entfernt, d.h., wenn die aktuelle Größe (Anzahl der Elemente) größer als *`capacity`* ist, erfolgt keine Änderung.

Parameter:

- `cont`: Zu modifizierender Container, mögliche Typen sind `vector`, `buffer` oder `charbuffer`.
- `capacity`: Neue Kapazität des Containers (Anzahl von Elementen)

Rückgabewert:

Neue Kapazität des Containers *`cont`*.

Funktion resize

```
bool resize(var cont, uint size, var fill=null)
```

Die Funktion ändert die Größe des Containers *`cont`* auf *`size`* Elemente. Ist die neue Größe kleiner als die bisherige, wird die entsprechende Anzahl von Elementen vom Ende des Containers entfernt. Übersteigt die neue Größe die bisherige, werden leere Elemente am Ende des Containers angefügt.

Parameter:

- `cont`: Zu modifizierender Container, mögliche Typen sind `vector`, `buffer` oder `charbuffer`.
- `size`: Neue Größe des Containers (Anzahl von Elementen)
- `fill`: Wert für die Initialisierung von neuen (angefügten) Elementen. Ist *`cont`* vom Typ `buffer` oder `charbuffer`, muss *`fill`* von einem integralen Typ sein.
  Ist *`fill`* nicht angegeben, wird `null` für Elemente von `vector` und `0` für Elemente von `buffer` bzw. `charbuffer` verwendet.

Rückgabewert:

`true` bei Erfolg, `false`, wenn nicht genügend Speicher verfügbar ist

Funktion reverse

```
var reverse([const] var cont, uint from = 0, uint to = NPOS)
```

Die Funktion kehrt die Reihenfolge der Elemente eines Teilbereichs von *`cont`* um.

Parameter:

- `cont`: Container, zulässige Typen sind `vector`, `buffer` oder `charbuffer`
- `from`: Startposition des Bereichs, in dem die Elementreihenfolge umgekehrt werden soll.
- `to`: Index des letzten Elements im Bereich, dessen Elementreihenfolge umgekehrt werden soll. Ist *`to`* größer als der letzte gültige Index des Containers, wird stattdessen der letzte gültige Index angenommen.

Rückgabewert:

Container nach der Modifikation.

Hinweise:

- Ist *cont* eine Konstante, so wird vor der Modifikation eine tiefe Kopie von *cont* angelegt, andernfalls wird *cont* selbst modifiziert.
- Ist *from* größer als *to*, wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion rfind

```
uint rfind(const var cont, const var searchval, uint pos = NPOS)
```

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

Parameter:

- `cont`: zu durchsuchender Container, zulässige Typen sind `vector`, `buffer` oder `charbuffer`
- `searchval`: Wert, nach dem in *vec* gesucht wird
- `pos`: Startposition für die Suche; ist *pos* größer als der höchstmögliche Index in *vec*, beginnt die Suche am Ende.

Rückgabewert:

Position (Index) des letzten Auftretens von *searchval* in *cont* bzw. `NPOS`, 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. 10.11.8) aufgerufen werden.

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(const 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.

Funktion slice

`var slice (const var cont, uint from = 0, uint cnt = NPOS)`

Die Funktion erzeugt einen neuen Container aus einer Teilfolge der in *`cont`* enthaltenen Elemente. Der Container *`cont`* selbst bleibt dabei unverändert.

Parameter:

- `cont`: Container, aus dem eine Teilfolge extrahiert werden soll. Zulässige Typen sind `vector`, `buffer` und `charbuffer`.
- `from`: Indexposition des ersten Elements in *`cont`*, Bestandteil der Teilfolge ist.
- `cnt`: Anzahl der Elemente aus *`cont`*, die in die Teilfolge übernommen werden sollen. Ist *`cnt`* größer als die Anzahl der vorhandenen Elemente ab *`from`*, so werden alle Elemente ab Index *`from`* in das Ergebnis aufgenommen.

Rückgabewert:

Neuer Container vom selben Typ wie *`cont`* mit der aus *`cont`* gebildeten Teilfolge.

Funktion swap

`var swap ([const] var cont, uint pos1, uint pos2)`

Die Funktion vertauscht die Elemente an den Indexpositionen *`pos1`* und *`pos2`* innerhalb des Containers *`const`*.

Parameter:

- `const`: Container, auf den sich die Operation bezieht. Zulässige Typen sind `vector`, `buffer` und `charbuffer`.
- `pos1`: Indexposition des ersten Elements
- `pos2`: Indexposition des zweiten Elements

Rückgabewert:

Container *const* nach der Modifikation.

Beispiel:

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

Hinweise:

- Im Falle eines ungültigen Indexwertes für *pos1* oder *pos2* wird eine Ausnahme vom `string`-Typ ausgelöst.
- Ist *cont* eine Konstante, so wird vor der Modifikation eine tiefe Kopie von *cont* angelegt, andernfalls wird *cont* selbst modifiziert.

Funktion ubound

```
uint ubound(const var cont, const var searchval,
            const var comparer = NULL)
```
bzw.
```
uint ubound(const var cont, uint startpos, uint endpos,
            const var searchval, const var comparer = NULL)
```

Die Funktion findet die obere Grenze für den Wert *searchval* im sortierten Container *cont* (Variante 1) oder im angegebenen Teilbereich davon (Variante 2).

Parameter:

- `cont`: zu durchsuchender Container, zulässige Typen sind `vector`, `buffer` oder `charbuffer`
- `startpos`: Startposition des Suchbereichs
- `endpos`: Endposition des Suchbereichs. Ist `endpos` größer als der Index des letzten Elements von *cont*, wird bis zum Ende von *cont* gesucht.
- `searchval`: Wert, nach dem in *cont* gesucht wird
- `comparer`: Vergleichsfunktion oder Funktor-Objekt der Form

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

  comparer wird für den Vergleich zweier Elemente aus *cont* aufgerufen und sollte ein negatives Ergebnis für *left < right*, ein positives Ergebnis für *left > right* oder `0` bei Gleichheit liefern.
  Ist *comparer* nicht angegeben, wird ein Standardvergleich der Werte durchgeführt.

Rückgabewert:

Position (Index) des ersten Elements Container bzw. im Suchintervall [*startpos*,*endpos*], das größer als *searchval* ist.
Wird kein solches Element gefunden, ist das Ergebnis die Anzahl der Elemente in *cont* (Variante 1) bzw. *endpos*+1 (Variante 2).

Hinweise:

- Die Funktion geht davon aus, dass *cont* unter Berücksichtigung des Verhaltens von *comparer* aufsteigend sortiert ist. Trifft dies nicht zu, ist das Ergebnis undefiniert.
- Der Standardvergleich von Elementen entspricht der Verwendung der in Abschnitt 6.2 beschriebenen Vergleichsoperatoren und realisiert eine aufsteigende Sortierung.

13.1.3 Vektor-Funktionen

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

Ab der Version 1.4 von B++ gibt es für alle diese Funktion verallgemeinerte Varianten ohne Präfix „v“ – siehe Abschnitt 13.1.2. Die Funktionen dieses Abschnitts sind nun größtenteils nur noch Alias-Bezeichner für die allgemeinen Funktionen und sollten in neuen Programmen nicht mehr verwendet werden.

Funktion vclear

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

Die Funktion löscht alle Elemente des Vektors *vec*.

Parameter:

- `vec`: Vektor

Funktion vecsize

```
uint vecsize(const 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, uint cnt = NPOS)
```

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`* größer als die Anzahl der vorhandenen Elemente ab *`pos`*, so werden alle Elemente ab Index *`pos`* entfernt.

Rückgabewert:

Vektor *`vec`* nach der Modifikation.

Funktion vinsert

`vector vinsert (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 (const vector vec, uint from = 0, uint cnt = NPOS)
```

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* 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.

13.1.4 Dictionary-Funktionen

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

Funktion dictclear

`void dictclear (dictionary dict)`

Die Funktion löscht alle Elemente des assoziativen Containers *`dict`*.

Parameter:

- `dict`: Dictionary

Funktion dictcontains

`bool dictcontains (const dictionary dict, const 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`, falls ein Eintrag mit dem Schlüssel *`key`* in *`dict`* enthalten ist, andernfalls `false`.

Funktion dicterase

`bool dicterase (dictionary dict, const 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`, falls ein Eintrag mit dem Schlüssel *`key`* aus *`dict`* entfernt wurde (darin enthalten war), andernfalls `false`.

Funktion dictgetkeys

`vector dictgetkeys (const 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 67.
- Die Reihenfolge der Elemente des Ergebnisvektors hängt davon ab, ob der Container sortiert ist oder nicht (vgl. Funktion newdict in Abschnitt 13.1.1). 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 67 Verwenden von dictgetkeys zum Iterieren durch ein Dictionary

Funktion dictgetvalues

```
vector dictgetvalues (dictionary dict)
const vector dictgetvalues (const dictionary dict)
```

Die Funktion liefert einen Vektor mit den Werten aller in *dict* enthaltenen Elemente.
Ist *dict* konstant, ist auch das Ergebnis konstant.

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 68 Verwenden von dictgetvalues

13.2 *Typkonvertierungen und RTTI*

B++ kennt keinen cast-Operator. Deshalb werden explizite Typkonvertierungen (vgl. Abschnitt 5.6) mit Hilfe von Funktionen durchgeführt, wobei der Funktionsname jeweils dem Bezeichner des Zieltyps entspricht.

B++ 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) Typumwandlungsoperator (vgl. 10.11.10) weitergeleitet.

Ferner stellt B++ einige Funktionen zur Ermittlung von Typinformationen zur Laufzeit (RTTI) bereit.

13.2.1 Funktionen zur Typkonvertierung

Funktion bool

`bool bool(const var` *`expr`*`)`

Die Funktion bildet aus einem Wert beliebigen Typs einen booleschen Wert.

Parameter:

- `expr`: Ausdruck beliebigen Ergebnistyps

Rückgabewert:

> `false` für alle numerischen Werte vom Betrag Null, den Wert **null** und alle Referenztypen mit dem Datenwert NULL (also ohne gültigen Wert);
> sonst `true`.

Hinweis:

Falls *`expr`* ein Objekt repräsentiert, dessen Klasse einen `bool()`-Operator definiert, so wird dieser Operator aufgerufen. Existiert kein solcher Operator, jedoch ein `int`()-Operator, so wird dieser Operator aufgerufen und das Resultat in den Typ `bool` konvertiert.

Funktionen zur Konvertierung in integrale Typen

Numerische Typen:

`sbyte sbyte(const var expr)`

`byte byte(const var expr)`

`short short(const var expr)`

`ushort ushort(const var expr)`

`int int(const var expr)`

`uint uint(const var expr)`

`long long(const var expr)`

`ulong ulong(const var expr)`

`intptr intptr(const var expr)`

`uintptr uintptr(const var expr)`

Zeichentypen:

`char char(const var expr)`

`uchar uchar(const var expr)`

`char8_t char8_t(const var expr)`

`char16_t char16_t(const var expr)`

`wchar_t wchar_t(const var expr)`

`char32_t char32_t(const var expr)`

`u8char u8char(const var expr)`

Alle hier aufgeführten Funktionen konvertieren einen Wert in den entsprechenden integralen Typ. Das Verhalten dieser Funktionen ist bis auf den Rückgabetyp identisch, weshalb sie hier gemeinsam behandelt werden.

Parameter:

- `expr`: Zu konvertierender Wert. Dabei kann *`expr`* von einem beliebigen numerischen Typ, einem Zeichentyp, 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(const var expr)`

`float float(const var expr)`

Die hier aufgeführten Funktionen konvertieren einen Wert in den Typ `double` bzw. `float`. Das Verhalten dieser Funktionen ist bis auf den Rückgabetyp 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(const 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 Ergebnispuffers 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(const 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` oder `uintptr`), 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* von Typ `intptr` oder `uintptr`, so ist das Ergebnis eine hexadezimale Adressdarstellung (entsprechend der C-Funktion `printf` mit dem Formatflag `%p`).
- Hat *expr* einen Zeichentyp, so erfolgt eine Konvertierung in den internen Zeichentyp, und es wird ein Zeichenpuffer mit einem Element erzeugt.
- 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(const 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(const 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 Zeichen, ein numerischer Wert oder ein Objekt sein.

Rückgabewert:

Ergebnis der Konvertierung von *`expr`*.

Hinweise:

- Hat *`expr`* einen numerischen Typ (außer `intptr` bzw. `uintptr`), 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.

- Hat *expr* einen Zeichentyp, so erfolgt eine Konvertierung in den internen Zeichentyp, und es wird eine Zeichenkette aus einem Zeichen erzeugt.
- Ist *expr* von Typ `intptr` oder `uintptr`, 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()`[76] zur Konvertierung definieren.
- In Falle eines unzulässigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion str2utf8buf

```
buffer str2utf8buf(const string src, bool raw = false)
```

Die Funktion wandelt den Inhalt einer Zeichenkette in den UTF-8-Zeichensatz um und gibt das Resutat als Puffer zurück.

Parameter:

- `src`: Zeichenkette, sie nach UTF-8 konvertiert werden soll.
- `raw`: Wenn das Flag auf *true* gesetzt wird, erfolgt keine Konvertierung aus dem internen Zeichensatz, der Inhalt von *src* wird also einfach in das Ergebnis übernommen.
  Der Wert von *raw* wird in Unicode-Builds ignoriert.

Rückgabewert:

Puffer mit der UTF-8-Darstellung von src, abgeschlossen mit einem Null-Byte.

Hinweis:

Für *src* kann ein Wert beliebigen Typs übergeben werden, sofern eine implizite Konvertierung nach string möglich ist.

Funktion utf8buf2str

```
string utf8buf2str(const buffer src, bool raw = false)
```

Die Funktion wandelt den Inhalt einers Puffers mit einer UTF-8-Zeichenkette in eine Zeichenkette im internen Zeichensatz um.

Parameter:

- `src` Puffer mit einer UTF-8-Zeichenkette, die konvertiert werden soll.

[76] Dies hat historische Gründe. In neuem Code sollte der Operator bevorzugt werden.

- `raw`: Wenn das Flag auf *`true`* gesetzt wird, erfolgt keine Konvertierung in den internen Zeichensatz, der Inhalt von *`src`* wird also einfach in das Ergebnis übernommen.
  Der Wert von *`raw`* wird in Unicode-Builds ignoriert.

Rückgabewert:

Zeichenkette in der internen Darstellung.

Hinweis:

Die Funktion prüft nicht, ob der Inhalt von *`src`* eine korrekte UTF-8-Zeichenkette enthält. Trifft dies nicht zu, ist das Ergebnis undefiniert.

Funktion vector:

`vector vector(const 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 `byte` abgebildet.
- Ist *`expr`* vom Typ `charbuffer`, so wird jedes im Argument enthaltene Zeichen auf ein Element des Ergebnisvektors vom internen Zeichentyp 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. 13.1.4).
- 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.

13.2.2 RTTI-Funktionen

Funktion dynamic_cast:

`object dynamic_cast(const Class dest, const var expr)`

`object dynamic_cast(const string classname, const 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 69 Verwenden von dynamic_cast

Funktion findmethod

```
function findmethod(const var objectOrClass, const string name,
                    bool isstatic = false)
```

Die Funktion sucht in *objectOrClass* nach einer Methode *name*, die im aktuellen Kontext aufgerufen werden kann.

Parameter:

- `objectOrClass`: Objektinstanz oder Klasse, die zur Suche nach der Methode verwendet wird. Es kann hier auch eine Zeichenkette angegeben werden, die einen Klassennamen bezeichnet.
- `name`: Name der Methode, nach der gesucht werden soll.
- `isstatic`: Flag, das angibt, ob nach einer statischen Methode (*`true`*) oder einer Elementfunktion (*`false`*) gesucht werden soll.

Rückgabewert:

Funktions- bzw. Methodenzeiger bei Erfolg, sonst `null`.

Hinweise:

- Da B++ keine Mehrfachvererbung und kein Interface-Konzept unterstützt, kann diese Funktion benutzt werden, um zu überprüfen, ob ein bestimmtes Objekt eine erwartete Funktion implementiert.
- Das Verhalten der Funktion hängt vom Aufrufkontext ab: Wird sie innerhalb einer Elementfunktion eines Objekts aufgerufen und *`objectOrClass`* verweist auf eine Basisklasse der Klasse dieses Objekts, werden auch geschützte Methoden (protected) gefunden. Verweist *`objectOrClass`* auf dieselbe Klasse wie die des Aufruferobjekts, werden zusätzlich private Methoden gefunden. Trifft keine dieser Bedingungen zu, muss *`name`* auf eine öffentliche Methode (public) verweisen.

Funktion getclass:

`Class getclass(const var` *`arg`*`)`

Die Funktion liefert die Klasse des in *`arg`* übergebenen Objekts bzw. findet die Klasse zum in *`arg`* angegebenen (qualifizierten) Klassennamen.

Parameter:

- `arg`: Ausdruck des Typs `Class, object` oder `string`

Rückgabewert:

zu *`arg`* gehörende Klasse

Hinweise:

- Ist *`arg`* nicht von einem gültigen Typ, wird eine Ausnahme von `string`-Typ ausgelöst.
- Ist *`arg`* vom Typ `string` und es wurde keine entsprechende Klasse gefunden, so ist das Ergebnis `null`.
- Ist *`arg`* vom Typ `Class`, wird einfach das Argument zurückgegeben.

Funktion getclassname:

```
string getclassname(const var expr, bool qualified = false)
```

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`
- `qualified`: Flag, das festlegt, ob im Fall innerer Klassen der vollständig qualifizierte Name oder nur der einfache Klassenname (Standard) zurück gegeben werden soll.

Rückgabewert:

Klassenname von *expr* bei Erfolg, sonst **`null`**

Funktion getsymbol

```
var getsymbol(const 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, anderseits 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 70 Illustriert die Verwendung von `getsymbol` in einer Funktion `mkDir`, die ein Verzeichnis anlegt. Weil B++ hierfür keine vordefinierte Funktion bereitstellt, werden systemspezifische Funktionen (unter MS-DOS ein Interruptaufruf und unter Windows die API-Funktion

`CreateDirectory`, deren Adresse mit `GetKernel32Func` bestimmt wird) verwendet. Im Beispiel wird zwischen den Betriebssystemen mit Hilfe der vordefinierten Konstante `__OSTYPE__` (siehe Abschnitt 5.7) unterschieden. Die Funktion `getsymbol` wird verwendet, um zur Laufzeit in Abhängigkeit von der Plattform, auf der das Programm ausgeführt wird, die jeweiligen Systemfunktionen zu referenzieren. Somit ist es auch als vorübersetzter Bytecode, also ohne Neuübersetzung, auf verschiedenen Plattformen ablauffähig.

```
// Creates a new directory
bool mkDir(const string path)
{
   intptr adr = addr(path);
   if (__OSTYPE__ == "DOS")
   {
      function segread = getsymbol("segread");
      function getreg = getsymbol("getreg");
      function setreg = getsymbol("setreg");
      function int86x = getsymbol("int86x");
      segread();
      setreg("ah",0x39);
      setreg("ds",adr >> 16);
      setreg("dx",adr &6 0xffff);
      int86x(0x21);
      return !getreg("cflag");
   }
   else if (__OSTYPE__ == "WIN32")
   {
      function GetFunc = getsymbol("GetKernel32Func");
      intptr pfunc = GetFunc("CreateDirectory?");
      return CallLibFunc(pfunc,adr,0);
   }
   // Fallback for other systems
   return system("mkdir "+path) ==0;
}
```

Beispiel 70 Anwendung von `getsymbol` für die Benutzung plattformspezifischer Funktionen

Funktion gettype:

`int gettype (const 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 17 listet die möglichen Werte auf:

Type-ID	Typname (Ergebnis)	Datentyp
0	NULL	null
1	VOID	void
2	AUTO	auto
64	SBYTE	sbyte
65	BOOL	bool
66	CHAR	char
68	BYTE	byte
70	UCHAR	uchar
71	U8CHAR	u8char
72	SHORT	short
76	USHORT	ushort
78	CHAR16	char16_t
79	WCHAR	wchar_t
80	INT	int
81	INTPTR	intptr
84	UINT	uint
85	UINTPTR	uintptr
86	CHAR32	char32_t
88	LONG	long
92	ULONG	ulong
112	FLOAT	float
120	DOUBLE	double
128	FILE	FILE
129	CODE	‚native' Funktion
130	VAR	Variable, Dictionary-Eintrag (intern)
160	OBJECT	object
161	STRONGOBJECT	streng typisiertes Objekt

Type-ID	Typname (Ergebnis)	Datentyp
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 17 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. 22).
- Die ID eines Datentyps kann mit der Funktion `gettype` ermittelt werden.

Funktiom isconst:

```
bool isconst(var expr)
```

Die Funktion prüft, ob der durch *expr* spezifizierte Wert eine Konstante ist.

Parameter:

- *expr*: beliebiger Ausdruck, dessen Ergebnis geprüft werden soll; typischerweise eine Variable oder das Ergebnis eines Funktionsaufrufs.

Rückgabewert:

true, wenn *expr* eine Konstante ist

Funktiom mkconst:

```
bool mkconst(var expr, bool setconst = true)
```

Die Funktion modifiziert die Konstanteneigenschaft des Ergebnisses von *expr*.

Parameter:

- *expr*: beliebiger Ausdruck, dessen Ergebnis modifiziert werden soll; typischerweise eine Variable oder das Ergebnis eines Funktionsaufrufs.
- *setconst*: neuer Zustand der Konstanteneigenschaft

Rückgabewert:

Wert von *expr* mit geänderter Konstanteneigenschaft

Hinweis:

Die Funktion modifiziert niemals den Zustand einer durch *expr* spezifizierten Variablen. Das Ergebnis ist entweder eine Kopie des Wertes von expr (in Falle eines eines einfachen Wertes) oder eine Referenz (bei Werten vom Referenztyp) mit modifizierter Konstanteneigenschaft. Sie ist in der Verwendung mit `const_cast` in C++ vergleichbar.

Funktion newobject:

`object newobject(Class classtype, ...)`

Die Funktion erzeugt ein neues Objekt (eine Instanz) der in *classtype* spezifizierten Klasse. Sie verhält sich wie der `new`-Operator, erlaubt aber die dynamische Bestimmung des Klassentyps des zu erzeugenden Objekts. Sie ist als Hilfsfunktion zur generischen Objekterzeugung zur Laufzeit gedacht.

Parameter:

- `classtype`: Klasse, die zur Objekterzeugung verwendet werden soll.
- `...`: Liste der Argumentwerte, die an den Konstruktor des erzeugten Objekts übergeben werden.

Rückgabewert:

erzeugtes Objekt

Hinweis:

Im Fall eines ungültigen Argumenttyps oder des Wertes `null` für *classtype* wird eine Ausnahme vom `string`-Typ ausgelöst.

In Beispiel 71 wird die typische Verwendung der Funktionen `newobject` und `getclass` zum Aufbau einer allgemeinen Factory-Funktion gezeigt.

```
object factoryFunc(string classname, ...) {
        vector args = argv();
        // replace class name by class reference in arguments
        args[0] = getclass(classname);
        return call(newobject,args);
}
```

Beispiel 71 Generische Objekterzeugung mit den Funktionen `getclass` und `newobject`

Diese Funktion ersetzt das erste Argument (den Klassennamen) der variablen Argumentliste durch des Resultat des Aufrufs von `getclass` und ruft `newobject` anschließend indirekt mit der Funktion `call` auf, um die variable Argumentliste weitergeben zu können.

13.3 *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 fflush:

`int fflush(FILE fp)`

Die Funktion leert den Puffer der durch *fp* angegebenen geöffneten Datei, d.h. sie synchronisiert den zu *fp* gehörenden internen Puffer mit dem tatsächlichen Dateiinhalt. Das Verhalten entspricht genau dem der gleichnamigen C-Funktion.

Parameter:

- `fp`: Handle der Datei; hat *`fp`* den Wert `null`, bezieht sich der Aufruf auf alle geöffneten Dateien.

Rückgabewert:

0 bei Erfolg, sonst verschieden von 0

Hinweis:

Die Funktion sollte beim Wechsel zwischen Lese- und Schreibzugriffen auf ein und dieselbe geöffnete Datei verwendet werden. Beim Schließen einer Datei mit `fclose` wird der zugehörige Puffer automatisch geleert.

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(const string` *`filename`*`, const 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(const 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(const 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 freopen:

`FILE freopen(const string filename, const string mode, FILE fp)`

Die Funktion versucht, einen bereits geöffneten File-Stream (ein FILE-Handle) erneut mit einer anderen Datei oder in einem anderen Modus zu öffnen.
Hierzu wird *`fp`* zunächst geschlossen und anschließend mit *`filename`* und *`mode`* neu geöffnet. Das Verhalten entspricht dem 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+).

- `fp`: Handle der neu zu öffnenden Datei

Rückgabewert:

`FILE`-Handle bei Erfolg, sonst `null`

Hinweis:

Mit dieser Funktion geöffnete Dateien sollten stets mit `fclose` geschlossen werden.

Funktion fseek:

```
int fseek(FILE fp, int offset, int origin)
```

Die Funktion setzt den internen Datenzeiger einer geöffneten Datei auf eine bestimmte Position. Auf die so gesetzte Position beziehen sich nachfolgende Dateizugriffe.
Das Verhalten entspricht dem der gleichnamigen C-Funktion.

Parameter:

- `fp`: Handle einer geöffneten Datei
- `offset`: Abstand in Bytes vom durch *`origin`* gegebenen Ausgangspunkt
- `origin`: Ausgangspunkt der Operation; hierfür muss einer der folgenden Werte verwendet werden:
  - SEEK_SET (0): Dateianfang / absolute Position
  - SEEK_CUR (1): Positionierung relativ zur aktuellen Position
  - SEEK_END (2): Positionierung relativ zum aktuellen Dateiende

Rückgabewert:

Bei Erfolg wird 0, sonst ein von 0 verschiedener Wert zurückgegeben..

Hinweis:

Im Allgemeinen ist die Verwendung der Funktion nur mit Binärdateien sinnvoll. Für Textdateien ist die Bedeutung von *`offset`* und somit das Verhalten nicht allgemein definiert.

Funktion ftell:

```
int ftell(FILE fp)
```

Die Funktion liefert die aktuelle Position des Datenzeigers innerhalb einer geöffneten Datei..
Das Verhalten entspricht dem der gleichnamigen C-Funktion.

Parameter:

- `fp`: Handle einer geöffneten Datei

Rückgabewert:

Position bezogen auf den Dateianfang von *`fp`*

Funktion fwrite

```
uint fwrite(const buffer buf, FILE fp)
uint fwrite(const buffer buf, uint cnt, FILE fp)
uint fwrite(const 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:

```
bool fwriteval(const 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 true, sonst false

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 72 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 5.4) und zur Implementierung des Punkt-Operators (vgl. 10.11.3) 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 72 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 zurückgegeben.

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 kbhit:

```
bool kbhit()
```

Die Funktion prüft, ob eine Taste auf der Tastatur gedrückt wurde, die einen Zeichencode liefert.

Rückgabewert:

`true`, wenn eine Taste gedrückt wurde

Hinweis:

Das Verhalten der Funktion weist in Abhängigkeit von der Ausführungsplattform Unterschiede auf:

- Unter MS-DOS und in Windows-Konsolenanwendungen verhält sich `kbhit` wie die gleichnamige C-Funktion. Ist das Ergebnis *`true`*, so kann über `stdin` mindestens ein Zeichen abgerufen werden.
- In Windows-Anwendungen ohne Konsole wird das Verhalten mit Hilfe der Windows-Funktion `GetAsyncKeyState` emuliert. Dabei ist zu beachten, dass ein Rückgabewert *`true`* hier nicht gleichbedeutend damit ist, dass auf `stdin` ein Zeichen verfügbar wäre.
- Auf UNIX-Plattformen wird das Verhalten von `kbhit` ebenfalls emuliert. Die Funktion verhält sich hier wie bei einer Windows-Konsolenanwendung, falls der aktuelle Prozess mit einem Terminalfenster verbunden ist, andernfalls ist das Verhalten undefiniert.

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 Dezimalzahlen ausgegeben.
- Werte der Typen `double` und `float` werden als unformatierte Double-Werte (wie mit dem Formatflag `%g` der C-Funktion `printf`) ausgegeben.

- Werte der Typen `intptr` und `uintptr` 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.

13.4 *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(const 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. 13.2.1) 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, char 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(const 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 der Typen `intptr/uintptr`), Zeichen oder Zeichenketten sein. Für Werte anderer Typen (Referenztypen) wird die Adresse des zugehörigen Datenbereichs (Ergebnis der Funktion `addr()`) übergeben, was zur Ausgabe der Adresse des Datenbereichs von Referenztypen verwendet werden kann. Es erfolgt grundsätzlich keine implizite Typkonvertierung.

Funktion strcmp:

```
int strcmp(const string s1, const 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([const] string str, uint pos, uint cnt = NPOS)
```

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* größer als die Anzahl der hinter *pos* verbleibenden Zeichen, wird der gesamte Inhalt ab *pos* bis zum Ende von *str* gelöscht.
- Ist *str* eine Konstante, so wird vor dem Löschen eine Kopie von *str* angelegt, andernfalls wird *str* selbst modifiziert.

Funktion strfind

```
uint strfind(const string str, const string search, uint start = 0)
uint strfind(const 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 `NPOS`.

Funktion stricmp:

```
int stricmp(const string s1, const 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 ([const] string str, uint pos, const string insstr)
string strinsert ([const] 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 Einfügen.

Hinweise:

- Ist *pos* größer als der Index des letzten Zeichens von *str*, so erfolgt ein Anfügen an das Ende von *str*.
- Ist *str* eine Konstante, so wird vor dem Einfügen eine Kopie von *str* angelegt, wird *str* selbst modifiziert.

Funktion strlen:

```
uint strlen(const 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([const] string s)
```

Die Funktion konvertiert alle Großbuchstaben in *s* in Kleinbuchstaben.

Parameter:

- `s`: Zeichenkette

Rückgabewert:

- Modifizierte Zeichenkette nach der Umwandlung

Hinweise:

- Das Verhalten der Funktion hängt von den aktuellen Lokalisierungseinstellungen ab, die bei Bedarf mit Hilfe der Funktion `setlocale` eingestellt werden können.
- Ist *`s`* eine Konstante, so wird vor der Ersetzung eine Kopie von *`s`* angelegt, andernfalls wird *`s`* selbst modifiziert.

Funktion strreplace

```
string strreplace([const] string str, const string search,
                  const string replacement)
string strreplace([const] 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:

Ist *`str`* eine Konstante, so wird vor der Ersetzung eine Kopie von *`str`* angelegt, andernfalls wird *`str`* selbst modifiziert.

Funktion strsplit:

```
vector strsplit(const string source, const string delimiter,
 bool storeEmptyTokens = true, bool storeDelimitChar = 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.
- `storeDelimitChar`: 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 *`storeDelimitChar`* 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 73 Verwenden von `strsplit`

Funktion strupr:

```
string strupr([const] string s)
```

Die Funktion konvertiert alle Kleinbuchstaben in *`s`* in Großbuchstaben.

Parameter:

- `s`: Zeichenkette

Rückgabewert:

Modifizierte Zeichenkette nach der Umwandlung

Hinweise:

- Das Verhalten der Funktion hängt von den aktuellen Lokalisierungseinstellungen ab, die bei Bedarf mit Hilfe der Funktion `setlocale` eingestellt werden können.
- Ist *`s`* eine Konstante, so wird vor der Ersetzung eine Kopie von *`s`* angelegt, andernfalls wird *`s`* selbst modifiziert.

Funktion substr

```
string substr(const string str, uint start, uint len = NPOS)
```

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`* 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.

13.5 *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`.

Funktionen gmtime und localtime:

```
vector gmtime(long t)
vector localtime(long t)
```

Die Funktionen erzeugen aus der numerischen Datums- und Zeitangabe in *t* einen Vektor mit den Bestandteilen der Zeitangabe in GMT (`gmtime`) oder lokalisiert (`localtime`).

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 mktime:

```
long mktime(vector tm)
long mktime(const vector tm)
```

Die Funktion erzeugt aus einem Vektor, der die Bestandteile einer Zeitangabe beschreibt, einen numerischen Zeitstempel.

Parameter:

- `tm`: 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.

Rückgabewert:

Numerischer Datums- und Zeitwert (wie time_t in C)

Hinweise:

- Ist *tm* nicht vom Typ vector oder unterschreitet seine Größe 9 Elemente, wird eine Ausnahme vom Typ string ausgelöst.
- 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.
- Ist *tm* keine Konstante, werden die darin enthaltenen Werte während des Funktionsaufrufs aktualisiert und so normalisiert, dass alle Felder entsprechend dem angegebenen Datum gültige Werte enthalten. Die Funktion lässt sich somit auch zur Normalisierung der Werte in *tm* verwenden.
- Die Werte in *wday* und *yday* von *tm* werden bei der Bestimmung des Resultats ignoriert.

Funktion strftime

`string strftime(const string fmt, const var tm)`

Die Funktion konvertiert Datums- und Zeitangaben des in *tm* übergebenen Kalenderzeitpunkts in eine Zeichenkettendarstellung gemäß der in *fmt* angegebenen Formatspezifikation.
Sie verhält sich ähnlich der gleichnamigen C-Funktion (und basiert auch darauf).

Parameter:

- fmt: Formatspezifikation. Die Formatspezifikation ist eine Zeichenkette, bestehend aus Formatierungsspezifizierern und beliebigen Zeichen (außer %). Alle gewöhnlichen Zeichen, einschließlich des terminierenden NULL-Zeichens, werden unverändert in das Ergebnis kopiert.. Jeder Konvertierungsspezifizierer beginnt mit einem Prozent-Zeichen (%), gefolgt von einem Zeichen, das das Verhalten des jeweiligen Spezifizierers festlegt. Mindestens die in Tabelle 18 angegebenen Spezifizierer werden unterstützt, abhängig von der verwendeten Laufzeitplattform können noch weitere unterstützt werden.
- tm: Lokalisierter Zeitpunkt (Vektor aus 9 Elementen, entsprechend dem Ergebnis von localtime) oder C-Timestamp (wie das Ergebnis der time-Funktion).

Spezifizierer	Bedeutung
%%	Zeichen %
%a	Abkürzung des Wochentags
%A	Wochentag
%b	Abkürzung des Monatsnamens
%B	Monatsname
%c	Datum und Zeit
%d	Tag des Monats (01 .. 31)
%H	Stunde (00 .. 23)
%I	Stunde (01 .. 12)
%j	Tag im Jahr (001 .. 366)
%m	Monat als Dezimalzahl (01 .. 12)
%M	Minute (00 .. 59)
%p	AM oder PM (abhängig von der angegebenen Zeit)
%S	Sekunde (00 .. 59)
%U	Wochennummer (00 .. 53) mit Sonntag als erstem Wochentag
%w	Wochentagsnummer (0 .. 6), 0 = Sonntag
%W	Wochennummer (00 .. 53) mit Montag als erstem Wochentag
%x	Bevorzugte Repräsentation des Datums
%X	Bevorzugte Repräsentation der Uhrzeit
%y	Jahr ohne Jahrhundert (00 … 99)
%Y	Jahr mit Jahrhundert
%Z	Zeitzonenname (leer wenn keine Zeitzone verfügbar)

Tabelle 18 Formatspezifizierer für `strftime`

Rückgabewert:

Bei Erfolg Zeichenkettendarstellung der Datums- und Zeitinformation, sonst eine leere Zeichenkette.

Hinweis:
Bei Angabe eines ungültigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

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.
Unter Windows-CE liefert sie die vergangene Zeit seit den Systemstart, unter allen anderen Plattformen die Zeit seit dem Start des aktuellen Programms (genauer: dem Start der B++-Laufzeitumgebung).

Rückgabewert:

Vergangene Zeit in Millisekunden.

Hinweis:

Abhängig von der verwendeten Plattform kann die Auflösung geringer als eine Millisekunde sein, unter MS-DOS liegt sie z. B. bei etwa 55 ms.

Beispiel:

```
main() {
   uint t1 = timer();
   // do anything here
       ...
   uint t2 = timer();
   print(t2-t1, " ms needed for operation.\n");
}
```

Beispiel 74 Verwenden von `timer()` zur Laufzeitmessung

13.6 *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 10.11.10 und 13.2.1).

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 13.7) abfragen.

Hinweis:

B++ definiert nur einen minimalen Satz mathematischer Funktionen. Zusätzlich werden über den Header `cmath.bph` die weiteren in C/C++ (dort deklariert in `cmath.h`) vorhandenen mathematischen Grundfunktionen bereitgestellt – siehe hierzu die Referenz.

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[77] zulässig. Eine implizite Typumwandlung erfolgt hier nicht.

Funktion atan:

`double atan(double x)`

Die Funktion berechnet den Arcus-Tangens des als Argument übergebenen Wertes.

Parameter:

- `x`: Argument

Rückgabewert:

Winkel im Bogenmaß im Bereich (-π/2, +π/2)

Funktion cos:

`double cos(double rad)`

Die Funktion berechnet den Kosinus des als Argument übergebenen Winkels.

Parameter:

- `rad`: Winkelangabe im Bogenmaß

Rückgabewert:

Kosinus von *rad*

Funktion exp:

`double exp(double x)`

Die Funktion berechnet den Wert der Exponentialfunktion e^x.

[77] Zeichentypen sind hier nicht zugelassen.

Parameter:

- `x`: Exponent

Rückgabewert:

Wert von e^x

Funktion log:

`double log(double x)`

Die Funktion berechnet den natürlichen Logarithmus `ln(x)`.

Parameter:

- `x`: Argument; der Argumentwert muss positiv sein.

Rückgabewert:

Wert von `ln(x)`

Funktion pow:

`double pow(double x, double y)`

Die Funktion berechnet den Wert der Exponentialfunktion x^y.

Parameter:

- `x`: Basis
- `y`: Exponent

Rückgabewert:

Wert von x^y

Hinweis:

Wenn der Exponent y nicht ganzzahlig ist, darf die Basis x nicht negativ sein.

Funktion sin:

`double sin(double rad)`

Die Funktion berechnet den Sinus des als Argument übergebenen Winkels.

Parameter:

- `rad`: Winkelangabe im Bogenmaß

Rückgabewert:

Sinus von *rad*

Funktion sqrt:

`double sqrt(double 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(double rad)`

Die Funktion berechnet den Tangens des als Argument übergebenen Winkels.

Parameter:

- `rad`: Winkelangabe im Bogenmaß

Rückgabewert:

Tangens von *rad*

13.7 *Systemfunktionen*

Funktion addr:

`intptr addr(const var variable)`

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:

- `variable`: 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 Funktionsaufrufs.

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 Windows eine nach STDCALL-, sonst eine nach C-Konvention implementierte Funktion[78] 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 MS-DOS sowie in den 32-Bit-Windows-Desktop-Versionen ist die Anzahl der Argumente beliebig, auf allen anderen Plattformen sind maximal 16 Argumente möglich.

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

[78] Unter 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.

```
int login(const char* userName, const char* passwd)
```

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 75 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 75 Aufruf einer DLL-Funktion aus B++

Funktion chdir:

```
bool chdir(const string dirpath)
```

Die Funktion wechselt das aktualle Arbeitsverzeichnis.

Parameter:

- `dirpath`: Verzeichnispfad, der als Arbeitsverzeichnis festgelegt werden soll.
  Dies kann eine absolute oder relative Pfadangabe sein.

Rückgabewert:

`true` bei Erfolg.

Hinweise:

- Für das Argument *`dirpath`* wird kann jeder Typ übergeben werden, der implizit in `string` konvertierbar ist. Ist keine solche Konvertierung möglich, wird eine Ausnahme vom `string`-Typ erzeugt.
- Auf MS-DOS- und Windows-Systemen erfolgt, sofern *`dirpath`* eine Laufwerksangabe enthält, auch ein Wechsel des aktiven Laufwerks. Enthält `dirpath` hier nur eine Laufwerksangabe, erfolgt nur ein Laufwerkswechsel, wobei das aktive Verzeichnis auf dem jeweiligen Laufwerk unverändert bleibt.
- Ist der Aufruf nicht erfolgreich, kann über `errno()` die Fehlerursache abgefragt werden.

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 (Literale) 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 hdll)
```

Die Funktion gibt eine mit `LoadLibrary` geladene dynamische Bibliothek wieder frei.

Parameter:

- `hdll`: Handle der freizugebenden Bibliothek

Rückgabewert:

1 bei Erfolg, sonst (d. h. wenn *`hdll`* ungültig war) 0

Funktion fullpath:

```
string fullpath(const string relpath)
```

Die Funktion ermittelt den vollständigen Pfad zum angegebenen relativen Pfad, ausgehend vom aktuellen Arbeitsverzeichnis.

Parameter:

- `relpath`: Relative Pfadangabe

Rückgabewert:

Bei Erfolg die vollständig qualifizierte Pfadangabe zu relpath, sonst eine leere Zeichenkette.

Hinweise:

- Für das Argument rel*`path`* wird kann jeder Typ übergeben werden, der implizit in `string` konvertierbar ist. Ist keine solche Konvertierung möglich, wird eine Ausnahme vom `string`-Typ erzeugt.
- Wenn *`relpath`* leer ist oder nur einen Punkt enthält, entspricht das Ergebnis dem des Aufrufs von `getcwd()`
- Auf MS-DOS- und Windows-Systemen kann *`relpath`* eine Laufwerksangabe enthalten. In diesem Fall bezieht sich das Ergebnis auf das angegebene Laufwerk. Enthält *`relpath`* ausschließlich eine Laufwerksangabe (Buchstabe, gefolgt von einem Doppelpunkt), ist das Ergebnis der Pfad des Arbeitsverzeichnisses auf dem betraffenden Laufwerk.

Funktion getcwd:

```
string getcwd()
```

Die Funktion ermittelt das aktuelle Arbeitsverzeichnis.

Rückgabewert:

Vollständiger Pfad des aktuellen Arbeitsverzeichnisses.

Funktion getenv:

```
dictionary getenv()
string getenv(const 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 hdll, const string funcname)
```

Die Funktion liefert die Adresse der unter dem Namen *funcname* aus der durch *hdll* spezifizierten DLL exportierten Funktion.

Parameter:

- `hdll`: 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 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(const 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 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
  - d : bezeichnet eine Debug-Version
  - _64 : bezeichnet eine 64-Bit-Version

Beispiel 76 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 76 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 memsize:

```
uint memsize()
```

Die Funktion ermittelt die Größe des noch verfügbaren freien Speichers.

Rückgabewert:

Größe des verfügbaren freien Speichers in Bytes

Hinweise:

- Genau genommen ist das Ergebnis die Größe des größten zusammenhängenden freien Speicherbereichs auf dem Heap.
- Weil der Typ des Rückgabewertes nur 32 Bit breit ist, ist der maximale Ergebniswert 2^{32}-1. Auf 64-Bit-Systemen kann der tatsächlich verfügbare Speicher wesentlich größer sein.
- Auf Plattformen mit virtuellem Speicher (Windows, Linux …) ist das Ergebnis der verfügbare virtuelle Speicher, nicht der physische.

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 77 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 77 Verwenden der Funktionen `nerr()` und `strerror(int)`

Funktion setenv:

`bool setenv(const string` *`varname`*`, const string` *`value`*`)`

Die Funktion definiert eine neue Umgebungsvariable oder ändert den Wert einer vorhandenen.

Parameter:

- `varname`: Name der Umgebungsvariablen
- `value`: neuer Variablenwert

Rückgabewert:

`true` bei Erfolg, sonst `false`

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`*`, const string` *`locale`*`)`

Die Funktion setzt das Lokalisierungsschema der Laufzeitumgebung oder fragt dessen aktuelle Einstellungen ab. Sie verhält sich wie die gleichnamige C-Funktion[79].

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

[79] Die Implementierung in B++ ruft die C-Funktion direkt auf.

- `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“, .„.UTF-8“ 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.

Rückgabewert:

Bei Erfolg, eine Zeichenkette, die dem aktuell gesetzten Wert

Hinweise:

- Das konkrete Verhalten der Funktion (insbesondere die Interpretation der Angabe in *`locale`*) hängt vom verwendeten Betriebssystem ab.
  Enthält die Zeichenkette eine Zeichensatzbeschreibung (nach einem Punkt), so wird auf Windows-Systemen eine Ziffernfolge direkt als Codepage-ID interpretiert, andernfalls wird die Zeichenfolge als Zeichensatzname interpretiert und versucht, die zugehörige Codepage-ID zu ermitteln. Auf Unix-Systemen wird dagegen bei einer Ziffernfolge versucht, einen zugehörigen Zeichensatznamen zu ermitteln.
- 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 sleep:

```
void sleep(uint milliseconds)
```

Die Funktion hält die Programmausführung für die angegebene Zeitdauer an.

Parameter:

- `milliseconds`: Zeitdauer in Millisekunden.

Rückgabewert:

Fehlerbeschreibung

Hinweis:

Unter MS-DOS wird das gesamte Programm, sonst der aktuelle Thread angehalten.

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

Hinweise:

- Der Aufruf ohne Parameter entspricht einem Aufruf von `strerror(errno())`.
- Wird für `errcode` ein ungültiger Wert angegeben, ist das Ergebnis „Unknown error“.

13.7.1 Funktionen zur Prozesssteuerung

Die Funktionen dieses Abschnitts dienen der Ausführung und Steuerung von Kindprozessen. Plattformspezifische Besonderheiten werden in der Beschreibung der einzelnen Funktionen benannt.

Funktion existspid

```
bool existspid(int pid)
```

Die Funktion prüft, ob der durch *`pid`* spezifizierte Prozess existiert.

Parameter:

- `pid`: Identifikator des Prozesses, dessen Existenz geprüft werden soll. Typischerweise ist dies das Ergebnis eines vorangegangenen Aufrufs von `spawn`.

Rückgabewert:

Bei Erfolg, also wenn *`pid`* auf einen existierenden Prozess verweist, ist das Ergebnis *`true`*, andernfalls *`false`*.

Hinweise:

- Die Funktion ist unter MS-DOS nicht verfügbar, da es dort keine asynchronen Prozesse gibt.
- Das Argument *`pid`* ist streng typisiert, es wird bei Bedarf also eine implizite Typkonvertierung durchgeführt. Im Falle eines ungültigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

Funktion kill

```
bool kill(int pid)
```

Die Funktion bricht den durch *pid* spezifizierten Prozess ab.

Parameter:

- pid: Identifikator des Prozesses, der beendet werden soll. Typischerweise ist dies das Ergebnis eines vorangegangenen Aufrufs von spawn.

Rückgabewert:

Bei Erfolg, also wenn *pid* auf einen existierenden Prozess verweist, ist das Ergebnis *true*, andernfalls *false*.

Hinweise:

- Die Funktion ist unter MS-DOS nicht verfügbar, da es dort keine asynchronen Prozesse gibt.
- Das Argument *pid* ist streng typisiert, es wird bei Bedarf also eine implizite Typkonvertierung durchgeführt. Im Falle eines ungültigen Argumenttyps wird eine Ausnahme vom Typ string ausgelöst.
- Es ist nicht garantiert, dass der Prozess bei der Rückkehr aus der Funktion tatsächlich bereits beendet ist. Ein Programm, das dies für die weitere Abarbeitung voraussetzt, sollte nach dem Aufruf von kill(*pid*) zusätzlich waitpid(*pid*) aufrufen.

Funktion pclose:

```
int pclose(FILE stream)
```

Die Funktion wartet auf den Abschluss des zuvor bei einem Aufruf von popen übergebenen Kommandos und schließt den *stream* auf der zugeordneten Pipe.

Parameter:

- stream: Rückgabewert des vorangegangenen Aufrufs von popen

Rückgabewert:

Ergebnis (Exit-Status) des aufgerufenen Kommandos oder -1, wenn ein Fehler aufgetreten ist.

Hinweise:

- Auf DOS-Plattformen sowie in Windows-Anwendungen ohne Konsole wird die Funktion mit Hilfe einer temporären Datei und einem Aufruf von system nachgebildet.
- Ist das Ergebnis -1, kann mit Hilfe der Funktion errno ein Fehlerstatus ermittelt werden.

Funktion popen:

```
FILE popen(const string cmd, const string mode)
```

Die Funktion eröffnet eine Pipe und führt das in *cmd* angegebene Kommando aus..

Parameter:

- `cmd`: Auszuführendes Kommando
- `mode`: Modus des als Ergebnis gelieferten Streams; kann folgende Zeichen enthalten:
  - `r`: Der aufrufende Prozess kann den als Ergebnis gelieferten Stream verwenden, um die Standardausgabe des aufgerufenen Kommandos *cmd* zu lesen.
  - `w`: Der aufrufende Prozess kann in die Standardeingabe des aufgerufenen Kommandos schreiben.
  - `t`: Die Pipe wird im Textmodus eröffnet.
  - `b`: Die Pipe wird im Binärmodus eröffnet.

Rückgabewert:

Stream, der mit einem der „Enden“ der erzeugten Pipe verbunden ist; das andere „Ende“ ist entweder mit der Standardein- oder der Standardausgabe des Kommandos verbunden.
Im Fehlerfall ist das Ergebnis Null.

Hinweise:

- Auf DOS-Plattformen sowie in Windows-Anwendungen ohne Konsole wird die Funktion mit Hilfe einer temporären Datei und einem Aufruf von `system` nachgebildet.
- Ist das Ergebnis -1, kann mit Hilfe der Funktion `errno` ein Fehlerstatus ermittelt werden.

Funktion spawn

```
int spawn(uint mode, const string cmd, const string args = null)
```

Die Funktion startet ein Programm *cmd* mit den in *args* angegebenen Argumenten als Kindprozess.

Parameter:

- `mode`: Ausführungsmodus. Folgende Modi sind definiert:

Symbol	Wert	Beschreibung
P_WAIT	0	Synchrone Prozessausführung
P_NOWAIT	1	Asynchrone Prozessausführung
P_OVERLAY	2	Ersetzen des aktuellen Prozesses durch den Kindprozess
P_NOWAITO	3	wie P_NOWAIT
P_DETACH	4	Asynchrone Prozessausführung ohne Zugriff des Kindprozesses auf Standardein- und -ausgabe

- cmd: Name oder Pfad des auszuführenden Programms. Enthält *cmd* keine Pfadtrennzeichen, wird eine systemspezifische Suche über die in der Umgebungsvariablen PATH spezifizierten Verzeichnisse durchgeführt, andernfalls wird nur der angegebene Pfad verwendet. Ist für *cmd* keine Dateierweiterung angegeben, wird unter MS-DOS und Windows nach ausführbaren Dateien mit den Erweiterungen com, exe, bat und cmd (in dieser Reihenfolge, cmd nur unter Windows) gesucht.
- args: Kommandozeilenargumente, die an das auszuführende Programm übergeben werden. Die Übergabe an den Kindprozess erfolgt wie bei der Eingabe auf der Kommandozeile. Das erste übergebene Argument (Index 0) ist stets der Wert von *cmd*.

Rückgabewert:

Bei erfolgreichem Aufruf ist der Rückgabewert für den Modus P_WAIT der Rückgabewert (exit code) des aufgerufenen Programms, für den Modus P_DETACH stets Null und für P_NOWAIT bzw. P_NOWAITO die Prozess-ID des erzeugten Kindprozesses.
Beim Modus P_OVERLAY kehrt die Funktion nicht zurück, es gibt also kein Ergebnis.
Schlägt der Aufruf fehl, ist das Ergebnis -1.

Hinweise:

- Unter MS-DOS gibt es keine asynchrone Prozessausführung. Deshalb werden nur die Modi P_WAIT und P_OVERLAY unterstützt. Nicht unterstützte oder ungültige Werte für *mode* werden wie P_WAIT behandelt.
  Modus P_DETACH wird nur unter Windows unterstützt.
  Auf Windows- und UNIX-Systemen werden ungültige oder nicht unterstützte Werte für mode wie P_NOWAIT behandelt.
- Auf Windows-Systemen entspricht der Ergebniswert dem Prozess-Handle, nicht dem von der Windows-API-Funktion GetProcessId ermittelten Wert.
- Schlägt der Aufruf fehl, kann mit Hilfe von errno() ein Fehlercode bestimmt werden.
- Die Funktionsargumente sind streng typisiert, es wird bei Bedarf also eine implizite Typkonvertierung durchgeführt. Im Falle ungültiger Argumenttypen wird eine Ausnahme vom Typ string ausgelöst.
- Im Unterschied zur Funktion system benutzt spawn nicht den Kommandozeileninterpreter. Deshalb ist die Ausführung interner Shell-Kommandos nicht direkt möglich. Dateiumleitungen und Pipes in *args* sind dagegen zulässig.

Funktion system:

`int system(const 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` unter DOS/Windows bzw. `SHELL` unter UNIX-Systemen.
- 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.

Funktion waitpid

`int waitpid(int pid)`

Die Funktion wartet auf die Beendigung des durch *pid* spezifizierten Prozesses und gibt dessen Ergebnis zurück.

Parameter:

- `pid`: Identifikator des Prozesses, auf dessen Beendigung gewartet werden soll. Typischerweise ist dies das Ergebnis eines vorangegangenen Aufrufs von `spawn`.

Rückgabewert:

Bei Erfolg, also wenn *pid* auf einen existierenden Prozess verweist, ist das Ergebnis der Rückgabewert (exit code) des betreffenden Prozesses, andernfalls -1.

Hinweise:

- Die Funktion ist unter MS-DOS nicht verfügbar, da es dort keine asynchronen Prozesse gibt.
- Das Argument *pid* ist streng typisiert, es wird bei Bedarf also eine implizite Typkonvertierung durchgeführt. Im Falle eines ungültigen Argumenttyps wird eine Ausnahme vom Typ `string` ausgelöst.

13.7.2 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(const 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. `segread`) 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 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` 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 78 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 78 Verwenden von Interrupt-Aufrufen unter MS-DOS

Funktion outport:

`void outport(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(const 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.

13.7.3 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 hierfür kein expliziter Aufruf von `LoadLibrary` bzw. `FreeLibrary` erforderlich ist. Die `GetXXXFunc`-Funktionen entsprechen dem Aufruf von `GetProcAddress` für die jeweilige Standardbibliothek.

```
intptr GetCoreFunc (const string name)
intptr GetGdi32Func (const string name)
intptr GetKernel32Func (const string name)
intptr GetOle32Func (const string name)
intptr GetShell32Func (const string name)
intptr GetUser32Func (const 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.

13.8 *Callbacks*

Manche plattformspezifischen System-APIs stellen Funktionen bereit, die Callbacks in Form von C-Funktionen aufrufen. Die direkte Implementierung solcher Callback-Funktionen ist in B++ nicht möglich, da hierfür zur Laufzeit plattformspezifischer Code generiert werden müsste.

Ab Version 1.5 stellt B++ jedoch zwei Sätze vordefinierter Callback-Funktionen bereit, die die Weiterleitung von Callbacks auf aufrufbare B++-Objekte gestatten.

Die erste Form nutzt die Tatsache aus, dass Callback-Funktionen häufig mit einem Zeiger auf benutzerdefinierte Daten, meist als letztes Argument und oft als *UserData* bezeichnet, aufgerufen werden. Diese Eigenschaft wird genutzt, um beim Aufruf einen Verweis auf ein B++-Funktorobjekt zu übergeben, an das die eigentliche Behandlung des Callbacks delegiert wird. Solche Callback-Funktionen werden in B++ als *dynamische Callbacks* bezeichnet. Es gibt insgesamt acht solcher vordefinierter Callback-Funktionen (mit einem bis acht Argumenten), die mit Hilfe der Funktion `GetCallbackFunc` abgerufen werden können.

Die zweite Form setzt keinen UserData-Zeiger voraus. Stattdessen ist hierfür eine Tabelle von jeweils fünf fest vordefinierten Callback-Funktionen mit 0 bis 4 Argumenten vorhanden. Für jede dieser (insgesamt 25) Funktionen kann ein B++-Funktorobjekt installiert werden, an das die Behandlung des Callback-Aufrufs delegiert wird. Die Funktionen dieser Tabelle werden als `statische Callbacks` bezeichnet, weil jeder Callback-Funktion ein Funktor fest zugeordnet ist. Sie werden mit der Funktion `GetStaticCallbackFunc` abgerufen. Die Handler-Objekte (Funktoren) werden mit `SetStaticCallbackHandler` installiert.

Für den Rückgabewert und alle Argumente der Callback-Funktionen wird der Typ `intptr` angenommen, d. h., sie haben die Größe eines Zeigers. Aus Sicht des Clientcodes sind somit alle einfachen Datentypen (B++-Werttypen) als Argumente bzw. Rückgabetyp nutzbar, die die Größe eines Zeigers nicht überschreiten. Werte von Referenztypen müssen als Zeiger, ermittelt durch die Funktion `addr()`, übergeben werden.

Hinweis:

In den MS-DOS-Builds ist die Callback-Unterstützung standardmäßig deaktiviert, weil Systemaufrufe hier üblicherweise über Interrupts realisiert werden und B++ nicht für die Implementierung von Interrupt-Serviceroutinen vorgesehen ist. Wird die Funktionalität benötigt, muss B++ mit gesetztem Symbol `USECALLBACK` kompiliert werden.

Funktion GetCallbackFunc

`intptr GetCallbackFunc(uint` *`numArgs`*`)`

Die Funktion liefert einen Zeiger auf eine dynamische Callback-Funktion zur Übergabe als Parameter an Systemfunktionen mit der in *`numArgs`* angegebenen Anzahl von Argumenten.

Jede dieser Funktionen erwartet als letztes Argument einen Zeiger auf einen B++-Funktor, also ein Objekt, das einen call-Operator (`operator()`) definiert. Dieser Funktor wird von der Callback-Funktion mit den übergebenen Argumenten (bis auf das letzte, das die Adresse des Funktors selbst enthält) aufgerufen. Hierzu muss die jeweilige Systemfunktion, die den Callback aufruft, einen UserData-Verweis als letztes Argument unterstützen, wie das z.B. bei den meisten

Windows-API-Funktionen der Fall ist.

Parameter:

- `numArgs`: Anzahl der Argumente der Callback-Funktion, einschließlich des Zeigers auf das Funktor-Objekt. Hierbei sind Werte von 1 bis 8 zulässig.

Rückgabewert:

Adresse der Callback-Funktion

Hinweis:

Im Falle eines ungültigen Wertes von *`numArgs`* wird eine Ausnahme vom Typ `string` erzeugt.

Beispiel 79 demonstriert die Verwendung einer Callback-Funktion für den Aufruf von `EnumWindows` (Win32-API).

```
// Functor for callback
class WinEnumerator {
  private:
     charbuffer txtbuf = null;
     intptr func = 0;
  public:
   WinEnumerator() {
      txtbuf = newcbuffer(256); // initialize buffer
      // get address of GetWindowText function
      func = GetUser32Func("GetWindowText?");
   }
   // call operator
   uint operator()(intptr hwnd) {
      // get window text for handle
      CallLibFunc(func,hwnd,addr(txtbuf),size(txtbuf)-1);
      auto str = string(txtbuf); // convert buffer to string
      if (size(str) > 0)
         print(ulong(hwnd),":",str,"\n");
      return 1;
   }
};

void main() {
    intptr pfunc = GetUser32Func("EnumWindows");
    // get callback function of 2 arguments
    // (first for window handle, second for enumerator object)
    intptr pcallback = GetCallbackFunc(2);
    WinEnumerator we; // build instance of functor object
    // call EnumWindows with callback function and address of functor object
    CallLibFunc(pfunc,pcallback,addr(we));
}
```

Beispiel 79 Implementierung eines Window-Enumerators für das Win32-API

Hier wird zunächst ein Funktor `WinEnumerator` definiert, dessen call-Operator als Argument ein Window-Handle erwartet und das Handle sowie den jeweiligen Fenstertext ausgibt.

In der *`main`*-Funktion wird eine Callback-Funktion mit 2 Argumenten angefordert, die wiederum zusammen mit der Adresse einer Instanz von `WinEnumerator` dem Aufruf der API-Funktion `EnumWindows` übergeben wird.

Funktion GetStaticCallbackFunc

`intptr GetStaticCallbackFunc(uint numArgs, uint idxInstance)`

Die Funktion liefert einen Zeiger auf die durch *idxInstance* festgelegte statische Callback-Funktion mit der in *numArgs* angegebenen Anzahl von Argumenten.

Parameter:

- `numArgs`: Anzahl der Argumente des Callbacks; es sind Werte von 0 bis 4 erlaubt.
- `idxInstance`: Index der Instanz der Callback-Funktion mit numArgs Argumenten; es sind Werte von 0 bis 4 erlaubt.

Rückgabewert:

Adresse der Callback-Funktion

Hinweise:

- Bevor die Callback-Funktion aufgerufen wird muss durch einen Aufruf von `SetStaticCallbackHandler` ein entsprechendes Funktor-Objekt installiert werden.
- Im Falle ungültiger Werte von *numArgs* bzw. *idxInstance* wird eine Ausnahme vom Typ `string` erzeugt.

Funktion SetStaticCallbackHandler

`void SetStaticCallbackHandler(uint numArgs, uint idxInstance, var handler)`

Parameter:

- `numArgs`: Anzahl der Argumente des Callbacks; es sind Werte von 0 bis 4 erlaubt.
- `idxInstance`: Index der Instanz der Callback-Funktion mit *numArgs* Argumenten; es sind Werte von 0 bis 4 erlaubt.
- `handler`: Funktor-Objekt für die Behandlung des Callbacks. Das Argument kann ein Objektwert, eine Weak-Referenz auf ein Objekt (vgl. 16.1) oder die Adresse eines Objekts als `intptr` sein.

Hinweise:

- Wenn *handler* ein Objektwert ist, wird dessen Referenzzähler inkrementiert, d. h., das Funktor-Objekt kann nicht freigegeben werden, solange entweder ein anderer Handler installiert oder der Handler explizit deinstalliert wurde (Aufruf von `SetStaticCallbackHandler` mit `null` als Wert für *handler*).
- Ist *handler* eine Weak-Referenz oder eine Adresse, muss der Clientcode dafür sorgen, dass das entsprechende Objekt existiert, solange es aufgerufen werden kann.
- Im Falle ungültiger Werte von *numArgs* bzw. *idxInstance* wird eine Ausnahme vom Typ `string` erzeugt.

13.9 *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 80 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 80 Zeichnen von Linien im Grafikmodus unter MS-DOS

13.10 *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 81 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 81 Implementieren einer Funktion mit variabler Argumentzahl

Funktion argcnt:

```
int argcnt()
```

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 `argcnt` 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(const var 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 oder ein Funktionsobjekt (Funktor) 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) oder Funktor
- `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 82 illustriert die Verwendung von `call` zur Nachbildung der `printf`-Funktion aus C.

```
int printf(const string format, ...)
{
   vector v = argv();
   print(call(newstring,v));
   return int(size(v))-1;
}
```

Beispiel 82 Nachbildung der `printf`-Funktion

Funktion callafter:

`void callafter(const var` *`callable`*`, vector` *`args=null`*`)`

Die Funktion installiert eine angehängte Funktionserweiterung (vgl. Abschnitt 12.2), die unmittelbar nach dem Verlassen der aufrufenden Funktion ausgeführt wird.

Parameter:

- `callable`: Funktion oder Funktor die/der beim Verlassen der Funktion aufgerufen werden soll
- `args`: Vektor mit Argumenten für den Aufruf von *`callable`*

Hinweis:

Für *`callable`* kann eine einfache Funktion, eine Elementfunktion oder ein Funktionsobjekt angegeben werden. Falls *`callable`* eine Elementfunktion ist, muss als erster Wert in *`args`* der Verweis auf das zugehörige Objekt übergeben werden.

Funktion callfail:

`void callfail(const var` *`callable`*`, vector` *`args=null`*`)`

Die Funktion installiert eine angehängte Funktionserweiterung (vgl. Abschnitt 12.2), die unmittelbar nach dem Verlassen der aufrufenden Funktion ausgeführt wird, falls die aufrufende Funktion eine Ausnahme ausgelöst hat.

Parameter:

- `callable`: Funktion oder Funktor die/der beim Verlassen der Funktion aufgerufen werden soll

- `args`: Vektor mit Argumenten für den Aufruf von *`callable`*

Hinweise:

- Für *`callable`* kann eine einfache Funktion, eine Elementfunktion oder ein Funktionsobjekt angegeben werden. Falls *`callable`* eine Elementfunktion ist, muss als erster Wert in *`args`* der Verweis auf das zugehörige Objekt übergeben werden.
- Die Ausnahme wird an den Aufrufer der aktuellen Funktion weitergeleitet. Sie kann von der angehängten Erweiterung nicht behandelt werden.

Funktion callsucc:

```
void callsucc(const var callable, vector args=null)
```

Die Funktion installiert eine angehängte Funktionserweiterung (vgl. Abschnitt 12.2), die unmittelbar nach dem Verlassen der aufrufenden Funktion ausgeführt wird, falls die aufrufende Funktion keine Ausnahme ausgelöst hat.

Parameter:

- `callable`: Funktion oder Funktor die/der beim Verlassen der Funktion aufgerufen werden soll
- `args`: Vektor mit Argumenten für den Aufruf von *`callable`*

Hinweis:

Für *`callable`* kann eine einfache Funktion, eine Elementfunktion oder ein Funktionsobjekt angegeben werden. Falls *`callable`* eine Elementfunktion ist, muss als erster Wert in *`args`* der Verweis auf das zugehörige Objekt übergeben werden.

Funktion cleartimer:

```
bool cleartimer(intptr handle)
```

Die Funktion entfernt (deinstalliert) einen zuvor mit `settimer` installierten synchronen Zeitgeber (vgl. Abschnitt 12.1).

Parameter:

- `handle`: Handle (ID) des Zeitgebers, der entfernt werden soll.

Rückgabewert:

`true` bei Erfolg

Funktion cleartimers:

```
void cleartimers()
```

Die Funktion entfernt (deinstalliert) alle zuvor mit `settimer` installierten synchronen Zeitgeber (vgl. Abschnitt 12.1).

Funktion clone:

`var clone(const var val)`

Die Funktion erzeugt eine Kopie eines Wertes[80] beliebigen Typs.

Sie dient der Erzeugung einer vollständigen (tiefen) Kopie von Werten der Containertypen (`vector`, `buffer`, `charbuffer` bzw. `dictionary`), des Typs `string` sowie von Bytecode-Funktionen mit statischen Variablen[81] 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, const string srcname = "", bool debug = false)`

`bool compile(const string src, const 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
- `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 B++, indem der Code nicht zur Übersetzungs-, sondern zur Laufzeit eingebunden wird.

[80] Im Fall einer Weak Reference wird nur die Referenz, nicht das referenzierte Datum kopiert.
[81] Eine solche Kopie ist dann eine eigenständige Funktion, deren Implementierung der der Vorlage entspricht, deren statische Variablen aber von der Vorlage unabhängig sind.

- 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 83 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 83 Dynamisches Ersetzen einer Funktion mit compile

Funktion compilefn:

```
function compilefn(const string src, const string args = "",
                   const string rtype = "",
                   var ownerClass = null, const 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 gewöhnlichen 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 84 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 84 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 (const string src, const 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(const 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` bei Erfolg, sonst `false`
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 settimer:

```
intptr settimer(uint milliseconds, const var callable,
                bool cyclic=false, vector args=null)
```

Die Funktion installiert einen synchronen Zeitgeber (vgl. Abschnitt 12.1).

Parameter:

- `milliseconds`: Zeitspanne (Intervall) in Millisekunden
- `callable`: Funktion, Elementfunktion oder Funktor die/der nach der in *`milliseconds`* angegebenen Zeitspanne ausgeführt werden soll
- `cyclic`: Flag für den Ausführungsmodus; der Wert *`true`* bedeutet eine zyklische (wiederholte) Ausführung, *`false`* eine einmalige Ausführung
- `args`: Vektor mit Aufrufargumenten für *`callable`*

Rückgabewert:

Handle (ID) des installierten Zeitgebers

Hinweise:

- Wird für `callable` eine Elementfunktion (Methodenzeiger) angegeben, muss *`args`* als ersten Wert den Verweis auf die zugehörige Objektinstanz enthalten.
- Wird für die Zeitspanne (`milliseconds`) Null angegeben, wird kein Zeitgeber installiert. Das Ergebnis ist dann ebenfalls Null.
- Der Rückgabewert kann verwendet werden, um einen installierten Zeitgeber mit *`cleartimer`* wieder zu entfernen.

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 85 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 85 Zufallszahlen mit den Funktionen `rand` und `srand`

14 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 (T-Rex 1.3) [7] aufsetzt. Der dabei unterstützte Sprachumfang entspricht im Wesentlichen einer Teilmenge der ECMA-Syntax für reguläre Ausdrücke (siehe hierzu auch [8]) mit einigen implementierungsbedingten Besonderheiten.

14.1 *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
`\`	Hebt die Sonderbedeutung des nächsten Zeichens auf; insbesondere steht `\\` für die Repräsentation eines Backslashs.
`.`	Der Punkt steht für ein beliebiges Zeichen außer dem Zeilenumbruch.
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator
`\n`	Zeilenumbruch
`\r`	Wagenrücklauf
`\f`	Seitenumbruch
`|`	Trennt zwei alternative Teilausdrücke (ODER).

Tabelle 19 Sonderzeichen in regulären Ausdrücken

Hinweis:

Escape-Sequenzen für beliebige Zeichencodes in der Form `\xhh` 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 beide zu einem Wort oder nicht.

Tabelle 20 Assertions in regulären Ausdrücken

Klammern

Klammern werden zur Gruppierung von Teilausdrücken, für die Definition benutzerdefinierter Zeichenklassen sowie für benutzerdefinierte Quantifizierer verwendet. Tabelle 21 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 21 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 22 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. (Zeichencodes von 0x09 bis 0x0D sowie 0x20)
\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 \w, \s oder \c gehören.
\P	Umkehrung von \p, also alle Zeichen, die keine Sonderzeichen sind.

Tabelle 22 Vordefinierte Zeichenklassen

Hinweis:
Für die Bildung der Standardklassen werden intern Funktionen der C-Bibliothek (Header ctype.h) verwendet. Daraus können sich Abweichungen auf Grund der aktuellen Lokalisierungseinstellungen ergeben.

Benutzerdefinierte Zeichenklassen werden nach der Syntax

```
'[' [^] characterset ']'
```

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 *characterset* auch Zusammenfassungen aufeinander folgender Zeichen in der Form *start-end*, Sonderzeichen aus Tabelle 19 sowie vordefinierte Zeichenklassen aus Tabelle 22 stehen.

In Tabelle 23 sind einige Beispiele benutzerdefinierter Zeichenklassen angegeben.

Definition	Bedeutung
`[0-7]`	Oktalziffern
`[\d\.]`	Dezimalziffern oder ein Punkt
`[ÄäÖöÜü]`	Umlaute
`[^a-zäöüß]`	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\C]`	Alle druckbaren Zeichen (kein Whitespace und kein Steuerzeichen)

Tabelle 23 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 24 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.
`{`*n*`}`	Das Symbol kommt genau *n*-mal vor.
`{`*n*`,}`	Das Symbol kommt mindestens *n*-mal vor.
`{,`*m*`}`	Das Symbol kommt höchstens *m*-mal vor.
`{`*n*`,`*m*`}`	Das Symbol kommt mindestens *n*-mal und höchstens *m*-mal vor.

Tabelle 24 Quantifizierer

Hinweis:
In obiger Tabelle sind *n* und *m* natürliche Dezimalzahlen, wobei *n* <= *m* gilt.

14.2 *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 18.2.2.2 (S. 245) 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 13) 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.

14.2.1 Methoden

Konstruktor

```
RegEx(const string pattern = null, const 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[82]), ‚`i`' (ignore case) und ‚`g`' (global search) enthalten; alle anderen ggf. vorhandenen Zeichen werden ignoriert.

Methode compile

```
bool compile(const string pattern, const 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`, sonst `false`. 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 das neue ersetzt.

[82] Das ‚`m`'-Flag bewirkt, dass sich die Assertions `^` und `$` (vgl. Tabelle 20) in mehrzeiligen Texten auf einzelne Zeilen statt auf den gesamten Text beziehen.

- 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(const 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(const 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` im Fall einer Übereinstimmung, sonst `false`.

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 86 illustriert die Verwendung von `match` zur Überprüfung einer Mail-Adresse.

```
// checks if adr is a valid mail address
bool checkMailAdress(string adr)
{
        RegEx rex("\\w+[\\w\\.]*\\w+@\\w+\\.\\a+");
        return rex->match(adr);
}
```

Beispiel 86 Verwenden der Methode `match`

Methode search

`bool search(const string txt, uint start=0, uint len=NPOS)`

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; reicht *len* über das Ende von *txt* hinaus, so wird bis zum Ende von *txt* gesucht.

Rückgabewert:

`true` im Fall einer Übereinstimmung, sonst `false`.

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 87 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 87 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 87.

Rückgabewert:

`true` falls eine weitere Übereinstimmung existiert, sonst `false`.

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(const string txt, vector dst, uint start=0,
                uint len=NPOS)
```

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; reicht *len* 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 nach dem 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(const string txt, vector dst, uint start=0,
                       uint len=NPOS)
```

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; reicht *`len`* 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 88 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. 33). In Zeile 9 wird der Punkt-Operator für den Zugriff auf die Dictionary-Einträge der einzelnen Ergebnisse verwendet (vgl. Abschnitt 6.7, S. 53).

```
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 88 Bestimmen von Submatches mit `getSubExpressions`

Methode getSubExpressions (Variante 2)

```
uint getSubExpressions(const string txt, vector dst, uint start=0,
                       uint len=NPOS)
```

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`* 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; reicht *`len`* 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 89 verwendet die zweite Variante von `getSubExpressions` für die gleiche Aufgabe wie in Beispiel 88 und benutzt `foreach` (siehe 13.1.2, S. 109) mit anonymen Funktionen (vgl. 9.8, S. 71) 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 89 Verwenden der zweiten Variante `getSubExpressions`

Methode replace

`string replace(const string` *`txt`*`, const 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, const 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.

14.2.2 Operatoren

Funktionsoperator

```
bool operator()(const string txt, uint start=0, uint len=NPOS)
```

bzw.

```
bool operator()()
```

Der Operator ist ein Alias für die Methode search (Variante mit Argumenten) bzw. next (parameterlose Form) und dient der syntaktischen 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; reicht *`len`* über das Ende von *`txt`* hinaus, so wird bis zum Ende von *`txt`* gesucht.

Rückgabewert:

`true` falls eine (weitere) Übereinstimmung existiert, sonst `false`.

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 90 modifiziert den Code aus Beispiel 87 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 90 Iteration über Treffer mit dem Funktionsoperator

Property-Operator (Punktoperator)

`var operator.(const string key)` const (Lesezugriff)

bzw.

`var operator.=(const 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 25 zusammengestellt.

Lesezugriff		
Schlüssel (key)	**Rückgabetyp**	**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`	`uint`	Länge des Suchbereichs
Schreibzugriff		
Schlüssel (key)	**Typ**	**Beschreibung**
`flags`	`string`	Setzen aller als Zeichenkette (wie `setFlags`)
`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`	`uint`	Setzen der Länge des Suchbereichs

Tabelle 25 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.

15 Emulation von C-Strukturen

Häufig werden beim Aufruf von Betriebssystemfunktionen Verweise auf C-Datenstrukturen als Argumente übergeben. Weil B++ keine zu C kompatiblen Datenstrukturen kennt, müssen sie bei Bedarf emuliert werden, typischerweise mit Hilfe eines Puffers.
Die korrekte Abbildung der Strukturdaten auf einen linearen Puffer kann kompliziert und fehleranfällig sein, weil hierbei neben der korrekten Abbildung verschiedener Datentypen Regeln für die Anordnung der Strukturelemente innerhalb der Struktur (alignment) beachtet werden müssen, die u. U. Bestimmte Zwischenräume zwischen benachbarten Elementen (padding) erfordern.

Um dies zu vereinfachen, wurde mit Version 1.3 eine vordefinierte abstrakte Klasse **`CStruct`** eingeführt, die die Emulation C-kompatibler Strukturen mit Hilfe einer einfachen Beschreibung erlaubt. Die Klasse realisiert die Abbildung einfacher C-kompatibler Strukturen (POD struct), einschließlich eingebetteter Strukturen, unter Berücksichtigung der Standard-Ausrichtungsregeln auf einen linearen Puffer. Dies bedeutet, dass einfache Datentypen auf ein Vielfaches ihrer Typgröße und eingebettete (Sub-)Strukturen auf ein Vielfaches der Größe ihres größten einfachen Einzelelements ausgerichtet werden.
Dabei gehören zu den ‚einfachen' Elementen alle numerischen Typen, was die Typen `intptr` und `uintptr` zur Abbildung von Adressen (Zeigern) einschließt, sowie Zeichentypen. Als eingebettete strukturierte Elemente werden wiederum Strukturen und eindimensionale Arrays fester Größe unterstützt. Elemente der eingebetteten Arrays können von einem einfachen Typ oder vom Strukturtyp, nicht aber selbst Arrays sein.

Ab Version 1.5 gibt es Erweiterungen, die die Konstruktion gepackter Strukturen (siehe 15.2), das Einbetten von Unions (15.3) in Strukturen sowie Felder variabler Länge in Strukturen (15.5) erlauben.

15.1 *Einfache Strukturen*

Um eine C-kompatible Struktur zu erzeugen, ist eine konkrete Klasse von `CStruct` abzuleiten, die mindestens einen Konstruktor definiert und die Methode

```
dictionary getStructDesc()
```

implementiert.

Der Konstruktor muss die von `CStruct` geerbte Methode

```
void create(const dictionary data = null, bool isdynamic = false)
```

aufrufen. Die Methode `getStructDesc` muss eine Beschreibung der Struktur als `dictionary` mit folgenden Eigenschaften als Ergebnis liefern:

- Jeder Eintrag der Beschreibungsstruktur beschreibt ein Element der zu emulierenden Struktur, wobei die hier angegebene Reihenfolge auch die Reihenfolge der Strukturelemente bestimmt.
- Der Name jedes Eintrags muss ein gültiger B++-Bezeichner und innerhalb einer Strukturebene eindeutig sein.
- Der Wert eines Eintrags ist entweder der interne Typidentifikator eines numerischen Datentyps (vgl. Tabelle 2, S. 22) oder ein `dictionary` mit der Beschreibung einer eingebetteten Struktur oder ein `vector` aus zwei Elementen für eingebettete Arrays, wobei dann der erste Wert die Anzahl der Elemente und der zweite wiederum deren Typ beschreibt.

Beispiel 91 veranschaulicht das Prinzip anhand einer einfachen Struktur für ein Rechteck, das durch die Positionskoordinaten *x* und *y* sowie eine Ausdehnung *ext* beschrieben wird, die ihrerseits die Elemente *w* (Breite) und *h* (Höhe) enthält.

```
class Rect : CStruct
{
public:
   Rect(dictionary data = null) {
        create(data);
      }
protected:
   dictionary getStructDesc() {
      return {
            x : BVT_INT,
            y : BVT_INT,
            ext : {
               w : BVT_UINT,
               h : BVT_UINT
            }
         };
      }
};
```

Beispiel 91 Definition einer einfachen Datenstruktur

Zur Initialisierung einer Struktur gibt es drei Möglichkeiten:

- Datenübergabe als Argument des Konstruktors,
- Aufruf der Methode `applyData` mit Initialisierungsdaten als Argument oder
- durch explizite Wertzuweisung an an einzelne Strukturelemente.

Der Zugriff auf Elemente einer Struktur erfolgt in ‚Punktnotation', also mit Hilfe des Punkt-Operators.

Beispiel 92 füllt eine Rect-Struktur aus Beispiel 91 mit Hilfe des Konstruktors, modifiziert Werte und gibt das Resultat auf die Konsole aus:

```
void main() {
      Rect r({ x : 10, y : 10, ext : { w : 20, h : 30}});
      r.x *= -1;
      r.ext.w += 5;
      print("x:", r.x, " y:", r.y, " w:", r.ext.w, " h:", r.ext.h,"\n");
}
```

Beispiel 92 Verwenden einer Struktur

Beispiel 93 zeigt die Nachbildung der `PAINTSTRUCT`-Struktur aus dem Windows-API. Hier wird zunächst eine Struktur `RECT` definiert (ebenfalls eine Nachbildung der gleichnamigen Windows-Struktur), die Bestandteil von `PAINTSTRUCT` ist.

Weil Strukturbeschreibungen aber keine eingebetteten Objekte, sondern lediglich eingebettete Dictionarys erlauben, wird hier die Beschreibung von `RECT` (`DescRect`) als benanntes Literal definiert, das sowohl für `RECT` als auch als Bestandteil der Beschreibung von `PAINTSTRUCT`, hier als Feld `rcPaint`, verwendet wird.

Das Feld `rgbReserved` in `PAINTSTRUCT` schließlich ist ein eingebettetes Array, beschrieben durch die Anzahl seiner Elemente und deren Typ.

```
#define DescRECT { left : BVT_INT, top: BVT_INT, right : BVT_INT, bottom :
BVT_INT }

class RECT: CStruct
{
public:
   RECT(dictionary data = null) {
      create(data);
   }
protected:
   dictionary getStructDesc() {
      static dictionary desc = DescRECT;

      return desc;
   }
};

class PAINTSTRUCT: CStruct
{
public:
   PAINTSTRUCT(dictionary data = null) {
      create(data);
   }
protected:
   dictionary getStructDesc() {
      static dictionary desc = {
         hdc : BVT_INTPTR,
         fErase : BVT_INT,
         rcPaint : DescRECT,
         fRestore : BVT_INT,
         fIncUpdate : BVT_INT,
         rgbReserved : [32,BVT_BYTE]
      };
      return desc;
  }
};
```

Beispiel 93 Nachbildung der Windows-`PAINTSTRUCT`

15.2 *Gepackte Strukturen*

Mitunter erfordern Systemfunktionen, z. B. einige Windows-API-Funktionen, ‚gepackte' Datenstrukturen, bei denen von der Standardausrichtung abgewichen wird. In C/C++ werden hierfür Attribute bei der Deklaration oder Compileranweisungen (#pragma pack) verwendet.
Zur Emulation des entsprechenden Verhaltens kann die Methode

```
uint getStructPacking() const
```

überschrieben werden. Gibt sie 0 zurück, bedeutet dies die Verwendung der Standardausrichtung (Ausrichtung der Elemente auf einem Vielfachen ihrer eigenen Größe), andernfalls die Ausrichtung an einem Vielfachen des Rückgabewerts (in Bytes). Zur Kompatibilität mit C/C++ sollte die Methode einen der Werte 0, 1, 2, 4, 8 oder 16 zurückgeben.

Beispiel 94 demonstriert das Vorgehen anhand der Windows-Struktur BITMAPFILEHEADER.

```
class BITMAPFILEHEADER : CStruct
  {
     public:
        BITMAPFILEHEADER(dictionary data = null) {
             create(data);
        }
     protected:
          // define WORD alignment
        uint getStructPacking() const { return 2; }

        dictionary getStructDesc() const {
           static dictionary desc = {
              bfType : BVT_USHORT,
              bfSize : BVT_UINT,
              bfReserved1 : BVT_USHORT,
              bfReserved2 : BVT_USHORT,
              bfOffBits : BVT_UINT
           };
           return desc;
        }
  };
```

Beispiel 94 Emulation der Windows-Struktur `BITMAPFILEHEADER`

Das ‚Packen' der Struktur sorgt hier dafür, dass die Elemente `bfSize` und `bfOffBits` am Offset 2 bzw. 10 statt 4 bzw. 12 beginnen.

15.3 *Unions*

Unions dienen gewöhnlich der Abbildung von Varianten, also Werten, die kontextabhängig unterschiedlichen Typs sein können. Sie sind gewissermaßen eine Sonderform einer Struktur, bei der alle Elemente an derselben Adresse beginnen, also denselben Speicherplatz belegen.

Ihre Emulation in `CStruct` (bzw. Ableitungen davon) erfolgt, indem man ein Strukturelement wie ein Array definiert, wobei die Anzahl der Feldelemente mit Null angegeben wird und der Typ wiederum ein Typdeskriptor (dictionary) ist, der die Elemente der Union beschreibt.

In Beispiel 95 wird das Prinzip am Beispiel der Windows-Struktur `DISPLAYCONFIG_PATH_SOURCE_INFO` gezeigt.

```
// define symbolic name for union definition
#literal UNION 0
// define dummy union name
#literal DUMMYUNIONNAME "-u"
// define dummy structure name
#literal DUMMYSTRUCTNAME "-s"
// define literal for Descriptor of LUID
#literal DescLUID {LowPart : BVT_UINT, HighPart : BVT_INT }

class DISPLAYCONFIG_PATH_SOURCE_INFO : CStruct
{
 public:
    DISPLAYCONFIG_PATH_SOURCE_INFO(dictionary data = null) {
       create(data);
    }
 protected:
    dictionary getStructDesc() const {
      static dictionary desc = {
         adapterId : DescLUID,
         id : BVT_UINT,
         // define union
         DUMMYUNIONNAME : [UNION, {
            modeInfoIdx : BVT_UINT,
            DUMMYSTRUCTNAME : {
               cloneGroupId : BVT_USHORT,
               sourceModeInfoIdx : BVT_USHORT
            }
         }],
         statusFlags : BVT_UINT
      };
      return desc;
   }
};
```

Beispiel 95 Emulation einer Union in der Struktur `DISPLAYCONFIG_PATH_SOURCE_INFO`

Hier wird einleitend außerdem ein Deskriptor für den Typ `LUID` (als eingebettete Struktur `adapterId` verwendet) als benanntes Literal definiert. Zudem wird für den Wert `0` zur Kennzeichnung einer Union der besseren Lesbarkeit halber das Symbol `UNION` definiert. Dabei wird die Definition mit `#literal` nur dann wirksam, wenn ein entsprechender symbolischer Name nicht schon existiert (vgl. Abschnitt 4.5.4).

15.4 *Anonyme Strukturen und Unions*

In C/C++ ist es möglich, als Elemente einer Struktur unbenannte Unterstrukturen zu definieren, auf deren Elemente so zugegriffen wird, als gehörten sie direkt zur übergeordneten Struktur.

Dieses Verhalten lässt sich nachbilden, indem man dem Namen der betreffenden Substruktur im Strukturdeskriptor ein Minuszeichen (-) voranstellt.

In Beispiel 95 wird dies bei der eingebetteten Union `DUMMYUNION` und der darin eingenbetten Struktur `DUMMYSTRUCT` verwendet. `DUMMYUNION` und `DUMMYSTRUCT` sind benannte Literale mit den Werten „-u“ bzw. „-s“.

15.5 *Felder variabler Länge in Strukturen*

Normalerweise hat eine Struktur in C eine feste Größe, die sich aus ihrer Definition ergibt. Jedoch werden C-Strukturen mitunter auch für die Übergabe von Daten variabler Größe verwendet. Üblicherweise geschieht dies, indem als letztes Strukturelement ein Array mit einem Element definiert wird. Zudem gibt es gewöhnlich ein weiteres Strukturelement, das die tatsächliche Größe dieses Arrays angibt. Der Speicher für die Struktur wird dann dynamisch entsprechend größer alloziert.

```
typedef struct tagEXTLOGPEN {
    DWORD       elpPenStyle;
    DWORD       elpWidth;
    UINT        elpBrushStyle;
    COLORREF    elpColor;
    ULONG_PTR   elpHatch;
    DWORD       elpNumEntries;
    DWORD       elpStyleEntry[1];
} EXTLOGPEN;
```

Beispiel 96 Definition der Struktur EXTLOGPEN in wingdi.h

Beispiel 96 zeigt dies anhand der EXTLOGPEN-Struktur aus der Windows-GDI-Bilbliothek.

In B++ lässt sich dieses Verhalten nachbilden, indem man das Verhalten der `getStructDesc`-Methode entsprechend modifiziert.

```
   class EXTLOGPEN : CStruct
   {
    private:
      int _numentries = 1;
    public:
       EXTLOGPEN(dictionary data = null, int numentries = 1) {
          if (data && dictcontains(data,"elpStyleEntry"))
             _numentries = size(data.elpStyleEntry);
          else
          _numentries = numentries;
          create(data,true);
       }
    protected:
         dictionary getStructDesc() const {
            static dictionary desc = {
              elpPenStyle : BVT_UINT,
              elpWidth : BVT_UINT,
              elpBrushStyle : BVT_UINT,
              elpColor : BVT_UINT,
              elpHatch : BVT_UINTPTR,
              elpNumEntries : BVT_UINT,
              elpStyleEntry : [1,BVT_UINT] //  DWORD elpStyleEntry[1]
            };
            if (_numentries > 1) {
               auto res = clone(desc);
               res.elpStyleEntry[0] = _numentries;
               return res;
            }
            return desc;
          }
   };
```

Beispiel 97 Nachbildung von EXTLOGPEN

In Beispiel 97 wird hierzu die Anzahl der Einträge des Feldes `elpStyleEntry` im Konstruktor entweder aus der Anzahl der im Argument `data` im Feld „elpStyleEntry“ übergebenen Werte bestimmt oder alternativ dem Wert des Arguments `numentries` entnommen und in der Member-Variablen :`_numentries` zwischengespeichert.

Die Methode `create` wird im Konstruktor Argument `isdynamic=`*`true`* aufgerufen, was bewirkt, dass kein Caching des von `getStructDesc` zurückgegebenen Deskriptors erfolgt.

Die Methode `getStructDesc` prüft ihrerseits, ob die Anzahl der geforderten Elemente vom Standard (1) abweicht und erzeugt ggf. eine modifizierte Kopie des Deskriptors.

15.6 *Makros zur vereinfachten Definition von Strukturen und C-Arrays*

Zur Vereinfachung der Definition von Strukturen und C-Arrays werden ab Version 1.5 im Header `cstruct.bph` einige Makros bereitgestellt, die direkt von `CStruct` abgeleitete Klassen erzeugen. Im Einzelnen sind dies:

- `CSTRUCT(NAME,DESC)` : C-Struktur mit Standardausrichtung der Elemente und fester Größe
- `PACKEDCSTRUCT(NAME, DESC, PACKING)` : C-Struktur fester Größe mit benutzerdefinierter Ausrichtung der Elemente
- `DYNAMICCSTRUCT(NAME, DESC, ARRKEY)` : C-Struktur mit einem Feld variabler Größe
- `CARRAY_T(NAME, VDESC, T) : C-Array mit festem (einfachem) Elementtyp und bei der Instantiierung anzugebender Größe (Elementzahl)`
- `CARRAY(NAME, VDESC)` : C-Array mit bei der Instantiierung anzugebender Größe (Elementzahl)

Eine detaillierte Beschreibung der Verwendung ist der Referenzdokumentation zu entnehmen.

16 Speicherverwaltung

In Abschnitt 5.3 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 13.3) angefordert bzw. freigegeben wird.

Für alle anderen Referenztypen verwendet B++ standardmäßig eine automatische Speicherverwaltung mit Referenzzählung[83]. 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 98 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 98 Objektfreigabe beim Verlassen des Gültigkeitsbereichs

[83] 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 98 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.

16.1 *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:

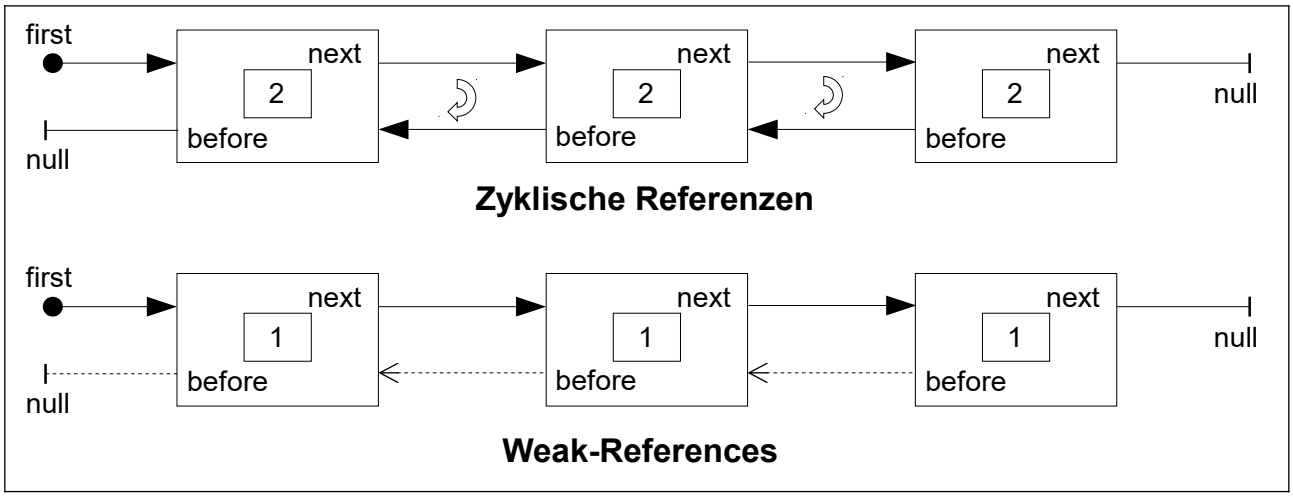


Abbildung 3 Zyklische Referenzen und deren Auflösung

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 99 demonstriert die praktische Anwendung von Weak References etwas ausführlicher.

```
// Double linked list class example
// --------------------------------

// List class
class List
{
   // List element class
   class element
   {
   private:
      var _prev = null; // Weak reference to previous element
      element _next = null; // Reference to next element
      var _value = null; // List elements value
   public:
      // ctor
      element(var v = null; ) { _value = v; }
      // Definition of 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 iterator
   {
   private:
      var _lst = null; // Weak reference to list
      var _item = null; // Weak reference to list item
   public:
      // ctor
      iterator(List lst, var item = null) { _lst = &lst; _item = &item; }
      // Increment
      iterator operator++() {
         if (_item) _item = &_item.next;
         else _item = &_lst->first();
         return this;
      }
      // Post-Increment
      iterator operator++(int) {
         iterator it(_lst, _item);
         return ++it;
      }
      // Decrement
      iterator operator--() {
         if (_item) _item = &_item.prev;
         else _item = &_lst->back();
         return this;
      }
      // Post-Decrement
      iterator operator--(int) {
         iterator it(_lst, _item);
         return --it;
      }
      var item() { return _item; }
      var lst() { return _lst; }

      // Equality operator
      bool equals(iterator other)
      {
         if (_lst != other._lst) return false;
         return _item == other._item;
      }
      // Gets referenced list item.
      element operator*() { return *_item; }
      // Delegates operator. to referenced list element
      var operator.(string key) { return _item->(element::OP_PREF)(key); }
      void operator. = (string key, var val){
        _item->(element::OP_PSET)(key,val);
      }
   };

private:
   element _first = null;
   var _last = null;
   uint _size = 0;
   // Internal helpers
   // Initialization
   void _init(var first, var last, uint siz) {
      _first = first; _last = last; _size = siz;
   }
   // Removes an element.
   element _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
   element _insert(var val, var beforeEl) {
      element el(val);
      if (!beforeEl)
      {
         if (_last)
         {
            el.prev = _last;
            _last.next = el;
         }
         else _first = el;
         _last = &el;
      }
      else {
         if (!_size) throw "List is empty";
         print("#insert ", val, "\n");
         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
   element front() { return _first; }
   element back() { return _last ? *_last : null; }
   iterator begin() { return new iterator(this, _first); }
   iterator end() { return new iterator(this); }
   void push_back(var val) { this->(_insert)(val, null); }
   void push_front(var val) { this->(_insert)(val, _first); }
   void pop_back() { this->(_remove)(_last); }
   void pop_front() { this->(_remove)(_first); }
   iterator insert(iterator before, var val) {
      if (before->(iterator::lst)() != this) throw "Invalid iterator";
      return new iterator(this,
         this->(_insert)(val, before->(iterator::item)()));
   }
```

```
   iterator erase(iterator first, iterator last = null) {
      if (first->(iterator::lst)() != this) throw "Invalid iterator";
      if (!last) return new iterator(this,
         this->(_remove)(first->(iterator::item)()));
      if (last->(iterator::lst)() != this) throw "Invalid iterator";
      auto itend = end();
      while (first != itend) {
         auto item = *first;
         ++first;
         this->(_remove)(item);
         if (first == last) break;
      }
      return new 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");
   auto i = l->begin();
   ++i;
   auto j = clone(i);
   ++j;

   l->erase(i, ++j);

   for (auto 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");
   print("--------------\n");
   l->swap(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");
```

```
   List l3 = clone(l2);

   l2 = null;
   print("reverse:\n");
   for (auto itr = l3->end(); (--itr) != l3->end();)
      print(*itr.value, " ");
   print("\n");
}
```

Beispiel 99 Implementierung einer doppelt verketteten Liste

Hier wird eine doppelt verkettete Liste in Anlehnung an die C++-Standardbibliothek implementiert[84]. 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.

16.2 *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.

[84] Ab Version 1.5 von B++ könnte man das Beispiel eleganter formulieren, indem die Klasse `ListElement` sowie die Iteratoren als lokale Klassen von `List` definiert werden.

17 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 die Realisierung von Erweiterungen an B++ (siehe hierzu Kapitel 18) erleichtern.

17.1 *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[85], 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 5.6 und 10.11.10.

Die Einträge in den Symboltabellen bestehen wesentlich aus dem *Symbolbezeichner*, dem *Symboltyp* sowie einem *Wert*. Tabelle 26 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 26 Übersicht der Symboltypen

Die Symboltabellen sind auf Basis von Dictionarys 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.

17.2 *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 Member-Variablen. Kopien der Standardwerte werden zur Initialisierung der Member-Variablen beim Erzeugen einer neuen Objektinstanz verwendet (vgl. Abbildung 4).

[85] Klassen sind in B++ die einzige Form benutzerdefinierter Datentypen. Die Tabelle lässt sich also auch als Tabelle aller benutzerdefinierten Typen auffassen.

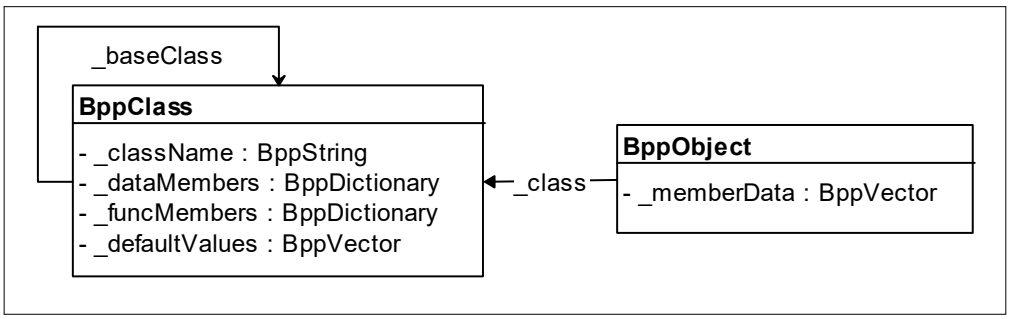


Abbildung 4 Struktur von Klassen und Objekten

In der `_dataMembers`-Tabelle bestehen die Werte der Einträge für Member-Variablen 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 Member-Variablen aufnimmt. Der `_defaultValues`-Vektor wird vom Compiler nach Abschluss der Übersetzung der Klassendeklaration (nachdem alle Member-Variablen 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) Member-Variablen. Beim Kopieren eines Objekts werden Klassenverweis und Member-Variablen (letztere als tiefe Kopie) aus dem Quellobjekt übernommen[86].

17.3 *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 Einsprung- und Rückkehradressen für Funktionsaufrufe.

Intern wird eine Funktion[87] in Form eines Vektors abgebildet, der jedoch die Typ-ID `BVT_BYTECODE` trägt (vgl. Tabelle 2 auf Seite 22). Der Aufbau dieses Vektors ist in Tabelle 27 zusammengefasst.

[86]Der 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 compilierter Bytecode-Module gewährleistet.
[87]Gemeint ist hier eine Bytecode-Funktion; externe „native“ Funktionen werden einfach durch einen Zeiger auf ihren Code repräsentiert.

<table>
<tr><th>Index</th><th>Symbol[88]</th><th>Beschreibung</th></tr>
<tr><td>0</td><td>IDX_CODE</td><td>Bytecode-Puffer (BppBuffer)</td></tr>
<tr><td>1</td><td>IDX_CLASS</td><td>Klassenverweis (BppClass) bei Elementfunktionen, sonst null</td></tr>
<tr><td>2</td><td>IDX_FUNCSTATE</td><td>Speicherplatz für den zwischengespeicherten Aufrufzustand einer Coroutine mit folgender Struktur (BppVector):
<table>
<tr><th>Index</th><th>Beschreibung</th></tr>
<tr><td>0</td><td>PC (Bytecode-Index des nächsten Befehls)</td></tr>
<tr><td>1</td><td>Vektor der Werte der Aufrufargumente</td></tr>
<tr><td>2</td><td>Vektor der Werte der temporären Variablen</td></tr>
</table>
oder null</td></tr>
<tr><td>3</td><td>IDX_NAME</td><td>Funktionsname</td></tr>
<tr><td>4</td><td>IDX_DEFAULTARGS</td><td>Vektor mit Standard-Argumentwerten</td></tr>
<tr><td>5</td><td>IDX_TEMPTYPES</td><td>Vektor mit Definitionen der nicht statischen lokalen Variablen</td></tr>
<tr><td>6</td><td>IDX_SELFREFERENCE</td><td>Rückverweis (BVT_BYTECODEREF) auf die Funktion selbst</td></tr>
<tr><td>...</td><td></td><td>Werte aller innerhalb der Funktion definierten Literale und lokalen statischen Variablen sowie Verweise auf innerhalb der Funktion referenzierte globale Variablen</td></tr>
<tr><td>(last)</td><td></td><td>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></td></tr>
</table>

Tabelle 27 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 Vektor der Definitionen der lokalen Variablen (IDX_TEMPTYPES) enthält deren statische Typen (sofern festgelegt) sowie ggf. eine Kennzeichnung als Konstante.

Der Rückverweis (IDX_SELFREFERENCE) wird intern für den rekursiven Aufruf anonymer Funktionen mittels self verwendet (siehe Abschnitt 9.8).

Der Aufrufzustand einer Coroutine (siehe 9.9) wird bei IDX_FUNCSTATE zwischengespeichert, wenn die Coroutine mit yield verlassen wird. Beim nächsten Aufruf der Corouitine wird der gespeicherte Zustand wiederhergestellt und der Eintrag an IDX_FUNCSTATE auf null gesetzt.

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.

[88]Die 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. IDX_FUNCSTATE existiert ab Version 1.6; in den früheren Versionen ist der Wert (Index) der nachfolgenden Symbole entsprechend um 1 niedriger. IDX_TEMPTYPES existiert ab Version 1.8.

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 Datenwerts 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).

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 werden[89]. Von dieser Erweiterung macht B++ allerdings nur für wenige spezielle Befehle Gebrauch, die normalerweise selten verwendet werden[90].

Tabelle 28 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 27
- nargs : Anzahl der Argumente einer Funktion
- nvar : Anzahl lokaler Variablen einer Funktion
- Index : Indexwert
- MDATA : Vektor der Elementvariablen eines Objekts
- flags : Flags eines Wertes

Die in der Spalte „Symbol“ mit `*` gekennzeichneten Einträge bezeichnen Befehle, die für Objekte mit Hilfe gleichnamiger Elementfunktionen (Operatorfunktionen) explizit definiert werden können (vgl. Abschnitt 10.11). Mit `**` gekennzeichnete Befehle (Vergleichsoperationen) können für Objekte indirekt mit Hilfe der Elementfunktionen `equals` bzw. `compare` definiert werden (siehe 10.11.8).

[89]256 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.

[90]Das 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
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
06	OP_NOT*		Logische Negation des Wertes bei SP: wenn [SP] == true → [SP] = false, sonst [SP] := true
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]

Code (hex)	Symbol	Daten	Beschreibung
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] → [SP-1] := true, sonst [SP-1] := false --SP
14	OP_LE**		Vergleich der obersten beiden Werte des Stacks (kleiner / gleich) wenn [SP-1] <= [SP] → [SP-1] := true, sonst [SP-1] := false --SP
15	OP_EQ**		Vergleich der obersten beiden Werte des Stacks (Gleichheit) wenn [SP-1] == [SP] → [SP-1] := true, sonst [SP-1] := false --SP
16	OP_NE**		Vergleich der obersten beiden Werte des Stacks (Ungleichheit) wenn [SP-1] != [SP] → [SP-1] := true, sonst [SP-1] := 0 --SP
17	OP_GE**		Vergleich der obersten beiden Werte des Stacks (größer / gleich) wenn [SP-1] >= [SP] → [SP-1] := true, sonst [SP-1] := 0 --SP
18	OP_GT**		Vergleich der obersten beiden Werte des Stacks (größer) wenn [SP-1] > [SP] → [SP-1] := true, sonst [SP-1] := false --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]

Code (hex)	Symbol	Daten	Beschreibung
1D	`OP_CALL*`	nargs	Funktionsaufruf (nargs ist Argumentzahl): VP := SP-nargs Aufruf der Funktion bei [VP] [VP] := [SP] SP := VP oder Wiedereintritt in eine Coroutine (falls Wert an IDX_FUNCSTATE nicht `null`): Verhalten zunächst wie bei einfachem Aufruf, dann Einsetzen gespeicherter Argumentwerte als Vorgabe-werte Wiederherstellen der temporären Variablen Löschen des Wertes an IDX_FUNCSTATE PC := gespeicherter Index
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/initialisieren: FP := SP SP := SP+nvar
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])

Code (hex)	Symbol	Daten	Beschreibung
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
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
3E	OP_STOS		Zwischenspeichern von [SP][91]
3F	OP_LDTOS		Restaurieren von [SP] aus gespeichertem Wert

[91]Der Wert wird temporär in einem funktionslokalen Puffer abgelegt. Die Opcodes OP_STOS und OP_LDTOS werden als Hilfskonstrukt zur Realisierung der Postfix-Inkrement- und Dekrementoperatoren verwendet.

Code (hex)	Symbol	Daten	Beschreibung
Erweiterte Befehle			
40	`OP_MKCONST`	Index	Temporäre Variable in Konstante umwandeln: [FP+Index+1].flags |= CONST
41	`OP_SPCONST`		Wert an TOS in Konstante umwandeln: [SP].flags |= CONST
42	`OP_YIELD`	(TYPE)	Rückkehr aus Coroutine, ggf. mit Typprüfung des Rückgabewertes [SP]. Temporäre Variablen, Parameterwerte und Index des nächsten Befehls werden am Index IDX_FUNCSTATE im Bytecode-Vektor zwischengespeichert.
FC	`OP_TRON`		Trace On
FD	`OP_TROFF`		Trace Off
FE	`OP_TRSTEP`		Einzelschrittbetrieb
FF	`OP_NEWLINE`		Neue Zeile im Quelltext

Tabelle 28 Übersicht der Opcodes der B++-VM

Hinweis:

Die in Tabelle 28 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.

17.4 *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 Funktion[92].
- Es wird eine Hilfsstruktur (`BppProcInfo`[93]) 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 29 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 29 Stack-Frame beim Eintritt in eine Funktion

[92]Der 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.
[93]Siehe hierzu die Referenz.

18 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 entgegengestanden. 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.

18.1 *B++-Bibliotheksmodule*

Die einfachste Möglichkeit, wiederverwendbare Bibliotheken für spezifische Aufgaben bereitzustellen, ist deren direkte Implementierung in B++. Solche Bibliotheken können ihrerseits Systembibliotheken von Windows[94] 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 100 zeigt einen Ausschnitt aus einem Bibliotheksmodul zur Dateiverwaltung[95].

```
// filesys.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(const string dirname); ///< Changes working directory.
   string getcwd(); ///< Gets current working directory.
   bool mkdir(const string dirname); ///< Creates new directory.
   bool rmdir(const string dirname); ///< Removes a directory.
   bool remove(const string fname); //< Removes file.
   bool copyfile(const string src, const string dst,
                 bool overwrite = true); ///< Copies file
   bool rename(const string oldname,
               const 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();
```

[94] Dies gilt prinzipiell auch für LINUX-/UNIX-Versionen.
[95] 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(const 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 100 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.

18.2 *Erweitern von B++ mit dynamischen Bibliotheken*

Wenngleich es mit den von B++ bereitgestellten Mitteln grundsätzlich möglich sein sollte, nahezu jede benötigte Funktionalität 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) implementiert[96], wobei zwischen *allgemeinen Funktionsbibliotheken* und *B++-spezifischen Bibliotheken* zu unterscheiden ist.

18.2.1 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 13.7.

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 16- und 32-Bit-Plattformen bzw. 64 Bit auf 64-Bit--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 18.2.3 behandelt.

18.2.2 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.

[96] 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` gelinkt[97].

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 101 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 101 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

[97] 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).

18.2.2.1 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 102 illustriert die Implementierung anhand einer Funktion `copyfile`, die ihrem Zweck nach der Methode `FileSys::copyfile` aus Beispiel 100 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 variable
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_BOOL,_T("bool"))) return false;
24       overwrite = v.asBool();
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 = BppValue::makeBool(res != FALSE);
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 102 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 29). 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 ‚von 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, das `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 und zulässig, 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 langsameres Verhalten zur Laufzeit.

18.2.2.2 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 17.2 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 30.

Methode	Beschreibung
`bool addDataMember(const BppChar* name)`	Hinzufügen einer Elementvariablen
`bool addDataMember(const BppChar* name, const BppValue& initValue)`	Hinzufügen einer Elementvariablen mit Initialwert
`bool addStaticDataMember(const BppChar* name)`	Hinzufügen einer Klassenvariablen
`bool addStaticDataMember(const BppChar* name, const BppValue& initValue)`	Hinzufügen einer Klassenvariablen mit Initialwert
`bool addMethod(const BppChar* name, BPPUSERFUNC pfunc, unsigned char flags = 0)`	Hinzufügen einer Elementfunktion (Methode)
`bool addStaticMethod(const BppChar* name, BPPUSERFUNC pfunc, unsigned char flags = 0)`	Hinzufügen einer Klassenfunktion

Tabelle 30 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 103 Klasse String (B++-Pseudodeklaration)

Beispiel 103 zeigt die Pseudodeklaration der Klasse `String` aus der Bibliothek `bp2class`, die ein Zeichenkettenobjekt bereitstellt. In Beispiel 104 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 104 Implementierung der Klasse String (Auszug)

Der Prototyp der Implementierungsfunktionen für Methoden von Klassen entspricht dem für gewöhnliche Funktionen (siehe Abschnitt 18.2.2.1). Zur Implementierung von Methoden sind im Header `bppfimpl.h` folgende weitere 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 104 übernimmt den Zusammenbau der Klasse und wird von der Registrierungsfunktion der Bibliothek (vgl. Beispiel 101) 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.

18.2.3 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 sein[98], 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 „von Hand" aufgebaut werden.

Wie in [4] beschrieben, müssen DOS-DLLs mit dem Speichermodell LARGE übersetzt werden.

[98] 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 Assembler[99] demonstriert. Die Beispielbibliothek soll den Namen `SYSTEM.EXE`[100] tragen und folgende Funktionen bereitstellen[101]:

- `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 engegebenen 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 13.7) 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 Werttypen von B++ werden beim Aufruf implizit nach `intptr` umgewandelt. Die Adresse des Datenbereichs von Referenztypen kann mit Hilfe der Standardfunktion `addr` bestimmt werden.

18.2.3.1 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-Standardbibliothek[102]. Dadurch fällt der Quelltext vergleichsweise kurz aus. Er ist nachfolgend (Beispiel 105) 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;
```

[99] Hier wird der Assembler A86 (Version 3.22) verwendet.
[100] 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.
[101] Die Signaturen sind in C++-Syntax angegeben.
[102] 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

#if __TURBOC__ >= 0x0300
  static char dummybuf[255];
#endif


int main(int,char**)
{
      puts("Can't run standalone.\n");
      return 1;
}
```

Beispiel 105 Implementierung einer DLL-Funktionsbibliothek mit Turbo-C++

Die `_NOFLOAT_`-Definition am Anfang des Quelltextes verspricht, dass keinerlei Unterstützung für Gleitpunkt-Artihmetik 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 sind dennoch angebracht:

- Die `huge`-Modifier in den Funktionsköpfen erzwingen FAR-Aurufe der Funktionen auch dann, wenn die Bibliothek im COMPACT-Modell übersetzt wird[103].
- 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 gewährleisten[104]. 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.

18.2.3.2 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 länger[105] – dafür wird man aber durch eine wesentlich kleinere DLL entschädigt.

[103] Es würde hier eigentlich auch der Modifier `far` genügen, `huge` entspricht aber der Deklaration des Funktions-Prototypen in `dosdll.h`.

[104] 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.

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 Meldungsausgabe, 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 verwendet[106].

```
; 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
```

[105] Deshalb werden hier auch nur einige Auszüge angegeben. Die vollständigen Quellen liegen der B++-Distribution bei.

[106] Genau genommen ist `searchPath` lediglich ein Spezialfall von `searchEnv`.

```
           DW searchPath
           DW SEG CSEG
;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 106 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 bestimmen[107]. 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:

[107] Zweifellos ist dies nicht besonders elegant. Eine schönere Lösung ist aber ohne zusätzliche Anforderungen an den internen Aufbau der DLL kaum denkbar.

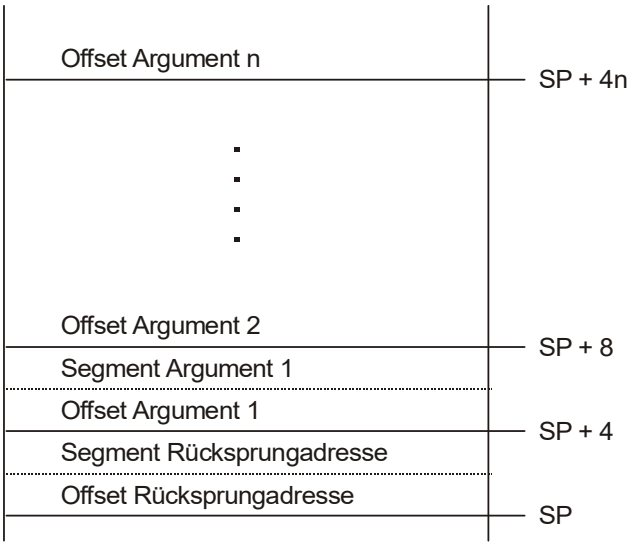


Abbildung 5 Aufbau des Stacks beim Eintritt in eine DOS-DLL-Funktion

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 niederwerige 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 wird[108]. Beispiel 107 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 bock
       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 107 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:

```
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 107 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.

[108] Genau genommen wird sie auch beim Entladen der DLL sowie bei jeder Änderung der Anzahl bestehender Referenzen aufgerufen. Dies spielt im Beisoiel jedoch keine Rolle.

Die Funktion execProgram:

Die Implementierung der Funktion (Beispiel 108) 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 108 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.

18.3 *Verwenden dynamischer Bibliotheken*

18.3.1 Allgemeine Funktionsbibliotheken

Der direkte Weg zur Verwendung von allgemeinen Funktionsbibliotheken (siehe 18.2.1) besteht in der Benutzung der von B++ bereitgestellten Funktionen `LoadLibrary`, `GetProcAddress`, `CallLibFunc` und `FreeLibrary` (vgl. Abschnitt 13.7). 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 hdll=LoadLibrary("testdll.dll");
   if (!hdll) {
      print("Loading 'testdll.dll' failed.\n");
      return;
   }
   // assign address of sayHello function
   intptr proc = GetProcAddress(hdll,"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(hdll),"\n");
}
```

Beispiel 109 Dynamisches Einbinden einer allgemeinen Funktionsbiblithek

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 110 wird dies anhand einer B++-Klasse „System“ illustriert, die die Kapselung der in Abschnitt 18.2.3 vorgestellten Bibliothek übernimmt[109]:

```
/**
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.
*/
```

[109] Die Klasse kann mit beiden Implementierungen der Bibliothek umgehen.

```
class System
{
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 hdll;              // 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()
{
   hdll=LoadLibrary("system.exe");
   // We need not check hdll value since GetProcAddress returns null on
   // invalid handle.
   execproc = GetProcAddress(hdll,"execProgram");
   getenvproc = GetProcAddress(hdll,"getEnv");
   searchproc = GetProcAddress(hdll,"searchPath");
   geterrproc = GetProcAddress(hdll,"getLastError");
   strbuf=newcbuffer(400);
}

/**
Destructor frees releases the library.
*/
System::~System()
{
   FreeLibrary(hdll);
}

/**
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 110 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 18.1 beschrieben. Entsprechend einfacher und übersichtlicher wird nun die Benutzung der Bibliothek im Anwendungsprogramm:

```
// import precompiled System class
#use "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`.

18.3.2 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 vorkompilierten 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. 18.2.2). 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 werden[110].

Hinweis:

B++ kennt keine Standardfunktion, die es – analog zur Funktion `loadmodule` (siehe 13.10) – 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.

[110] Theoretisch 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.

19 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 18.2.2 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.

20 Quellen

[1] Bob-Source http://www.atari-portfolio.co.uk/library/language/bob.zip[111]

[2] Betz, D. M.: Bob: A Tiny Object-Oriented Language. In: Dr.Dobbs Journal, Sep. 1991, S.26ff. https://www.txbobsc.com/misc/bob-article/bob-article.html

[3] Neue Bob-Sourcen von D. M. Betz: XLISP HomePage, https://github.com/dbetz/bob

[4] Ebert, R.-E.: DOS-DLL – Dynamische Bibliotheken für MS-DOS. Berlin 2007 https://www.kiezsoft.de/download/dosdll_manual_de.pdf

[5] van Heesch, Dimitri: Doxygen – Generate documentation from source code. https://www.doxygen.nl/

[6] Wolf, Jürgen: C von A bis Z. Das umfassende Handbuch. Abschnitt 18.4: Variadic Makros - __VA_ARGS__, Rheinwerk Openbook https://openbook.rheinwerk-verlag.de/c_von_a_bis_z/018_c_stdarg_h_004.htm

[7] Demichelis, Alberto: T-Rex a tiny regular expression library. https://tiny-rex.sourceforge.net

[8] ECMAScript regular expression pattern syntax. https://cplusplus.com/reference/regex/ECMAScript

[9] Replacing text macros https://en.cppreference.com/w/cpp/preprocessor/replace

[111] Der Server existiert (Stand Nov. 2025) offenbar nicht mehr. Eine Kopie der Quellen ist unter https://kiezsoft.de/download/bobsrc.zip abrufbar.