

André Willms

Einstieg in Visual C++ 2008

Auf einen Blick

Teil I Ansi C++	23
1 Visual C++ 2008	25
2 Ausgabe & Variablen	43
3 Eingabe & Rechenoperatoren	59
4 Verzweigungen	73
5 Schleifen	93
6 Funktionen & Module	103
7 Arrays & Vektoren	123
8 Zeichen & Strings	133
9 Zeiger & Referenzen	143
10 Klassen	155
11 Vererbung	187
12 Dynamische Speicherverwaltung	213
13 Ausnahmen	219
14 Templates	227
Teil II C++/CLI	233
15 .NET Framework	235
16 C++/CLI-Grundlagen	245
17 Klassen II	261
18 Vererbung II	293
19 Strings & StringBuilder	307
20 Dateiverwaltung	321
21 Delegaten & Ereignisse	347
22 Collections	357
23 Nützliche Klassen	381
24 Der Debugger	397
Teil III Oberflächenprogrammierung	405
25 Windows Forms	407
26 Nützliche Klassen II	451
27 Einfache Steuerelemente	477
28 Praktische Anwendung I	505
29 Komplexere Steuerelemente	525
30 Menüs & Leisten	557
31 GDI+	571
32 Praktische Anwendung II	585
Anhang	611
A Nützliche Tipps	611
B Literaturverzeichnis	621

Inhalt

Einleitung	21
------------------	----

TEIL I Ansi C++

1 Visual C++ 2008	25
1.1 Installation von Visual C++ 2008	25
1.2 Der erste Start	28
1.3 Anlegen eines Projekts	30
1.4 Neue Datei dem Projekt hinzufügen	33
1.5 Eigenes Programm kompilieren	35
1.6 Eigenes Programm starten	37
1.7 Fehler beheben	38
1.8 Projektmappe öffnen	39
1.9 Arbeiten mit Visual C++ 2008	39
1.9.1 Projekt in Projektmappe anlegen	40
1.9.2 Projekt einer Projektmappe hinzufügen	40
1.9.3 Datei einem Projekt hinzufügen	40
1.9.4 Projektdatei kopieren	40
1.9.5 Projektdatei löschen	41
1.9.6 Startprojekt festlegen	41
1.9.7 Der Konfigurations-Manager	41
1.10 Zusammenfassung	42
2 Ausgabe & Variablen	43
2.1 Die Hauptfunktion	43
2.2 Die Ausgabe	44
2.2.1 Namensbereiche	45
2.2.2 Bezugsrahmenoperator	45
2.2.3 cout	46
2.2.4 endl	46
2.2.5 Escape-Sequenzen	47
2.3 Die include-Direktive	48
2.4 using	49
2.5 Variablen	49
2.5.1 Ganzzahlige Variablen	50
2.5.2 Fließkommavariablen	53

2.6	Konstanten	54
2.6.1	define	54
2.7	Kommentare	55
2.7.1	Einzeilige Kommentare	55
2.7.2	Mehrzeilige Kommentare	55
2.7.3	Kommentare verschachteln	56
2.7.4	Kommentare mit Visual C++ 2008	56
2.8	Zusammenfassung	56

3 Eingabe & Rechenoperatoren 59

3.1	Die Eingabe	59
3.2	Der Zuweisungsoperator	60
3.3	Die Grundrechenarten	60
3.3.1	Bindungsstärke	63
3.3.2	Ausdrücke mit unterschiedlichen Typen	63
3.3.3	Explizite Typumwandlung	64
3.4	Zusammengesetzte Zuweisungsoperatoren	66
3.5	Modulo	68
3.6	Inkrement & Dekrement	69
3.6.1	Post-Operatoren	70
3.6.2	Prä-Operatoren	70
3.6.3	Anwendungsgebiete	70
3.7	Zusammenfassung	71
3.8	Übungen	71

4 Verzweigungen 73

4.1	Bedingungen & bool	74
4.2	Vergleichsoperatoren	75
4.3	if	75
4.4	else	77
4.4.1	Vereinfachungen	79
4.5	Logische Operatoren	80
4.5.1	Kurzschlussseigenschaften	82
4.6	Negationsoperator	84
4.6.1	Exklusives Oder	84
4.7	?:-Operator	85
4.8	switch & case	86
4.8.1	default	91
4.9	Zusammenfassung	91
4.10	Übungen	92

5	Schleifen	93
5.1	while	93
5.2	do	96
5.3	for	98
5.4	Wann welche Schleife?	100
5.5	break	100
5.6	continue	101
5.7	Zusammenfassung	102
5.8	Übungen	102
6	Funktionen & Module	103
6.1	Funktionen	103
6.2	Lokale Variablen	105
6.2.1	Lokale Variablen und for	107
6.3	Funktionen mit Parametern	108
6.3.1	Parameter sind lokale Variablen	108
6.3.2	Mehrere Parameter	109
6.4	Rückgabewerte	110
6.5	Funktionsdeklarationen	112
6.6	Module	113
6.6.1	Definition auslagern	114
6.6.2	Deklaration auslagern	114
6.6.3	Kompilationsvorgang	115
6.6.4	Mehrfachdeklarationen vermeiden	117
6.7	Zusammenfassung	119
6.8	Übungen	120
7	Arrays & Vektoren	123
7.1	Arrays	123
7.1.1	Variabler Index	124
7.1.2	Mehrdimensionale Arrays	125
7.1.3	Einschränkungen von Arrays	126
7.2	Vektoren	126
7.2.1	IntelliSense	127
7.2.2	Elemente anhängen	128
7.2.3	Das letzte Element ansprechen	128
7.2.4	Elemente am Ende entfernen	128
7.2.5	Ein beliebiges Element ansprechen	128

7.2.6	Ein beliebiges Element löschen	129
7.2.7	Vektoren kopieren	130
7.2.8	Vektoren als Funktionsparameter	130
7.3	Zusammenfassung	131
7.4	Übungen	131

8 Zeichen & Strings 133

8.1	char	133
8.1.1	cctype-Funktionen	134
8.2	C-Strings	135
8.2.1	Texteingabe	136
8.2.2	Initialisierung von C-Strings	138
8.3	Strings	138
8.3.1	Zuweisung	138
8.3.2	Ein- und Ausgabe	138
8.3.3	Strings verarbeiten	139
8.3.4	Strings vergleichen	140
8.4	Zusammenfassung	140
8.5	Übungen	141

9 Zeiger & Referenzen 143

9.1	Adressoperator	143
9.2	Zeiger	144
9.3	Dereferenzierungsoperator	145
9.4	Zeiger als Funktionsparameter	146
9.5	Zeiger auf Klassenobjekte	148
9.6	Zeiger auf Arrays	149
9.7	Zeigerarithmetik	150
9.8	Referenzen	151
9.9	Zusammenfassung	152
9.10	Übungen	153

10 Klassen 155

10.1	Definition einer Klasse	155
10.1.1	Erstellen einer Klasse mit Visual C++ 2008	155
10.2	Attribute	159
10.3	Zugriffsrechte	160
10.4	Methoden	162

10.4.1	Externe Definition	163
10.4.2	this	164
10.5	Konstruktoren	165
10.5.1	Externe Definition	166
10.5.2	Private Attribute	167
10.5.3	Elementinitialisierungsliste	167
10.6	Konstanzwahrende Methoden	168
10.6.1	Veränderliche Attribute	170
10.7	Überladen von Methoden	170
10.8	Statische Klasselemente	172
10.8.1	Statische Methoden	173
10.8.2	Statische Attribute	175
10.9	Typedef	176
10.10	Die Klassenansicht	178
10.10.1	Der Objektbrowser	180
10.10.2	Der Aufrufbrowser	181
10.11	Namensbereiche	182
10.12	Zusammenfassung	184
10.13	Übungen	185

11 Vererbung **187**

11.1	Das Wesen der Vererbung	187
11.2	Die Syntax der Vererbung	189
11.2.1	Vererbung mit Visual C++	190
11.3	Konstruktoren	191
11.4	Erweitern durch Vererbung	194
11.5	Methoden überschreiben	195
11.6	Geschützte Attribute	196
11.7	Polymorphie	198
11.8	Virtuelle Methoden	199
11.9	UML	201
11.10	Schnittstellen	202
11.10.1	Rein virtuelle Methoden	205
11.10.2	Rein abstrakte Klassen	209
11.11	Downcasts	210
11.12	Zusammenfassung	210
11.13	Übungen	211

12 Dynamische Speicherverwaltung	213
12.1 Erzeugen von Objekten	213
12.2 Erzeugen von Arrays	214
12.3 Destruktoren	214
12.3.1 Virtuelle Destruktoren	216
12.4 Wenn new fehlschlägt	217
12.5 Zusammenfassung	217
12.6 Übungen	217
13 Ausnahmen	219
13.1 Ausnahmen werfen	219
13.2 Ausnahmen fangen	223
13.3 Unterschiedliche Ausnahmen auffangen	224
13.4 Ausnahmen weiter werfen	226
13.5 Zusammenfassung	226
13.6 Übungen	226
14 Templates	227
14.1 Funktionstemplates	227
14.2 Klassentemplates	229
14.3 Zusammenfassung	230
14.4 Übungen	231
 TEIL II C++/CLI	
15 .NET Framework	235
15.1 C++/CLI	235
15.2 .NET	236
15.2.1 Common Language Runtime (CLR)	238
15.2.2 Common Language Specification (CLS)	239
15.2.3 Common Type System (CTS)	240
15.2.4 Klassenbibliothek	242
16 C++/CLI-Grundlagen	245
16.1 CLR-Konsolenanwendung	245
16.1.1 Die Projekt-Dateien	247

16.2	Das Beispielprogramm	247
16.2.1	stdafx	248
16.2.2	Namensbereich System	248
16.2.3	main	248
16.2.4	WriteLine	248
16.3	Trackinghandle	249
16.4	Trackingreferenz	250
16.5	Ausgabe	251
16.5.1	Formatierte Ausgabe	252
16.6	Arrays	253
16.6.1	Arrays initialisieren	254
16.6.2	Mehrdimensionale Arrays	254
16.6.3	for each	254
16.7	Eingabe	255
16.8	Typumwandlung	255
16.9	Ausnahmen	256
16.9.1	finally	259
16.10	Zusammenfassung	259

17 Klassen II 261

17.1	Eine verwaltete Klasse erstellen	261
17.1.1	Zugriffsrecht auf die Klasse	263
17.2	Die Ausgabe	263
17.3	Eigenschaften	265
17.3.1	Externe Definition	268
17.3.2	Unterschiedliche Zugriffsrechte	268
17.3.3	Eigenschaften ohne Attribute	269
17.3.4	Virtuelle und statische Eigenschaften	270
17.4	Indexer	270
17.4.1	Eigenschaften-Indexer	271
17.4.2	Klassen-Indexer	271
17.4.3	Mehrdimensionale Indexer	272
17.5	Ressourcenfreigabe	273
17.5.1	Deterministische Freigabe verwalteter Ressourcen	273
17.5.2	Freigabe nicht verwalteter Ressourcen	275
17.5.3	Deterministische Freigabe nicht verwalteter Ressourcen	277
17.5.4	Zusammenfassung	277
17.6	Wertklassen	278
17.7	Operatoren überladen	279

17.7.1	Operatoren für Werttypen	281
17.7.2	Operatoren für Verweistypen	282
17.7.3	Vergleichsoperatoren	283
17.7.4	Umwandlungsoperatoren	284
17.8	Literale	285
17.9	Aufzählungen	285
17.9.1	Explizite Typangabe	288
17.9.2	Flags	288
17.10	Zusammenfassung	291
17.11	Übungen	291

18 Vererbung II 293

18.1	override vs. new	296
18.2	Abstrakte Methoden und Klassen	297
18.3	Versiegelte Methoden	299
18.4	Versiegelte Klassen	299
18.5	Schnittstellen	301
18.5.1	Schnittstellen und Mehrfachvererbung	302
18.5.2	Identische Methoden	303
18.6	Zusammenfassung	305

19 Strings & StringBuilder 307

19.1	CultureInfo	307
19.2	String	308
19.2.1	Stringinhalte ansprechen	309
19.2.2	Stringinhalte verändern	310
19.2.3	Stringinhalte hinzufügen	310
19.2.4	Stringinhalte löschen	311
19.2.5	Strings vergleichen	312
19.2.6	Suchen in Strings	313
19.2.7	Strings formatieren	315
19.3	StringBuilder	315
19.3.1	Stringinhalte ansprechen	315
19.3.2	Stringinhalte hinzufügen	316
19.3.3	Stringinhalte verändern	316
19.3.4	Stringinhalte löschen	317
19.3.5	StringBuilder-Objekte vergleichen	317
19.4	Char	317
19.5	Zusammenfassung	318
19.6	Übungen	319

20 Dateiverwaltung 321

20.1	DateTime	322
20.1.1	Öffentliche Eigenschaften	322
20.1.2	Öffentliche Methoden	323
20.2	Laufwerke	325
20.3	Verzeichnisse	327
20.3.1	Directory	327
20.3.2	FileSystemInfo	330
20.3.3	DirectoryInfo	331
20.4	Dateien	331
20.4.1	File	331
20.4.2	FileInfo	333
20.5	Dateiströme	334
20.5.1	FileStream	335
20.6	Binärströme	337
20.6.1	BinaryWriter	337
20.6.2	BinaryReader	338
20.7	Zeichenströme	339
20.7.1	StreamWriter	339
20.7.2	StreamReader	340
20.8	Serialisierung	341
20.8.1	SerializableAttribute	342
20.9	Praktische Anwendung	343
20.9.1	Einlesen einer Textdatei	343
20.9.2	Beschreiben einer Log-Datei	344
20.9.3	Datei byteweise kopieren	345
20.10	Zusammenfassung	345
20.11	Übungen	346

21 Delegaten & Ereignisse 347

21.1	Delegaten	347
21.1.1	Multicast-Delegaten	350
21.2	Ereignisse	351
21.2.1	Die Klasse Becher mit Ereignissen	353
21.3	Zusammenfassung	355
21.4	Übungen	355

22 Collections 357

22.1	IComparer	357
22.2	IComparable	358
22.3	Collection-Schnittstellen	359
22.4	IEnumerable	359
22.5	ICollection	360
22.5.1	Queue	361
22.5.2	Stack	363
22.6	IList	364
22.6.1	ArrayList	365
22.7	IDictionary	367
22.7.1	Hashtable	369
22.7.2	SortedList	370
22.8	Generische Collections	371
22.8.1	Die generischen Schnittstellen	373
22.9	Anwendungsbeispiele	374
22.9.1	Personen verwalten mit List	375
22.9.2	Eine Collection mit int-Werten absteigend sortieren	377
22.10	Zusammenfassung	378
22.11	Übungen	379

23 Nützliche Klassen 381

23.1	Random – Zufallszahlen	381
23.2	Math – mathematische Funktionen	382
23.2.1	Methoden	383
23.2.2	Konstanten	386
23.3	Console – Konsole	386
23.3.1	Eigenschaften	386
23.3.2	Methoden	388
23.4	Environment – die Umgebung	389
23.4.1	Eigenschaften	390
23.4.2	Methoden	390
23.5	GC – Garbage Collector	392
23.6	Timer – Taktgeber	394
23.6.1	Ein Beispiel	395
23.7	Zusammenfassung	396

24 Der Debugger 397

24.1	Haltepunkte	399
24.2	Schrittweise Abarbeitung	400
24.3	Komplexere Haltepunkte	402
24.3.1	Halten bei Bedingung	402
24.3.2	Halten bei Trefferzahl	402
24.4	Variablen überwachen	403
24.5	Zusammenfassung	404
24.6	Übungen	404

TEIL III Oberflächenprogrammierung**25 Windows Forms 407**

25.1	Das Hauptprogramm	409
25.2	Die Form-Datei	410
25.3	Das Eigenschaftfenster des Designers	412
25.4	Component	414
25.5	Control – Basis aller Steuerelemente	415
25.5.1	Öffentliche Eigenschaften	415
25.5.2	Öffentliche Methoden	425
25.5.3	Öffentliche Ereignisse	428
25.6	ScrollableControl – scrollbare Container	435
25.6.1	Öffentliche Eigenschaften	436
25.6.2	Öffentliche Methoden	438
25.6.3	Öffentliche Ereignisse	438
25.7	Form – die Formulkasse	439
25.7.1	Öffentliche Eigenschaften	439
25.7.2	Öffentliche Methoden	444
25.7.3	Öffentliche Ereignisse	445
25.8	Ereignisse im Designer	447
25.8.1	Handler hinzufügen	447
25.8.2	Handler entfernen	449
25.9	Zusammenfassung	449

26 Nützliche Klassen II 451

26.1	Assembly-Verweise hinzufügen	451
26.2	Size – Größenangabe	454
26.2.1	Eigenschaften	454

26.2.2	Methoden	454
26.3	Point – Positionsangabe	455
26.3.1	Eigenschaften	455
26.3.2	Methoden	455
26.4	Rectangle – rechteckiger Bereich	456
26.4.1	Eigenschaften	456
26.4.2	Methoden	456
26.5	Color – Farbangaben	459
26.5.1	Eigenschaften	459
26.5.2	Methoden	461
26.5.3	Farben im Designer	462
26.6	Font – Schriftart	464
26.6.1	Eigenschaften	465
26.6.2	Fonts im Designer	465
26.7	MessageBox – Nachrichtenfenster	466
26.8	Image – Grundlage der Bilder	468
26.8.1	Eigenschaften	468
26.8.2	Methoden	468
26.9	Bitmap – Klasse für konkrete Bilder	469
26.9.1	Methoden	470
26.10	Icon – kleine Bilder	470
26.10.1	Eigenschaften	471
26.10.2	Methoden	471
26.10.3	SystemIcons – Klasse mit den System-Icons	471
26.11	ImageList – Bilderliste	472
26.11.1	Eigenschaften	472
26.11.2	Methoden	472
26.12	Cursor – Mauszeiger	473
26.12.1	Eigenschaften	473
26.12.2	Methoden	474
26.12.3	Cursors – ein Mauszeiger für jede Situation	474
26.13	Padding – Abstände und Ränder	475
26.13.1	Eigenschaften	475
26.13.2	Methoden	475
26.14	Zusammenfassung	476

27 Einfache Steuerelemente 477

27.1	Label – Beschriftungen	477
27.1.1	Eigenschaften	478
27.1.2	LinkLabel – anklickbare Beschriftungen	479

27.2	GroupBox – Gruppierungen	479
27.3	ButtonBase – Basis der Buttons	480
27.3.1	Eigenschaften	480
27.4	Button – Schaltfläche	481
27.5	CheckBox – Elemente zum Abhaken	482
27.5.1	Eigenschaften	483
27.5.2	Ereignisse	484
27.6	RadioButton – Optionen zur Auswahl	484
27.6.1	Eigenschaften	485
27.6.2	Methoden	485
27.6.3	Ereignisse	485
27.7	PictureBox – Bilderrahmen	486
27.7.1	Eigenschaften	486
27.7.2	Methoden	487
27.7.3	Ereignisse	488
27.8	TextBoxBase – Basis der Texteingabefelder	489
27.8.1	Eigenschaften	489
27.8.2	Methoden	491
27.8.3	Ereignisse	492
27.9	TextBox – ein einfaches Texteingabefeld	493
27.9.1	Eigenschaften	493
27.10	MaskedTextBox – Eingabe nach Vorschrift	495
27.11	RichTextBox – kleine Textverarbeitung	496
27.12	ListControl – Basis aller Listenelemente	497
27.12.1	Eigenschaften	498
27.13	ListBox – einfache Auflistung	498
27.13.1	Eigenschaften	499
27.13.2	Methoden	500
27.13.3	Ereignisse	502
27.13.4	ComboBox	502
27.14	ProgressBar – Fortschrittsbalken	502
27.14.1	Eigenschaften	503
27.14.2	Methoden	504
27.15	Zusammenfassung	504

28 Praktische Anwendung I 505

28.1	Bei Buttonklick den eingegebenen Text anzeigen	505
28.1.1	Entwurf im Designer	505
28.1.2	Implementierung der Ereignis-Handler	506
28.2	Auf Mausbewegungen reagieren	506

28.2.1	Entwurf im Designer	507
28.2.2	Implementierung der Ereignis-Handler	507
28.3	Die Listboxauswahl mit Zusatzinfos versehen	508
28.3.1	Entwurf im Designer	509
28.3.2	Arbeiten im Konstruktor	509
28.3.3	Implementierung des Ereignis-Handlers	511
28.4	Ein primitiver Texteditor	512
28.4.1	FileDialog	513
28.4.2	OpenFileDialog	516
28.4.3	SaveFileDialog	516
28.4.4	Problembeschreibung	517
28.4.5	Entwurf im Designer	518
28.4.6	Implementierung der Ereignis-Handler	518
28.5	Ein einfacher Bildbetrachter	521
28.5.1	Entwurf im Designer	522
28.5.2	Implementierung der Ereignis-Handler	523

29 Komplexere Steuerelemente 525

29.1	Panel – Basis komplexerer Gruppierungen	525
29.1.1	Eigenschaften	525
29.2	FlowLayoutPanel – Gruppierung wie Fließtext	526
29.2.1	Eigenschaften	526
29.3	TableLayoutPanel – Gruppierung zu Tabellenform	527
29.3.1	Eigenschaften	527
29.4	SplitContainer – größenveränderbare Aufteilung	528
29.4.1	Eigenschaften	529
29.4.2	Ereignisse	531
29.5	TabControl – Gruppierung über Registerkarten	532
29.5.1	Eigenschaften	532
29.5.2	Ereignisse	533
29.5.3	TabPage – Registerkarte	534
29.6	ListView – zweidimensionale Listen	535
29.6.1	Eigenschaften	536
29.6.2	Methoden	541
29.6.3	Ereignisse	542
29.6.4	ColumnHeader – Überschriften der Liste	544
29.6.5	ListViewGroup – Gruppen der Liste	545
29.6.6	ListViewItem – Elemente der Liste	545
29.6.7	ListViewSubItem – Unterelemente der Liste	547
29.7	TreeView – Baumdarstellung	548

29.7.1	Methoden	551
29.7.2	Ereignisse	551
29.7.3	TreeNode – Knoten des Baumes	553
29.8	Zusammenfassung	556

30 Menüs & Leisten 557

30.1	ToolStrip – Symbolleiste	557
30.1.1	Eigenschaften	558
30.2	MenuStrip – Menüleiste	559
30.3	StatusStrip – Statusleiste	560
30.4	ContextMenuStrip – Kontextmenü	560
30.5	Die ToolStrip-Elemente	561
30.5.1	ToolStripItem – Basis der ToolStrip-Elemente	561
30.5.2	ToolStripButton – einfacher Button	563
30.5.3	ToolStripComboBox – Combobox für ToolStrips	563
30.5.4	ToolStripDropDownButton – aufklappbare Schaltfläche	563
30.5.5	ToolStripLabel, ToolStripStatusLabel – Leistenbeschriftung	564
30.5.6	ToolStripMenuItem – Menüpunkt	565
30.5.7	ToolStripProgressBar – Fortschrittsbalken in der Leiste	566
30.5.8	ToolStripSeparator – Spalter unter den Elementen	567
30.5.9	ToolStripSplitButton – Kombischaltfläche	567
30.5.10	ToolStripTextBox – Textbox in der Leiste	567
30.6	ToolStripContainer – Spielwiese für Leisten	568
30.7	Zusammenfassung	569

31 GDI+ 571

31.1	Brush – Pinsel	572
31.1.1	SolidBrush – eine solide Füllung	572
31.1.2	TextureBrush – Stempel	573
31.1.3	LinearGradientBrush – Farbverlauf	574
31.1.4	HatchBrush – Pinsel mit Muster	575
31.1.5	Brushes – Pinsel für alle Fälle	576
31.2	Pen – Stift	576
31.2.1	Pens – für jede Farbe einen Stift	576
31.3	Graphics – Zeichenbrett	577
31.3.1	Methoden	577
31.4	Zeichnen über Paint	580
31.4.1	Partielles Neuzeichnen	581

31.4.2	Double Buffering	582
31.5	Zusammenfassung	583
31.6	Übungen	583
32 Praktische Anwendung II		585
32.1	Ein Scherzdialog	585
32.1.1	Entwurf im Designer	585
32.1.2	Die Implementierung der Handler	586
32.2	Ein Telefonbuch	588
32.2.1	Entwurf im Designer	589
32.2.2	Vorbereitende Programmierung	591
32.2.3	Behandlung der Ereignisse	598
32.3	Ein einfacher Dateexplorer	603
32.3.1	Die Klasse FolderTreeView	604
32.3.2	Die Klasse FolderListView	606
32.3.3	Zusammensetzen im Formular	609
Anhang		611
A	Nützliche Tipps	611
A.1	Festlegen des Arbeitsverzeichnisses	611
A.2	Eingebettete Ressourcen	613
A.3	Weitere Dialoge	616
A.3.1	FolderBrowserDialog – Suche nach Verzeichnissen	616
A.3.2	ColorDialog – bringt Farbe ins Leben	618
A.3.3	FontDialog – Auswahl der Schriftarten	619
B	Literaturverzeichnis	621
Index		623

Es ist bemerkenswert, dass nur vielleicht 10 % aller Programmierer Programme ohne Verwendung von Flussdiagrammen erfolgreich schreiben können. Unglücklicherweise glauben aber 90 %, dass sie der Gruppe dieser 10 % angehören.
– Rodney Zaks

11 Vererbung

Abgesehen von der Tatsache, dass Vererbung ein wesentliches Abstraktionsprinzip der Objektorientierten Programmierung darstellt, dient sie hauptsächlich einem einzigen Zweck: der Erweiterung bestehender Funktionalitäten.

Angenommen, die Klasse `Becher` aus Kapitel 10 soll um die Fähigkeit erweitert werden, einen Aufdruck zu besitzen (wie z. B. »Weltbester C++-Programmierer«). Sie könnten diese Eigenschaft problemlos in die Klasse `Becher` einbauen.

Dann tritt der Nächste mit einem Wunsch an Sie heran: Die Klasse `Becher` soll verschiedene »Stoffe« in unterschiedlicher Menge aufnehmen können (z. B. Kaffee, Milch und Zucker). Auch das bauen Sie in die Klasse ein. Nun hat der zukünftige Mischer aber auch noch den vorhin implementierten Aufdruck dabei, obwohl er ihn vielleicht nicht benötigt.

Sie können dieses Beispiel weiterspinnen, bis Sie eine riesige, monolithische Klasse erhalten, die zwar fast alles kann, wovon der Anwender aber immer nur maximal 10 % benötigt. Dieses Problem wird mit Vererbung umgangen.

Ein anderes Problem ist der Wunsch, eine Klasse zu erweitern, deren Quellcode Ihnen nicht zur Verfügung steht. Auch das ist mit Vererbung lösbar.

11.1 Das Wesen der Vererbung

Die in fast allen Fällen verwendete Vererbung ist die öffentliche Vererbung, die ein »ist ein(e)«-Verhältnis zwischen Klassen zum Ausdruck bringt.¹

¹ C++ kennt noch zwei andere Arten der Vererbung. Da vom .NET Framework aber nur die öffentliche Vererbung unterstützt wird, wollen wir uns hier auf sie beschränken. Weitere Informationen finden Sie in [Willms05].

Als einfaches Studienbeispiel soll die `Becher`-Klasse, wie im vorigen Abschnitt angesprochen, mit der Möglichkeit erweitert werden, einen Aufdruck zu bekommen. Wir können sagen: Ein Becher mit Aufdruck ist ein Becher, der zusätzlich noch einen Aufdruck besitzt. Der Kern dieser Aussage lautet: Ein Becher mit Aufdruck ist ein Becher. Das ist auch logisch, denn ein Becher mit Aufdruck kann all das, was auch ein normaler Becher kann, er hat eben nur noch einen Aufdruck.

Und genau das macht die Vererbung. Die Klasse, an die vererbt wird (auch »Subklasse« oder »abgeleitete Klasse« genannt), erbt alle Eigenschaften der vererbenden Klasse (auch Basisklasse oder Superklasse genannt) und kann diese nach Bedarf erweitern.

Bei der Vererbung stehen die öffentlichen Elemente der Basisklasse als öffentliche Elemente der Subklasse zur Verfügung. Die geschützten Elemente der Basisklasse sind als geschützte Elemente der Subklasse vorhanden.

Nur: Die privaten Elemente der Basisklasse werden zwar auch an die Subklasse vererbt, sie sind aber von der Subklasse aus nicht ansprechbar. Abbildung 11.1 stellt den Zusammenhang grafisch dar.

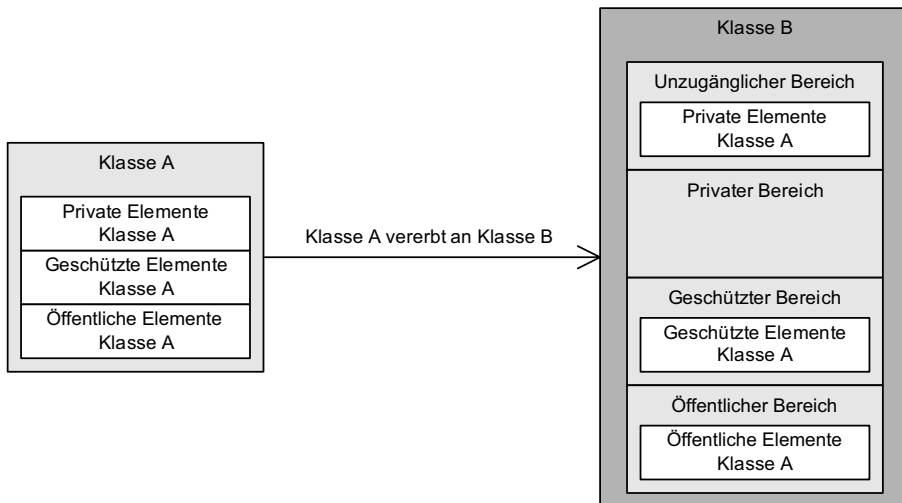


Abbildung 11.1 Das Prinzip der öffentlichen Vererbung

Es ist vielleicht etwas gewöhnungsbedürftig, dass im obigen Fall Klasse B Elemente von Klasse A geerbt hat, auf die sie nicht zugreifen kann. Aber andererseits ist dies eine logische Folge aus der Forderung, dass auf private Elemente einer Klasse nur die Klasse selbst zugreifen kann.

11.2 Die Syntax der Vererbung

Syntaktisch ist die Vererbung schnell umgesetzt. Die neue Klasse wird zunächst, wie in Abschnitt 10.1, »Definition einer Klasse«, beschrieben, mit leerem Anweisungsblock angelegt.

```
class BecherMitAufdruck
{
}
```

Hinter dem Klassennamen wird, durch einen Doppelpunkt getrennt, der Name der Basisklasse (in unserem Fall `Becher`) angegeben:

```
class BecherMitAufdruck : public Becher
{
}
```

Listing 11.1 Die Syntax der Vererbung

Das Schlüsselwort `public` vor dem Basisklassennamen leitet die gewünschte öffentliche Vererbung ein.

Sie erinnern sich, dass die Klasse `Becher` im Namensbereich `Getraenke` steht. Grundsätzlich ist es sinnvoll, die Klasse `BecherMitAufdruck` ebenfalls in diesem Namensbereich unterzubringen:

```
namespace Getraenke
{
    class BecherMitAufdruck : public Becher {
    }
}
```

Listing 11.2 Die Klasse »BecherMitAufdruck« im Namensbereich `Getraenke`

Sollte es aus logischer Sicht keinen Sinn ergeben, die Klasse `BecherMitAufdruck` in den Namensbereich `Getraenke` zu setzen, dann muss auf die Klasse `Becher` mit expliziter Angabe ihres Namensbereichs verwiesen werden. Es ist dabei unerheblich, ob `BecherMitAufdruck` in einem anderen oder in gar keinem Namensbereich steht:

```
class BecherMitAufdruck : public Getraenke::Becher
{
}
```

Listing 11.3 Angabe des Namensbereichs der Basisklasse

Im weiteren Verlauf soll der Ansatz aus Listing 11.2 weiterverfolgt werden.

11.2.1 Vererbung mit Visual C++

Die Entwicklungsumgebung unterstützt auch die Vererbung. Wenn eine Klasse, wie in Abschnitt 10.1.1 beschrieben, mit dem Klassenassistenten der Entwicklungsumgebung erstellt wird, erscheint irgendwann das in Abbildung 11.2 gezeigte Fenster.

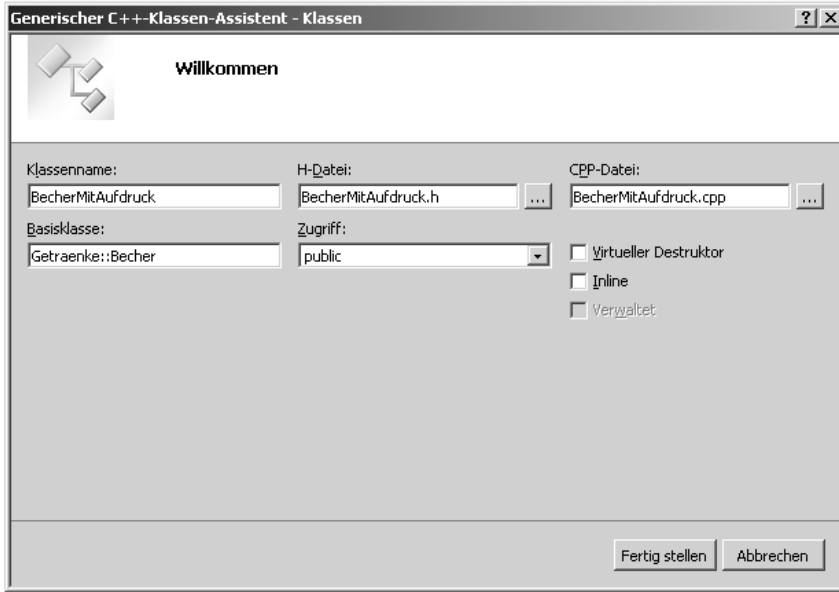


Abbildung 11.2 Die Vererbung im Klassenassistenten

Unter dem Punkt **Basisklasse** wird der Name der Basisklasse eingetragen. Der Assistent erstellt die Datei immer im globalen Namensbereich, deswegen muss der Namensbereich der Basisklasse auf jeden Fall angegeben werden.

Bei Zugriff ist `public` bereits eingetragen und muss nicht mehr verändert werden. Nach abschließendem Klick auf **Fertig stellen** erzeugt der Assistent die Header- und Quellcode-Datei für die Klasse. Die Header-Datei sieht so aus:

```
#pragma once
#include "becher.h"

class BecherMitAufdruck :
    public Getraenke::Becher
{
public:
    BecherMitAufdruck(void);
public:
```

```
    ~BecherMitAufdruck(void);
};
```

Listing 11.4 Vom Assistenten erzeugte »BecherMitAufdruck.h«

Der Assistent hat automatisch die Datei »becher.h« eingebunden, denn zum Ableiten muss dem Compiler die Definition der Basisklasse bekannt sein.

Auch besitzt die Klasse wieder einen Destruktor, dessen Bedeutung in Abschnitt 12.3, »Destruktoren«, besprochen wird und erst einmal entfernt werden kann. Auch das `void` in der Parameterliste des Konstruktors ist unnötig und kann gelöscht werden. Und der gewünschte Namensbereich muss noch manuell hinzugefügt werden. Nach entsprechender kosmetischer Aufbereitung sieht die Datei so aus:

```
#pragma once
#include "becher.h"

namespace Getraenke {
    class BecherMitAufdruck : public Becher {
    public:
        BecherMitAufdruck();
    };
}
```

Listing 11.5 Die aufbereitete »BecherMitAufdruck.h«

Die Datei »BecherMitAufdruck.cpp« muss dann noch entsprechend angepasst werden.

11.3 Konstruktoren

Völlig nebensächlich, ob Sie mit dem Klassenassistenten gearbeitet oder die Klassen »zu Fuß« erstellt haben, werden Sie die bittere Enttäuschung eines Fehlers während der Kompilation erfahren.

Und zwar beschwert sich der Compiler über einen fehlenden Standardkonstruktor in der Klasse `Becher`.

Ein guter Zeitpunkt also, ein wenig die Konstruktion von Objekten abgeleiteter Klassen zu beleuchten. Wir können der Lösung bereits mit einigen einfachen Fragen erstaunlich nahekommen.

Wie wird in einer Klasse ein Objekt erzeugt? Über den Konstruktor. Wie konstruiert ein Konstruktor ein Objekt? Indem er – entweder im Anweisungsblock oder in der Elementinitialisierungsliste – die Attribute des Objekts initialisiert – meist mithilfe übergebener Argumente. Hier tritt schon der erste Fauxpas zutage: Unser Konstruktor besitzt noch keine sinnvollen Parameter.

Diese kleine formale Unschärfe einmal kurz beiseite geschoben, tritt ein neues Problem auf: Wie soll der Konstruktor von `BecherMitAufdruck` die Objektattribute initialisieren? Einfach durch Zuweisung oder über die Elementinitialisierungsliste? Eher nicht, denn wie Abbildung 11.1 schön darstellt, können die Elemente der Subklasse – und dazu gehört der Konstruktor von `BecherMitAufdruck` – nicht auf die privaten Elemente der Basisklasse zugreifen.

Es wird also etwas benötigt, das auf der einen Seite Zugriff auf die privaten Elemente der Basisklasse hat – es kommt also nur ein Element der Basisklasse infrage – und in der Lage ist, die Basisklassenattribute zu initialisieren.

Einmal kurz über den Quellcode von `Becher` meditiert findet sich der geeignete Kandidat: Der Konstruktor von `Becher`. Er ist öffentlich und daher von der Basisklasse aus ansprechbar und er weiß, wie die Attribute zu initialisieren sind. Wir brauchen also nur vom Subklassenkonstruktor aus den Basisklassenkonstruktor aufzurufen.

Dummerweise benötigt der Basisklassenkonstruktor zur Initialisierung der Attribute drei Argumente. Damit ihm diese Argumente übergeben werden können, muss der Subklassenkonstruktor diese vom Anwender einfordern.

Die grobe Richtung steht also fest: Der Basisklassenkonstruktor benötigt drei Argumente, die der Subklassenkonstruktor über eigene Parameter beschaffen muss. Betrachten wir dazu kurz die Datei »`BecherMitAufdruck.cpp`«, die den aktualisierten Konstruktor enthält:

```
#include "BecherMitAufdruck.h"

using namespace std;

namespace Getraenke {

    BecherMitAufdruck::BecherMitAufdruck(string i,
                                           int fa,
                                           float fu) {

    }

}
```

Listing 11.6 Die Datei »`BecherMitAufdruck.cpp`«

Die Konstruktordeklaration in der Klassendefinition muss ebenfalls mit den Parametern versehen werden.

Kompilieren lässt sich das Programm aber immer noch nicht, denn der Basisklassenkonstruktor wurde noch nicht aufgerufen. Dafür gilt:

Der Basisklassenkonstruktor wird in der Elementinitialisierungsliste des Subklassenkonstruktors aufgerufen.

Der endgültige Konstruktor sieht so aus:

```
BecherMitAufdruck::BecherMitAufdruck(string i,
                                     int fa,
                                     float fu)
: Becher(i, fa, fu) {
}

```

Listing 11.7 Aufruf des Basisklassenkonstruktors

Und wenn Sie sich fragen, wie der ursprüngliche Fehler entstanden ist und was er bedeutet:

In C++ *muss* der Subklassenkonstruktor einen Basisklassenkonstruktor aufrufen.

Welcher Konstruktor aufgerufen wird – es können durch Überladung mehrere existieren –, spielt keine Rolle. Diese Regel ist so wichtig, dass selbst dann ein Basisklassenkonstruktor aufgerufen wird, wenn der Programmierer keinen expliziten Aufruf angegeben hat:

Besitzt der Subklassenkonstruktor keinen expliziten Aufruf eines Basisklassenkonstruktors, dann wird der Standardkonstruktor der Basisklasse aufgerufen.

Als Standardkonstruktor wird der Konstruktor ohne Parameter bezeichnet. Zu Beginn wurde also automatisch dieser Standardkonstruktor von `Becher` aufgerufen. Da wir einen solchen aber nicht programmiert haben, kam die Fehlermeldung »kein geeigneter Standardkonstruktor verfügbar«.

Nun endlich kann ein Objekt der neuen Klasse erstellt werden:

```
BecherMitAufdruck b("Milch", 300, 90);
```

Die Klasse `BecherMitAufdruck` hat durch die Vererbung alle Fähigkeiten der Basisklasse `Becher` geerbt. Ohne weiter entwicklerisch aktiv zu werden, kann ein

BecherMitAufdruck-Objekt daher über die geerbte Methode `ausgabe` ausgegeben werden:

```
b.ausgabe();
```

Alle anderen in `Becher` implementierten Methoden sind ebenfalls verfügbar.

11.4 Erweitern durch Vererbung

Der bisher betriebene Aufwand für die Klasse `BecherMitAufdruck` hat uns zu dem Punkt gebracht, dass sie exakt die gleichen Fähigkeiten besitzt wie ihre Basisklasse `Becher`. Das klingt nicht nach einem großartigen Gewinn, aber: Wir haben über die Vererbung die Funktionalitäten von `Becher` übernommen, ohne die Klasse `Becher` verändern zu müssen. Diese Technik funktioniert daher auch bei Klassen, auf deren Quellcode wir keinen Zugriff haben.²

Die Klasse `BecherMitAufdruck` kann jetzt allerdings nach Belieben erweitert werden und soll im weiteren Verlauf ihrem Namen alle Ehre machen.

Der Becheraufdruck soll in der Klasse als Text gespeichert werden. Wir benötigen dazu in `BecherMitAufdruck` ein entsprechendes Attribut. Damit dieses Attribut initialisiert werden kann, muss der Konstruktor um einen Parameter erweitert werden:

```
class BecherMitAufdruck : public Becher {
    std::string aufdruck;
public:
    BecherMitAufdruck(std::string i, int fa, float fu,
                     std::string auf);
};
```

Listing 11.8 Die Klasse »BecherMitAufdruck« mit neuem Attribut

Das neue Attribut steht direkt am Klassenanfang und besitzt deshalb implizit privates Zugriffsrecht. Im obigen Beispiel wurde der zusätzliche Konstruktorparameter an die Parameterliste angehängt. Die Parameterreihenfolge ist allerdings beliebig und kann nach Bedarf verändert werden. Die Parameterlisten von Deklaration und Definition müssen aber übereinstimmen.

Die Definition des Konstruktors sieht so aus:

```
BecherMitAufdruck::BecherMitAufdruck(string i,
                                       int fa,
```

² In diese Rubrik fallen später die Klassen der .NET-Bibliothek.

```

        float fu,
        string auf)
: Becher(i, fa, fu), aufdruck(auf) {
}

```

Listing 11.9 Der neue Konstruktor von »BecherMitAufdruck«

Weitere Funktionalität kann auf diese Weise nach Belieben hinzugefügt werden. Diese Zugriffsmethode für den Aufdruck wäre beispielsweise nicht schlecht:

```

std::string getAufdruck() const {
    return(aufdruck);
}

```

Listing 11.10 Die Zugriffsmethode »getAufdruck«

Oben ist nur die Definition der Methode zu sehen. Handelt es sich bei dieser Definition um eine externe oder um eine Inline-Definition?

Wäre es eine externe Definition, müsste angegeben werden, zu welcher Klasse die Methode gehört (`BecherMitAufdruck::getAufdruck`), insofern muss es sich um eine Methodendefinition innerhalb der Klassendefinition handeln.

Falls Sie nicht mehr genau wissen, welche Auswirkungen das `const` hinter dem Methodenkopf hat, dann schlagen Sie schnell in Abschnitt 10.6, »Konstanzwahrende Methoden«, nach.

11.5 Methoden überschreiben

Sie wissen von den vorigen Abschnitten, dass in der Klasse `BecherMitAufdruck` durch die Vererbungsbeziehung zu `Becher` deren Methode `ausgabe` geerbt wurde und aufgerufen werden kann.

Nun wäre es für die Klasse `BecherMitAufdruck` aber schön, wenn bei der Ausgabe auch der Becheraufdruck ausgegeben würde. Aber wie geht das?

Die bestehende `ausgabe`-Methode entsprechend abzuändern, ist aus einem einfachen Grund nicht möglich: Die Methode gehört zu `Becher` und dort existiert das auszugebende Attribut `aufdruck` noch nicht.

Die Konsequenz daraus führt zu einer neuen Methode für `BecherMitAufdruck`. Eine Methode mit demselben Namen ist auf den ersten Blick nicht möglich, denn die Parameterliste würde sich nicht von der geerbten Methode unterscheiden und ein Überladen unmöglich machen.

Eine Methode mit anderem Namen zu implementieren ist auch keine saubere Lösung, denn es würde weiterhin die geerbte `ausgabe`-Methode zur Verfügung stehen, die den Ausdruck nicht mit ausgibt.

Die Lösung ist erschreckend simpel: Wenn wir in `BecherMitAusdruck` eine Methode `ausgabe` implementieren, dann ist das kein Überladen. Zur Erinnerung³: Überladen ist ein Name nur dann, wenn er mehrfach im selben Bezugsrahmen definiert ist. Die geerbte Methode gehört zum Bezugsrahmen `Becher`, die neue Methode wird zum Bezugsrahmen `BecherMitAusdruck` gehören. Unterschiedliche Bezugsrahmen, daher kein Überladen, vielmehr:

Wird in einer abgeleiteten Klasse eine Methode mit gleichem Namen und Parameterliste einer geerbten Methode definiert, wird die geerbte Methode mit der neuen Methode überschrieben.

Praktisch heißt das: Wir können in `BecherMitAusdruck` eine neue `ausgabe`-Methode definieren und haben damit einfach die alte überschrieben:

```
void BecherMitAusdruck::ausgabe() const {
    cout << "Becher mit " << inhalt <<
        " und Ausdruck \"" << ausdruck <<
        "\"\" << endl;
}
```

Listing 11.11 Erster Versuch einer `ausgabe`-Methode

Fertig? Mitnichten! Die Methode wird bei der Kompilation einen Fehler melden. Der Grund müsste Ihnen bereits bekannt sein, wird aber im folgenden Abschnitt noch genauer besprochen.

11.6 Geschützte Attribute

Das Problem bei der Methode aus Listing 11.11 liegt im Zugriff auf das Attribut `inhalt`. Es handelt sich hierbei um ein `privates` Attribut von `Becher`, weshalb eine Methode von `BecherMitAusdruck` keinen Zugriff darauf hat.

Es gibt nun zwei Möglichkeiten. Die erste besteht darin, das Attribut mit geschütztem Zugriffsrecht zu versehen. Wie in Abschnitt 10.3, »Zugriffsrechte«, bereits beschrieben wurde, erlaubt das geschützte Zugriffsrecht auch abgeleiteten Klassen den Zugriff. Eigentlich genau das, was wir brauchen:

³ Abschnitt 10.7, »Überladen von Methoden«

```

class Becher {
public:
    typedef float AbsWertTyp;
    Becher(std::string i, int fa, float fu);
    void ausgabe() const;
    bool reichtKapazitaet(int ml) const;
    bool reichtKapazitaet(const Becher* b) const;
    static AbsWertTyp berechneAbsolutwert(float gw, float ps);

protected:
    std::string inhalt;

private:
    int fassungsvermoegen;
    float fuellhoehe;
    static int maxmenge;
};

```

Listing 11.12 Die Klasse »Becher« mit geschütztem inhalt-Attribut

Nun lässt sich das Programm problemlos kompilieren. Aber: Eine vollständige Datenkapselung ist nicht mehr gewährleistet, weil auf `inhalt` nun jede abgeleitete Klasse zugreifen kann.

Dieses Zugriffsrecht bezieht sich zwar nur auf das geerbte Attribut. Eine Methode der Klasse `BecherMitAufdruck` ist nicht in der Lage, auf das `inhalt`-Attribut eines `Becher`-Objekts zuzugreifen.

Trotzdem kann eine abgeleitete Klasse eventuell von der Basisklasse auferlegte Beschränkungen umgehen. Aus diesem Grund sieht ein sauberer Ansatz weiterhin `private` Attribute vor und ermöglicht den Zugriff über Methoden, die nach Bedarf dann mit geschütztem Zugriffsrecht versehen werden können. Dazu wird das Attribut `inhalt` wieder privat und die Klasse um eine öffentliche Methode `getInhalt` erweitert:

```

class Becher {
public:
    typedef float AbsWertTyp;
    Becher(std::string i, int fa, float fu);
    void ausgabe() const;
    bool reichtKapazitaet(int ml) const;
    bool reichtKapazitaet(const Becher* b) const;
    static AbsWertTyp berechneAbsolutwert(float gw, float ps);
    std::string getInhalt() const {
        return(inhalt);

```

```

    }

private:
    std::string inhalt;
    int fassungsvermoegen;
    float fuellhoehe;
    static int maxmenge;
};

```

Listing 11.13 Die Klasse »Becher« mit `getInhalt`

Die `ausgabe`-Methode von `BecherMitAufdruck` muss nun nur noch entsprechend angepasst werden:

```

void BecherMitAufdruck::ausgabe() const {
    cout << "Becher mit " << getInhalt() <<
        " und Aufdruck \"" << aufdruck <<
        "\"\" << endl;
}

```

Listing 11.14 Die endgültige Fassung von »`BecherMitAufdruck::ausgabe`«

11.7 Polymorphie

Wir wissen mittlerweile, dass die öffentliche Vererbung eine »ist ein(e)«-Beziehung darstellt. In den letzten Abschnitten haben wir deshalb über die Vererbung zum Ausdruck gebracht, dass ein `Becher` mit `Aufdruck` ein `Becher` ist. Im Umkehrschluss muss es deshalb möglich sein, einen `Becher` mit `Aufdruck` als gewöhnlichen `Becher` zu behandeln, denn er ist ja ein `Becher`. Dieser Sachverhalt wird in der Objektorientierten Programmierung **Polymorphie** genannt:

Überall dort, wo ein Objekt der Basisklasse erwartet wird, kann auch ein Objekt einer abgeleiteten Klasse verwendet werden.

Nehmen wir im einfachsten Fall einen Zeiger vom Typ `Becher`:

```
Becher *bptr;
```

Weil ein `Becher` mit `Aufdruck` ein `Becher` ist, kann die Adresse eines `BecherMitAufdruck`-Objekts einem `Becher`-Zeiger zugewiesen werden:

```
BecherMitAufdruck b("Milch", 300, 90, "Meine Privattasse");
bptr = &b;
```

Über diesen Zeiger können dann die `Becher`-Methoden des Objekts aufgerufen werden:

```
cout << bptr->getFuellmenge() << endl;
```

Polymorphie funktioniert nur mit Zeigern und Referenzen. Über den `Becher`-Zeiger können allerdings keine speziellen Methoden der Klasse `BecherMitAufdruck` aufgerufen werden, denn der Zeiger »weiß« schließlich nicht, dass das `Becher`-Objekt, auf das er zeigt, in Wirklichkeit ein `BecherMitAufdruck`-Objekt ist.

Praktischer Einsatz der Polymorphie könnte eine `maximum`-Funktion sein, die zwei `Becher`-Objekte übergeben bekommt und den Becher mit mehr Inhalt zurückliefert:

```
Becher* maximum(Becher* b1, Becher* b2) {
    if(b1->getFuellmenge()>=b2->getFuellmenge())
        return(b1);
    else
        return(b2);
}
```

Listing 11.15 Eine `maximum`-Funktion für »Becher«

Die Funktion ist so geschrieben, dass bei gleicher Füllmenge das erste Objekt zurückgegeben wird, genau so verhalten sich auch die Funktionen der C++-Standardbibliothek.

Wegen der Polymorphie können `maximum` beliebige `Becher` übergeben werden, solange deren Klassen von `Becher` abgeleitet sind. Diese Art der Programmierung bietet ein enormes Maß an Wiederverwendbarkeit, denn es kann Programmcode geschrieben werden, der mit Objekten arbeitet, an die der Programmierer bis dato nicht einmal gedacht hat.

11.8 Virtuelle Methoden

Im Rahmen der Polymorphie können Effekte auftreten, die vielleicht nicht auf Anhieb klar sind. Nehmen wir folgendes Beispiel:

```
BecherMitAufdruck b("Milch", 300, 90, "Privattasse");
Becher *bptr = &b;
bptr->ausgabe();
```

Was wird ausgegeben? Oder programmtechnisch gefragt, welche `ausgabe`-Methode wird aufgerufen, die von `Becher` oder die von `BecherMitAufdruck`?

Es gibt zwei Argumente:

- ▶ Die `ausgabe-Methode` von `BecherMitAufdruck` wird aufgerufen, weil das Objekt ein `BecherMitAufdruck-Objekt` ist.
- ▶ Die `ausgabe-Methode` von `Becher` wird aufgerufen, weil der Zeiger vom Typ `Becher*` ist.

Im Normalfall werden Datentypen zur Kompilationszeit geprüft. Dies wird *statische Typüberprüfung* genannt. Und zur Kompilationszeit zeigt `bptr` auf ein `Becher-Objekt`, weil `bptr` vom Typ `Becher*` ist. Es spielt dabei keine Rolle, dass dem Zeiger während des Programmlaufs ein `BecherMitAufdruck-Objekt` zugewiesen wird.

Es wird also die `ausgabe-Methode` von `Becher` aufgerufen. Obwohl programmtechnisch nachvollziehbar, ist das Verhalten nicht unbedingt erwünscht. Denn nur, weil der Zugriff auf das Objekt über einen Zeiger vom Typ `Becher*` stattfindet, sollte trotzdem die richtige `ausgabe-Methode` aufgerufen werden.

Um die richtige `ausgabe-Methode` aufzurufen, muss das Programm den tatsächlichen Typ des Objekts, auf das `bptr` zeigt, zur Laufzeit prüfen. Dies wird *dynamische Typüberprüfung* genannt.

Aktiviert wird die dynamische Typüberprüfung für eine Methode, indem vor der Deklaration in der Basisklasse das Schlüsselwort `virtual` geschrieben wird. Daher wird eine solche Methode in C++ auch virtuelle Methode genannt.

Für die dynamische Typüberprüfung spielt es keine Rolle, wie weit der tatsächliche Typ in der Klassenhierarchie von der Basisklasse entfernt ist. Sollte beispielsweise von der Klasse `BecherMitAufdruck` eine Klasse `BecherMitAufdruckUndBild` abgeleitet werden, würde über den `Becher-Zeiger`, wenn er auf ein solches Objekt zeigt, die `ausgabe-Methode` von `BecherMitAufdruckUndBild` aufgerufen – falls die Methode in der Klasse existiert. Es muss sich aber um eine tatsächliche Überschreibung handeln; die Methode in der Subklasse muss genau so heißen wie die überschriebene Methode und in Parameterliste und Rückgabetypp komplett übereinstimmen.

Virtuelle Methoden gewährleisten, dass sich das Verhalten eines Objekts nicht ändert, wenn über einen Basisklassenzeiger darauf zugegriffen wird.

Dynamische Typüberprüfung funktioniert nur mit Methoden. Ein Grund mehr, auf den direkten Attributzugriff zu verzichten.

11.9 UML

Um im weiteren Verlauf des Buches eine möglichst klare und einfache Darstellungsmöglichkeit für Klassen und deren Beziehungen einsetzen zu können, die darüber hinaus auch noch weit verbreitet ist, wollen wir diesen Abschnitt dem Klassendiagramm der UML⁴ widmen. Die UML definiert einen Satz an Diagrammen, mit denen dynamische und statische Eigenschaften von Methoden und Klassen dargestellt werden können. Die bisher verwendeten Diagramme zur Darstellung des Programmflusses in diesem Buch sind Aktivitätsdiagramme der UML.

Um den statischen Aufbau von Klassen und deren Beziehungen untereinander darzustellen, wird das UML-Klassendiagramm eingesetzt. In Abbildung 11.3 ist die aktuelle Becher-Hierarchie als UML-Klassendiagramm dargestellt.

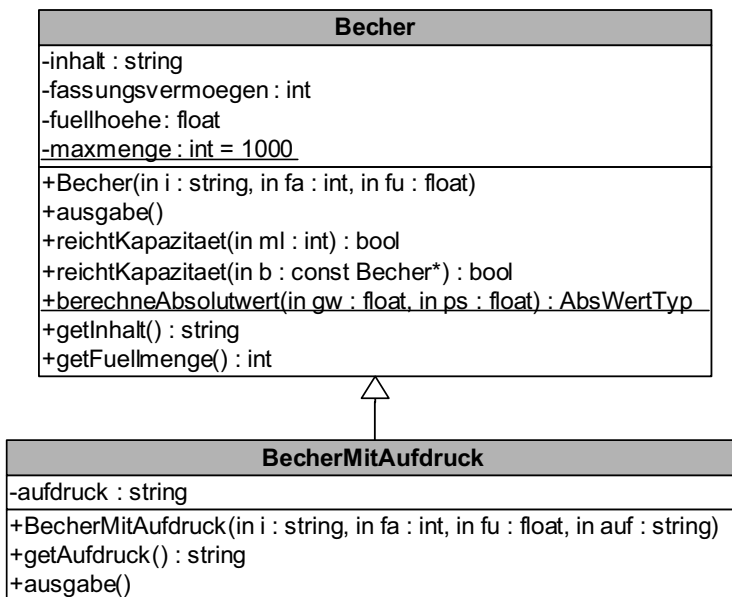


Abbildung 11.3 Die »Becher«-Hierarchie im UML-Klassendiagramm

Im Klassendiagramm wird eine Klasse durch ein in drei Bereiche aufgeteiltes Rechteck repräsentiert. Der obige Bereich beinhaltet den Klassennamen, darunter stehen die Attribute und im untersten Abschnitt die Methoden. Vor den Attributen und Methoden wird mit einem Zeichen das Zugriffsrecht des Elements angezeigt. Dabei steht

⁴ Abkürzung für »Unified Modeling Language«.

- ▶ - für privates Zugriffsrecht
- ▶ # für geschütztes Zugriffsrecht
- ▶ + für öffentliches Zugriffsrecht

Hinter Attributen steht, durch einen Doppelpunkt getrennt, deren Typ und dahinter optional mit Gleichheitszeichen der Initialisierungswert. Der Initialisierungswert wird nur angegeben, wenn er allgemeingültig ist, was nur bei statischen Elementen der Fall ist.

Statische Elemente sind unterstrichen.

Besitzt eine Methode einen Rückgabotyp, dann steht er, durch Doppelpunkt getrennt, hinter der Parameterliste. Ein Parameter besteht aus seinem Namen und seinem dahinter mit Doppelpunkt folgenden Datentyp.

Optional kann vor dem Parameternamen noch die Übergaberichtung spezifiziert werden. Hier gibt es drei Möglichkeiten:

- ▶ `In` – der Parameter dient nur zur Wertübergabe an die Methode. Für ihn muss beim Aufruf ein gültiges Argument angegeben werden. Entspricht der Wert- oder Adressübergabe in C++.
- ▶ `inout` – für diesen Parameter muss beim Aufruf ein gültiges Argument gegeben werden. Änderungen am Parameter innerhalb der Methode wirken sich jedoch auf das beim Aufruf übergebene Argument aus. Entspricht in C++ der Übergabe als Referenz.
- ▶ `Out` – das an die Methode übergebene Argument dient nur dazu, innerhalb der Methode beschrieben zu werden und muss beim Aufruf keinen gültigen Wert besitzen. Es wird innerhalb der Methode beschrieben und vom Aufrufer ausgelesen. Diese Variante wird von C++ nicht direkt unterstützt, kann aber mit Referenzen simuliert werden.

Die Vererbung wird durch den Generalisierungspfeil zum Ausdruck gebracht, der immer von der Subklasse zur Basisklasse weist.

Im weiteren Verlauf werden jeweils bei Bedarf noch einige Darstellungselemente hinzukommen.

11.10 Schnittstellen

Wir können nun für die Klasse `Becher` eine Bibliothek an Funktionen programmieren⁵ und wissen, dass jede Subklasse von `Becher`, egal wie tief sie in der Hierarchie liegen mag, mit dieser Bibliothek arbeiten kann. Und mit den virtuellen

Methoden ist gewährleistet, dass sich das Verhalten der Subklassenobjekte nicht ändert, wenn über einen Basisklassenzeiger auf sie zugegriffen wird.

Da für die Erweiterung der Klassenhierarchie keine Änderungen an der Bibliothek vorgenommen werden müssen, ist die auf `Becher` basierende Klassenhierarchie beliebig erweiterbar. Auch die Funktionsbibliothek kann um Funktionen erweitert werden, ohne dass die tatsächlichen Subklassen von `Becher` bekannt sein müssen.

Die Vererbung erlaubt also ein hohes Maß an Wiederverwendbarkeit, wie es ohne sie nicht möglich wäre.

Der bisherige Ansatz hat nur noch einen Schönheitsfehler: Jede Klasse, die mit der bestehenden Funktionsbibliothek zusammenarbeiten will, muss von der Klasse `Becher` oder einer ihrer Subklassen abgeleitet werden. Was aber, wenn die Objekte der neuen Klasse gar keine `Becher` sind?

Das Problem entstand dadurch, dass die Funktionsbibliothek für eine konkrete Klasse programmiert wurde. Geschickter wäre es, wenn die Bibliothek nicht von einer konkreten Klasse, sondern lediglich von einer bestimmten Fähigkeit oder Funktionalität abhängig wäre.

Nehmen wir als einfaches Beispiel die Funktionalität der Ausgabe. Wir wollen Funktionen schreiben, die von den zu bearbeitenden Objekten lediglich die Fähigkeit erwarten, ausgegeben werden zu können. Dies soll über eine Methode `ausgabe` geschehen. Wir könnten denselben Fehler wie zuvor begehen und die Funktionen mit Zeigern des Typs `Becher*` versehen, denn `Becher` besitzt eine `ausgabe`-Methode. Wollten wir dann aber vielleicht eine geometrische Form ausgeben, dann müsste diese von `Becher` erben, womit die merkwürdige Aussage »Eine geometrische Form ist ein `Becher`« getätigt würde, nur um die Funktionsbibliothek nutzen zu können.

Um solch unrealistische Beziehungen zu vermeiden, muss die Basisklasse weiter abstrahiert werden. Wenn die Funktionsbibliothek nur eine `ausgabe`-Methode benötigt, dann sollte die oberste Basisklasse auch nur diese `ausgabe`-Methode besitzen.

Wir werden diese Klasse `IAusgabe`⁶ nennen und als Inline-Klasse – also ohne Quellcodedatei – erstellen. Die Header-Datei sieht so aus:

⁵ Später unter .NET wird es keine Funktionen mehr geben. Dort wird eine solche Bibliothek durch statische Methoden einer Klasse realisiert.

⁶ Das vorangestellte »I« steht für »Interface«, auf Deutsch Schnittstelle.

```
#pragma once
#include <iostream>
class IAusgabe {
public:
    virtual void ausgabe() const {
        std::cout << "Basis-Ausgabe" << std::endl;
    }
};
```

Listing 11.16 Die Basisklasse »IAusgabe«

Die Methode `ausgabe` muss als virtuell deklariert werden, damit eine Überschreibung in der Subklasse korrekt aufgerufen wird. Die Klasse steht nicht im Namensbereich `Getraenke`, weil sie auch als Basisklasse anderer Hierarchien dienen kann.

Als primitives Beispiel einer auf dieser Basisklasse fußenden Bibliothek soll die folgende Funktion dienen:

```
void simpleAusgabe(IAusgabe* obj) {
    obj->ausgabe();
}
```

Listing 11.17 Die Funktion »simpleAusgabe«

Zugegeben, diese Funktion ist nicht würdig, in einer praxisrelevanten Bibliothek aufgenommen zu werden, sie demonstriert aber auf einfache Weise, worauf es ankommt. Auch eine komplexe Funktion wird nicht anders auf das Objekt zugreifen.

Noch kann die Klasse `Becher` aber nicht von der neuen Bibliothek bearbeitet werden, denn sie hat `IAusgabe` noch nicht als Basisklasse. Das ist schnell nachgeholt:

```
#pragma once
#include "IAusgabe.h"
#include <string>

namespace Getraenke {

    class Becher : public IAusgabe {
        // Klasseninhalt
    };
}
```

Listing 11.18 »Becher« als Subklasse von »IAusgabe«

Alles bestens. Aber immer noch hat der Ansatz drei Schönheitsfehler, die eine gemeinsame Ursache teilen. Betrachten wir `IAusgabe` etwas genauer. Sie besitzt eine Methode `ausgabe`, auf die die Klassenbibliothek zugreift und die von den Subklassen mit ihren spezifischen Methoden überschrieben wird. Die Methode in `IAusgabe` wird aber nie etwas Sinnvolles ausgeben können, weil sie nicht wissen kann, welche Klassen alle von ihr ableiten. Trotzdem mussten wir die Methode mit einem Anweisungsblock ausstatten, der einen sinnlosen Text ausgibt. Gut, die Ausgabe des Textes hat theatralische Gründe, denn der Anweisungsblock hätte auch leer bleiben können. Eine leere Methode ist aber auch kein sehr viel sauberer Ansatz.

Aber was wirklich verrückt ist: Obwohl die Klasse nichts macht, kann von ihr ein Objekt erzeugt und an eine der Bibliotheksfunktionen übergeben werden:

```
IAusgabe o;
simpleAusgabe(&o);
```

Und noch schlimmer: Da eine von ihr abgeleitete Klasse die unsinnige `ausgabe`-Methode erbt, ist die Subklasse nicht gezwungen, eine eigene `ausgabe`-Methode zu implementieren:

```
class Teddybaer : public IAusgabe {
    std::string fellfarbe;
public:
    Teddybaer(std::string ff)
        : fellfarbe(ff)
    {}
};
```

Listing 11.19 Die Klasse »Teddybaer«

Wenn jetzt jemand einen Teddybären mit der gewünschten Fellfarbe konstruiert, dann wird er bei der Ausgabe vermutlich mit einer Erwähnung eben jener Farbe rechnen, aber Pustekuchen:

```
Teddybaer baer("schwarz");
baer.ausgabe(); // Aufruf von IAusgabe::ausgabe
```

All das sind Auffälligkeiten im Verhalten der Klassenhierarchie, die im Optimalfall vermieden werden sollten.

11.10.1 Rein virtuelle Methoden

Es gibt in der Objektorientierten Programmierung glücklicherweise die Möglichkeit, die Existenz einer Methode einzufordern, ohne sie selbst implementieren zu müssen. Solche Methoden werden *abstrakte Methoden* genannt. C++ bezeichnet

sie auch als *rein virtuelle Methoden*. Deklariert werden sie durch ein `=0` hinter dem Methodenkopf. Schauen wir uns das einmal praktisch an der Klasse `IAusgabe` an:

```
class IAusgabe {
public:
    virtual void ausgabe() const =0;
};
```

Listing 11.20 Die Klasse »IAusgabe« mit rein virtueller Methode

Es wird auffallen, dass es sich bei der rein virtuellen Methode nur noch um eine Deklaration handelt. Eine Definition – also ausführbarer Programmcode – ist nicht mehr notwendig. Doch die abstrakte Methode hat weitreichende Konsequenzen:

Eine Klasse mit abstrakten Methoden ist eine abstrakte Klasse. Von abstrakten Klassen kann kein Objekt erzeugt werden.

Demnach kann `IAusgabe` nicht mehr instanziiert werden:

```
IAusgabe o; // Fehler
```

Eine abstrakte Methode wird erst dann konkret, wenn sie in einer Subklasse mit einer konkreten Methode – also einer Methode mit Definition – überschrieben wurde.

Dies ist bisher in der Klasse `Teddybaer` noch nicht geschehen. Die Klasse hat die abstrakte Methode von `IAusgabe` geerbt und ist damit ebenfalls eine abstrakte Klasse. Um einen Teddybären erzeugen zu können, muss `ausgabe` überschrieben werden:

```
class Teddybaer : public IAusgabe {
    std::string fellfarbe;
public:
    Teddybaer(std::string ff)
        : fellfarbe(ff)
    {}
    void ausgabe() const {
        std::cout << "Baer mit Fellfarbe " << fellfarbe
            << std::endl;
    }
};
```

Listing 11.21 Die Klasse »Teddybaer« mit eigener »ausgabe«-Methode

Nun kann auch wieder ein `Teddybaer`-Objekt erzeugt werden. Auf diese Weise wird jeder Programmierer, der von `IAusgabe` ableitet, gezwungen, eine `ausgabe`-Methode zu programmieren. Leider kann er nicht gezwungen werden, eine sinnvolle Methode zu implementieren, aber dass er eine implementieren muss, ist viel wert. Denn die auf `ausgabe` zurückgreifende Bibliothek hat die Gewissheit, dass immer eine `ausgabe`-Methode vorhanden ist. Denn eine Klasse ohne `ausgabe`-Methode ist abstrakt; von ihr kann kein Objekt erzeugt werden.

Eine abstrakte Klasse muss dabei nicht zwangsläufig von der direkt folgenden Subklasse implementiert werden. Im Gegenteil, eine Subklasse kann noch weitere abstrakte Methoden hinzufügen, wie `IGeometrischeFigur` zeigt:

```
namespace Geometrie {
    class IGeometrischeFigur : public IAusgabe {
    public:
        virtual double umfang() const =0;
        virtual double flaeche() const =0;
    };
}
```

Listing 11.22 Die Klasse »IGeometrischeFigur«

Die Klasse `IGeometrischeFigur` könnte die Basisklasse einer anderen Bibliothek sein, die mit Flächen und Umfängen arbeitet. Weil sie von `IAusgabe` abgeleitet ist, können ihre Subklassen zusätzlich noch die `Ausgabe`-Bibliothek verwenden, die bisher nur aus der Methode `simpleAusgabe` besteht. Um `IGeometrischeFigur` von anderen Klassen abzugrenzen, wurde sie in einen eigenen Namensbereich `Geometrie` gesetzt. Eine Subklasse von `IGeometrischeFigur` muss die abstrakten Methoden von `IGeometrischeFigur` und `IAusgabe` implementieren, damit Objekte von ihr erzeugt werden können. Als Beispiel sei hier die Klasse `Rechteck` vorgestellt:

```
namespace Geometrie {
    class Rechteck : public IGeometrischeFigur {
        double x,y,b,h;
    public:
        Rechteck(double x, double y, double b, double h)
            : x(x), y(y), b(b), h(h)
        {}
        void ausgabe() const {
            std::cout << "Rechteck mit Breite " << b
                << " und Hoehe " << h << std::endl;
        }
        double umfang() const {
            return(2*b+2*h);
        }
    };
}
```

```

    }
    double flaeche() const {
        return(b*h);
    }
};
}

```

Listing 11.23 Die Klasse »Rechteck«

Etwas merkwürdig mag der Konstruktor erscheinen, weil die Parameter denselben Namen wie die Attribute haben. Obwohl es problemlos funktioniert, sollte es in der Praxis vermieden werden. Da Sie in Ihrer C++-Karriere aber aller Wahrscheinlichkeit nach auch Code anderer Personen in die Finger bekommen werden, kann es nicht schaden, in loser Folge einige Unarten zu demonstrieren, damit Sie gewappnet sind. Deswegen gleich eine Frage: Welches `h` wird in der Ausgabe des Konstruktors unten ausgegeben?

```

Rechteck(double x, double y, double b, double h)
: x(x), y(y), b(b), h(h) {
    std::cout << h << std::endl;
}

```

In diesem Fall spielt es natürlich keine Rolle, welches `h` ausgegeben wird, weil beide den gleichen Wert haben. Es gibt aber Situationen, in denen die Frage entscheidend ist.

Die Frage kann mit einer anderen Frage beantwortet werden: Welches `h` ist lokaler? Eindeutig der Parameter. Deshalb wird in der Ausgabe auch der Inhalt des Parameters ausgegeben. Aber wie kann das Attribut trotzdem angesprochen werden, obwohl es vom Parameter verdeckt ist? Wir verwenden einfach den Objektzeiger `this`⁷:

```

Rechteck(double x, double y, double b, double h)
: x(x), y(y), b(b), h(h) {
    std::cout << h << std::endl;        // Parameter
    std::cout << this->h << std::endl; // Attribut
}

```

Schauen wir uns abschließend noch in Abbildung 11.4 die aktuelle Klassenhierarchie an:

⁷ Abschnitt 10.4.2.

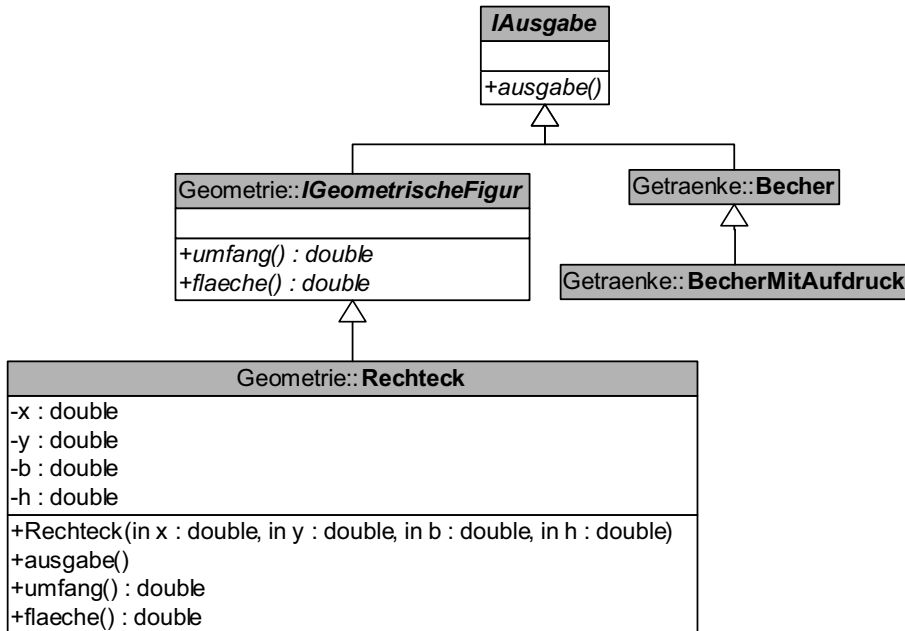


Abbildung 11.4 Die aktuelle Klassenhierarchie

Die Klassen `Becher` und `BecherMitAufdruck` sind ohne ihre Attribute und Methoden dargestellt, weil sich an ihnen nichts geändert hat. Wo vorhanden, sind die Namensbereiche der Klassen mit angegeben.

Im Diagramm ist zu erkennen, dass abstrakte Klassen und Methoden kursiv dargestellt werden.

11.10.2 Rein abstrakte Klassen

Klassen, die als Elemente nur abstrakte Methoden besitzen, werden als *rein abstrakte Klassen* bezeichnet. Sie entsprechen in Standard-C++ den Schnittstellen, die nicht direkt von C++ unterstützt werden, unter .NET (Abschnitt 18.5, »Schnittstellen«) aber eine wichtige Rolle spielen.

Grundsätzlich gilt als Regel – von der es natürlich auch Ausnahmen gibt –, dass ganz oben in der Klassenhierarchie immer eine Schnittstelle bzw. eine abstrakte Klasse stehen sollte.

11.11 Downcasts

Betrachten wir folgenden Codeschnipsel:

```
BecherMitAufdruck b("Tee", 200, 95, "Top-Becher");
Becher * bptr=&b;
```

Wir wissen, dass wegen der Polymorphie die Adresse des `BecherMitAufdruck`-Objekts in einem Zeiger des Typs `Becher*` gespeichert werden kann. Über diesen Zeiger können alle Methoden von `Becher` aufgerufen werden. Sollte es sich um virtuelle Methoden handeln, dann wird die Methode des tatsächlichen Typs aufgerufen, wie in Abschnitt 11.8, »Virtuelle Methoden«, gezeigt wurde.

Über den `Becher`-Zeiger können aber keine Methoden aufgerufen werden, die in der Subklasse neu angelegt wurden, wie in diesem Fall `getAufdruck`.

Manchmal ist aber genau das notwendig. Dazu muss der ursprüngliche Typ des Objekts über eine Typumwandlung wiederhergestellt werden. Der dafür notwendige Cast nennt sich `dynamic_cast`:

```
BecherMitAufdruck *p =
    dynamic_cast<BecherMitAufdruck*>(bptr);
```

Die in `bptr` gespeicherte Adresse wird in den Typ `BecherMitAufdruck*` umgewandelt und dem Zeiger `p` zugewiesen. Wir wissen an dieser Stelle, dass `bptr` tatsächlich auf ein `BecherMitAufdruck`-Objekt zeigt. Häufig wird der `dynamic_cast` aber in Situationen angewandt, bei denen keine hundertprozentige Klarheit darüber herrscht, von welchem Typ das Objekt wirklich ist. Sollte das Objekt nicht von dem Typ sein, in den umgewandelt wird, schlägt die Umwandlung fehl. Es ist in der Praxis daher meist nötig, den Erfolg der Umwandlung zu überprüfen. Sollte die Umwandlung misslingen, liefert der Cast einen Nullzeiger zurück:

```
if(p)
    cout << p->getAufdruck() << endl;
```

Der Aufruf der `BecherMitAufdruck`-Methode wird nur dann ausgeführt, wenn die Umwandlung erfolgreich war.

11.12 Zusammenfassung

Dieses Kapitel hat Ihnen den Mechanismus der Vererbung nähergebracht. Vererbung wird hauptsächlich eingesetzt, um bestehende Klassen zu erweitern.

Die bestehende Klasse wird Basisklasse genannt, die erweiterte Klasse Subklasse. Bei der hier besprochenen öffentlichen Vererbung werden die öffentlichen und

geschützten Elemente der Basisklasse zu öffentlichen und geschützten Elementen der Subklasse. Diese erbt zwar auch die privaten Elemente der Basisklasse, hat aber keinen Zugriff darauf.

Der Konstruktor der Subklasse muss einen Basisklassenkonstruktor aufrufen, andernfalls wird implizit der Standardkonstruktor der Basisklasse aufgerufen.

Geerbte Methoden können überschrieben werden. Soll auch bei einem Aufruf über einen Basisklassenzeiger die überschreibende Methode der Subklasse aufgerufen werden, muss die Basisklassenmethode als virtuell deklariert werden.

Methoden können als abstrakt (rein virtuell) deklariert werden. Eine abstrakte Methode besitzt keinen Anweisungsblock und macht ihre Klasse ebenfalls abstrakt. Von einer abstrakten Klasse können keine Objekte erzeugt werden.

Eine von einer abstrakten Klasse abgeleitete Klasse ist ebenfalls abstrakt, bis sie die geerbten abstrakten Methoden mit eigenen konkreten Methoden überschreibt.

11.13 Übungen

1. Sie möchten ein Programm schreiben, mit dem Sie Ihre auf Musikkassetten, CDs und DVDs befindliche Musiksammlung verwalten können. Erstellen Sie entsprechende Klassen. Welche Elemente besitzen diese Klassen und wie stehen die Klassen im Zusammenhang? Berücksichtigen Sie bei Ihrem Design typische Aktivitäten bei der Arbeit mit einem Archiv, wie »Finde alle Alben eines Interpreten«, Finde das Lied mit dem Titel XYZ« etc.
2. Sie planen, systematisch Ihre Vorfahren und Verwandten elektronisch zu erfassen und benötigen dazu ein Programm, mit dem die verwandtschaftlichen Verhältnisse datentechnisch abgebildet werden können. Überlegen Sie sich sinnvolle Klassen.

*Wer weinende Erben hinterlassen will,
darf keine Lebensversicherung abschließen.
– Marcel Pagnol*

18 Vererbung II

Vererbung war bereits Thema in Kapitel 11. An dieser Stelle soll lediglich besprochen werden, wie sich unter .NET die Vererbung syntaktisch verändert und in ihren Möglichkeiten erweitert hat.

Zur Demonstration sollen einfache, aber sinnlose Klassen erhalten, bei denen der Fokus auf das programmtechnische Verhalten gelegt werden kann. Begonnen wird mit der Klasse `Basis`, die später die Rolle der Basisklasse übernehmen soll:

```
ref class Basis {
    int basis;
public:
    Basis(int i)
        : basis(i)
    {}
    property int Wert {
        int get() {
            return(basis);
        }
        void set(int i) {
            basis=i;
        }
    }
    void ausgabe() {
        System::Console::WriteLine("Basis:"+basis);
    }
};
```

Listing 18.1 Die Klasse `Basis`

Sie besitzt als Attribut einen `int`-Wert, der über den Konstruktor initialisiert wird. Für den Zugriff steht eine Eigenschaft `Wert` und die Methode `ausgabe` zur Verfügung. Sie wissen bereits, wie eigene Klassen mit der `Console`-Klasse ausge-

geben werden können¹, um aber nicht bereits eine geerbte Methode überschreiben zu müssen (denn jede verwaltete Klasse hat `Object` als Basisklasse und erbt damit die `ToString`-Methode), wird zu Anschauungszwecken eine eigene Ausgabemethode implementiert. Ein Objekt von `Basis` ist schnell angelegt und die Methoden ausgetestet:

```
Basis^ b=gcnew Basis(20);
b->ausgabe();
Console::WriteLine(b->Wert);
```

Nun folgt eine von `Basis` abgeleitete Klasse mit dem passenden Namen `Abgeleitet`. Sie definiert eine eigene `ausgabe`-Methode und eine neue Eigenschaft `Wert`:

```
ref class Abgeleitet : Basis {
    int ab;
public:
    Abgeleitet(int i, int a)
        : Basis(i), ab(a)
    {}
    property int Wert {
        int get() {
            return(ab);
        }
        void set(int i) {
            ab=i;
        }
    }
    void ausgabe() {
        System::Console::WriteLine("Abgeleitet:"+ab);
    }
};
```

Listing 18.2 Die Klasse »Abgeleitet«

Ein Unterschied zu ANSI C++ tritt bereits zutage: Weil es unter .NET nur öffentliche Vererbung gibt, kann auf das Schlüsselwort `public` vor der Basisklasse verzichtet werden.

Auch von der abgeleiteten Klasse ist ein Objekt problemlos erstellt:

```
Abgeleitet^ a=gcnew Abgeleitet(20,30);
a->ausgabe();
Console::WriteLine(a->Wert);
```

Als Nächstes soll über einen Basisklassenzeiger zugegriffen werden:

¹ Abschnitt 17.2, »Ausgabe«

```
b=a;
b->ausgabe();
Console::WriteLine(b->Wert);
```

Über den Basisklassenzeiger verhält sich das Abgeleitet-Objekt wie ein Basis-Objekt. Mit den grundlegenden Vererbungsmechanismen bereits vertraut, erkennen Sie die Ursache. Die Methoden der Basisklasse sind nicht virtuell.

Basisklassen sollten die potenziell für Überschreiben infrage kommenden Methoden immer als virtuell deklarieren. Das folgende Listing zeigt die Änderungen in komprimierter Darstellung:

```
ref class Basis {
    int basis;
public:
    Basis(int i) : basis(i) {}
    property int Wert {
        virtual int get() {return(basis);}
        virtual void set(int i) {basis=i;}
    }
    virtual void ausgabe() {
        System::Console::WriteLine("Basis:"+basis);
    }
};
```

Listing 18.3 Die Klasse »Basis« mit virtuellen Funktionen

Der Compiler wird sich nun beschweren, dass die Methoden der abgeleiteten Klasse keine Kennzeichnung haben, ob es sich um eine überschreibende oder eine neue Methode handelt. Um das gewünschte Verhalten zu erreichen, müssen die Methoden der Basisklasse überschrieben und die Methoden der abgeleiteten Klasse daher mit `override` versehen werden. Zusätzlich muss die Subklasse ihre Methoden explizit als virtuell deklarieren:

```
ref class Abgeleitet : Basis {
    int ab;
public:
    Abgeleitet(int i, int a) : Basis(i), ab(a) {}
    property int Wert {
        virtual int get() override {return(ab);}
        virtual void set(int i) override {ab=i;}
    }
    virtual void ausgabe() override {
        System::Console::WriteLine("Abgeleitet:"+ab);
    }
};
```

Listing 18.4 Die Klasse »Abgeleitet« mit virtuellen Methoden

18.1 override vs. new

Um den Unterschied zwischen `override` und `new` zu erklären, soll folgende Klassenhierarchie aus vier Klassen erhalten, die allesamt eine Methode `Print` besitzen, mit der ausgegeben wird, um welche Klasse es sich handelt. Die Methoden sind virtuell und überschreiben jeweils die Methode der Basisklasse, damit bei Aufrufen über Basisklassenzeiger die korrekte Methode aufgerufen wird:

```
ref class A {
public:
    virtual void Print() {Console::WriteLine("A");}
};
ref class B : public A {
public:
    virtual void Print() override {Console::WriteLine("B");}
};
ref class C : public B {
public:
    virtual void Print() override {Console::WriteLine("C");}
};
ref class D : public C {
public:
    virtual void Print() override {Console::WriteLine("D");}
};
```

Listing 18.5 Eine Klassenhierarchie

Nun werden zwei `D`-Objekte erzeugt und unterschiedlichen Basisklassenzeigern zugewiesen. Die Ergebnisse der Methodenaufrufe sind wie erwartet:

```
A ^a=gnew D;
C ^c=gnew D;
a->Print(); // Ausgabe D
c->Print(); // Ausgabe D
```

Jetzt wird die Methode in der Klasse `C` mit `new` deklariert:

```
ref class C : public B {
public:
    virtual void Print() new {Console::WriteLine("C");}
};
```

Durch `new` beginnt für die Methode `Print` in Klasse `C` eine neue Klassenhierarchie. Die dynamische Typprüfung endet für `Print` daher vor `C`, also bei Klasse `B`, beziehungsweise beginnt bei `C` neu.

Oder anders ausgedrückt: Die Methode `Print` der Klasse `B` wird in Klasse `C` nicht überschrieben, sondern die Methode `Print` in `C` ist eine komplett neue Methode.

Wenn über ein Trackinghandle des Typs `A` oder `B` jetzt die `Print`-Methode eines Typs `D` aufgerufen wird, dann wird die `Print`-Methode von `B` ausgeführt, da ab `C` eine neue Hierarchie beginnt. Für alle Zeiger ab Klasse `C` bleibt das Verhalten gleich:

```
A ^a=gcnew D;
C ^c=gcnew D;
a->Print(); // Ausgabe B
c->Print(); // Ausgabe D
```

Diese Eigenschaft wird zugegebenermaßen weitaus seltener eingesetzt als `override`.

18.2 Abstrakte Methoden und Klassen

Beim Klassendesign kommt es häufiger vor, dass eine Basisklasse bereits eine gewisse Funktionalität zur Verfügung stellt, zur korrekten Ausführung dieser Funktionalität aber Informationen notwendig sind, die erst in den Subklassen bereitgestellt werden.

Nehmen wir die Klasse `Mitarbeiter`. Sie kann eine Methode `getMonatsgehalt` bereitstellen, die aus dem Jahresgehalt das durchschnittliche Monatsgehalt ermittelt. Wie das Jahresgehalt ausfällt, entscheidet sich aber erst in der Subklasse, weil ein einfacher Angestellter höchstwahrscheinlich eine andere Gehaltsstruktur haben wird als ein Abteilungsleiter. Werfen wir einen Blick auf einen ersten Entwurf der Klasse `Mitarbeiter`:

```
ref class Mitarbeiter {
public:
    float getMonatsgehalt() {
        return(getJahresGehalt()/12);
    }
};
```

Die Klasse wird sich so noch nicht kompilieren lassen, weil der Compiler nicht weiß, wo die Methode `getJahresgehalt` zu finden ist. Sie muss in der Klasse verfügbar gemacht werden, aber in den Subklassen durch eine spezialisiertere Variante ersetzt werden können. Die Wahl fällt daher auf eine virtuelle Methode.

Die Methode kann in `Mitarbeiter` aber keinen sinnvollen Rückgabewert liefern, also muss es eine bereits besprochene rein virtuelle Methode² werden, die auch

² Abschnitt 11.10.1

abstrakte Methode genannt wird. Unter .NET kann zwar auch die aus ANSI C++ bekannte Schreibweise `=0` verwendet werden, passender ist aber das Schlüsselwort **abstract**.

Eine Klasse mit abstrakten Methoden wird automatisch ebenfalls abstrakt, das heißt, von ihr können keine Objekte erzeugt werden. Das macht im Falle der Klasse `Mitarbeiter` auch Sinn, denn ihr fehlt schließlich eine funktionsfähige Methode `getJahresgehalt`. Obwohl die Klasse durch die abstrakte Methode schon abstrakt ist, muss sie unter .NET explizit als abstrakt deklariert werden:

```
ref class Mitarbeiter abstract {
public:
    float getMonatsgehalt() {
        return(getJahresgehalt()/12);
    }

    virtual float getJahresgehalt() abstract;
};
```

Listing 18.6 Die Klasse »Mitarbeiter«

Eine von einer abstrakten Basisklasse abgeleitete Subklasse erbt die abstrakte Methode und ist zunächst auch abstrakt. Erst, wenn alle abstrakten Methoden überschrieben wurden, wird die Klasse konkret und kann instanziiert werden.

Unter .NET ist es auch möglich, eine Klasse ohne abstrakte Methoden als abstrakt zu deklarieren. Eine davon abgeleitete Klasse ist automatisch konkret (weil sie keine abstrakten Methoden geerbt hat).

Doch zurück zu unserer Firmenhierarchie. Um eine konkrete Klasse zu erhalten, soll die Klasse `Abteilungsleiter` abgeleitet werden, die ein Jahresgehalt und Weihnachtsgeld erhält:

```
ref class Abteilungsleiter : Mitarbeiter {
    float jahresgehalt;
    float weihnachtsgeld;
public:
    Abteilungsleiter(float jg, float wg)
        : jahresgehalt(jg), weihnachtsgeld(wg)
    {}
    virtual float getJahresgehalt() override {
        return(jahresgehalt+weihnachtsgeld);
    }
};
```

Listing 18.7 Die Klasse »Abteilungsleiter«

Von dieser Klasse kann ein Objekt erzeugt und die geerbte Methode `getMonatsgehalt` aufgerufen werden:

```
Abteilungsleiter^ p=gnew Abteilungsleiter(4000,2000);
Console.WriteLine(p->getMonatsgehalt());
```

18.3 Versiegelte Methoden

Eine Methode ist versiegelt, wenn sie nicht mehr überschrieben werden kann.

Falls Sie zu dem Schluss kommen sollten, die Methode `getJahresgehalt` in `Abteilungsleiter` sei der Weisheit letzter Schluss und einfach undenkbar, dass jemand eine noch bessere oder sinnvollere Methode implementieren könnte, dann deklarieren Sie die Methode als `sealed`:

```
ref class Abteilungsleiter : Mitarbeiter {
    float jahresgehalt;
    float weihnachtsgeld;
public:
    Abteilungsleiter(float jg, float wg)
        : jahresgehalt(jg), weihnachtsgeld(wg)
    {}
    virtual float getJahresgehalt() override sealed {
        return(jahresgehalt+weihnachtsgeld);
    }
};
```

Listing 18.8 Eine versiegelte Methode in »Abteilungsleiter«

Nur virtuelle Methoden können versiegelt werden.

18.4 Versiegelte Klassen

Es ist aber auch möglich, eine komplette Klasse zu versiegeln. Von dieser Klasse kann dann nicht mehr abgeleitet werden.

In der darzustellenden Firmenhierarchie gibt es vielleicht keine Erweiterung mehr von `Abteilungsleiter` und Sie wollen eventuellen Versuchen, doch noch eine Subklasse zu erstellen, im Vorfeld bereits eine Abfuhr erteilen, dann deklarieren Sie einfach die Klasse als `sealed`:

```
ref class Abteilungsleiter sealed : Mitarbeiter {
    float jahresgehalt;
```

```

    float weihnachtsgeld;
public:
    Abteilungsleiter(float jg, float wg)
        : jahresgehalt(jg), weihnachtsgeld(wg)
    {}
    virtual float getJahresgehalt() override sealed {
        return(jahresgehalt+weihnachtsgeld);
    }
};

```

Listing 18.9 Die versiegelte Klasse »Abteilungsleiter«

Das `sealed` bei der Methode `getJahresgehalt` könnte theoretisch eingespart werden, denn wenn von einer Klasse nicht mehr abgeleitet werden kann, dann ist auch das Überschreiben einer ihrer Methoden nicht mehr möglich.

Die beiden Klassen funktionieren einwandfrei, sind aber nicht gerade .NET-mäßig programmiert. Unter .NET sollten die Zugriffsmethoden durch Eigenschaften ersetzt werden:

```

ref class Mitarbeiter abstract {
public:
    property float Monatsgehalt {
        float get() {
            return(Jahresgehalt/12);
        }
    }
    property float Jahresgehalt {
        virtual float get() abstract;
    }
};

ref class Abteilungsleiter sealed : Mitarbeiter {
    float jahresgehalt;
    float weihnachtsgeld;
public:
    Abteilungsleiter(float jg, float wg)
        : jahresgehalt(jg), weihnachtsgeld(wg)
    {}
    property float Jahresgehalt {
        virtual float get() override sealed {
            return(jahresgehalt+weihnachtsgeld);
        }
    }
};

```

Listing 18.10 »Mitarbeiter« und »Abteilungsleiter« mit Eigenschaften

Die `get`- und `set`-Funktionen der Eigenschaften können ebenso virtuell, abstrakt oder versiegelt sein wie normale Methoden. Wichtig ist nur, dass sich die Deklarationen auf die einzelnen Funktionen der Eigenschaft beziehen, nicht auf die gesamte Eigenschaft. Eine Eigenschaft könnte theoretisch eine abstrakt-virtuelle `get`-Funktion und eine versiegelte `set`-Funktion besitzen.

18.5 Schnittstellen

In Abschnitt 11.10, »Schnittstellen«, wurde das Grundprinzip von Schnittstellen bereits besprochen. Hier soll die Definition einer Schnittstelle unter .NET behandelt werden. Schnittstellen kommen immer dann zum Einsatz, wenn an eine Subklasse gestellte Anforderungen von der Basisklasse nicht geleistet werden können oder sollen. Gewissermaßen war das auch Thema des vorangegangenen Abschnitts, denn die Klasse `Mitarbeiter` hat von ihren Subklassen eine Eigenschaft `Jahresgehalt` gefordert.

In ANSI C++ ist eine Schnittstelle nichts anderes als eine Klasse, die nur aus rein virtuellen Methoden besteht. Unter .NET existiert ein eigenes Konstrukt, das mit dem Schlüsselwort `interface` eingeleitet wird. Im Folgenden wird eine Schnittstelle `IAusgebbar` definiert, die eine Methode `ausgeben` fordert:

```
interface class IAusgebbar {
    void ausgeben();
};
```

Listing 18.11 Definition einer Schnittstelle

Eine Klasse, die von der Schnittstelle ableitet – im Zusammenhang mit Schnittstellen wird die Formulierung »Eine Klasse implementiert eine Schnittstelle« verwendet – ist abstrakt, bis sie die von der Schnittstelle geforderten Methoden implementiert. Die in der Schnittstelle deklarierten Methoden entsprechen in ihrem Verhalten einer abstrakten Methode.

Syntaktisch gibt es zu Klassen und deren Methoden aber einige Besonderheiten:

- ▶ Die Methoden einer Schnittstelle sind automatisch öffentlich, eine Angabe von `public` ist nicht notwendig. Das ist logisch, denn nur eine öffentliche Methode ist überschreibbar und von außen zugänglich.
- ▶ Um eine Methode sinnvoll überschreiben bzw. implementieren zu können, muss sie virtuell sein. Eine Methode ohne Definition ist darüber hinaus auch noch abstrakt. Aus diesem Grund ist eine Schnittstellenmethode immer virtuell und abstrakt, ohne dass die entsprechenden Schlüsselwörter angegeben werden müssen.

- ▶ Schnittstellen dürfen Deklarationen von Methoden, Eigenschaften und deren Funktionen sowie Ereignissen³ enthalten.
- ▶ Eine Schnittstelle darf keine Definitionen – also Anweisungsblöcke – für Methoden, Eigenschaftsfunktionen oder Ereignisse besitzen.
- ▶ Die Definition von statischen Elementen (Attribute, Eigenschaften, Methoden, Ereignisse) ist allerdings erlaubt.
- ▶ Von einer Schnittstelle können Wert- und Verweisklassen sowie andere Schnittstellen abgeleitet werden.
- ▶ Die implementierende Methode der Subklasse darf kein Schlüsselwort `override` besitzen, da die Schnittstellenmethode nicht überschrieben, sondern implementiert wird.

Die Klasse `Ganzzahl` implementiert die Schnittstelle `IAusgebbar`:

```
ref class Ganzzahl : IAusgebbar {
    int zahl;
public:
    Ganzzahl(int i)
        : zahl(i)
    {}
    virtual void ausgeben() {
        System::Console::WriteLine(zahl);
    }
};
```

Listing 18.12 Die Klasse »Ganzzahl«

Wie oben erwähnt, ist die implementierte Methode virtuell, besitzt aber kein `override`.

18.5.1 Schnittstellen und Mehrfachvererbung

.NET unterstützt zwar keine Mehrfachvererbung, aber eine Klasse kann mehrere Schnittstellen implementieren. In den meisten Fällen lässt sich ein auf Mehrfachvererbung basierendes Klassendesign auf die Mehrfachimplementierung von Schnittstellen reduzieren, von daher treten die Vorteile einer echten Mehrfachvererbung gegenüber der damit einhergehenden Nachteile und Schwierigkeiten in den Hintergrund.

Zur Demonstration soll das obige Beispiel um eine Schnittstelle `IGanzzahlig` ergänzt werden, die `Ganzzahl` dann ebenfalls implementiert. Für Abwechslung sorgt der Einsatz einer Eigenschaft anstelle einer Methode:

³ Eigenschaften sind Thema von Abschnitt 21.2, »Ereignisse«.

```
interface class IGanzzahlig {
    property int Zahl {
        int get();
    }
};
```

Listing 18.13 Die Schnittstelle »IGanzzahlig«

Es folgt die Implementierung in der Klasse:

```
ref class Ganzzahl : IAusgebbar, IGanzzahlig {
    int zahl;
public:
    Ganzzahl(int i)
        :zahl(i)
    {}
    virtual void ausgeben() {
        System::Console::WriteLine(zahl);
    }
    property int Zahl {
        virtual int get() {
            return(zahl);
        }
    }
};
```

Listing 18.14 Mehrfachimplementierung bei »Ganzzahl«

Die zu implementierenden Schnittstellen werden einfach durch Komma getrennt im Klassenkopf aufgeführt.

18.5.2 Identische Methoden

Wenn es möglich ist, in einer Klasse mehrere Schnittstellen zu implementieren, dann kann es theoretisch auch passieren, dass zwei Schnittstellen zwei Methoden mit demselben Namen und derselben Signatur vorgeben:

```
interface class A {
    void ausgabe();
};

interface class B {
    void ausgabe();
};
```

Listing 18.15 Zwei Schnittstellen mit derselben Methode

Wenn diese beiden Methoden dieselbe Bestimmung haben – es also ausreicht, für beide Methoden eine Implementierung zu besitzen – unterscheidet sich der Mechanismus nicht von dem bisher angewandten:

```
ref class C : A, B {
public:
    virtual void ausgabe() {
        Console::WriteLine("Ausgabe");
    }
};
```

Listing 18.16 Gemeinsame Implementierung

Unerheblich, über welche Schnittstelle die Methode ausgerufen wird, das Ergebnis ist dasselbe:

```
C^ c=gcnew C;
A^ a=c;
B^ b=c;
a->ausgabe();
b->ausgabe();
```

Es ist aber durchaus möglich, dass die Methoden der beiden Schnittstellen, obwohl gleichen Namens, eine unterschiedliche Implementierung benötigen. Dazu ist eine spezielle Syntax notwendig, weil es keine zwei Methoden im selben Gültigkeitsbereich mit gleichem Namen und Signatur geben kann:

```
ref class C : A, B {
public:
    virtual void ausgabea() = A::ausgabe {
        Console::WriteLine("A");
    }
    virtual void ausgabeb() = B::ausgabe {
        Console::WriteLine("B");
    }
};
```

Listing 18.17 Individuelle Implementierung

Die Namen der Methoden in der implementierenden Klasse (hier `ausgabea` und `ausgabeb`) sind beliebig wählbar. Der Aufruf über die Schnittstellentypen bleibt gleich:

```
C^ c=gcnew C;
A^ a=c;
B^ b=c;
```

```
a->ausgabe();  
b->ausgabe();
```

Müssen die Methoden über `C` aufgerufen werden, dann ist der neue Name zu verwenden:

```
c->ausgabea();  
c->ausgabeb();
```

18.6 Zusammenfassung

In diesem Kapitel wurde die .NET-Vererbung unter die Lupe genommen.

Das Überschreiben von virtuellen Methoden muss mit dem Schlüsselwort `override` kenntlich gemacht werden. Zusätzlich gibt es mit `new` die Möglichkeit, eine geerbte Methode nicht zu überschreiben, sondern eine neue Hierarchie für diese Methode zu beginnen.

Schnittstellen werden unter .NET durch eine eigene Konstruktion (`interface`) unterstützt. Die in einer Schnittstelle deklarierten Methoden sind automatisch öffentlich und abstrakt.

Index

(Name) 591
?:-Operator 85

A

Abrunden 384
Absolutwert 383
AcceptButton 440
AcceptsReturn 493
AcceptsTab 489
Activate 444
Activated 445
ActiveForm 440
Add 364, 368
Addition 60
AddOwnedForm 444
AddRange 365
Adressooperator 143
AfterLabelEdit 542, 552
Alignment 537
AllowColumnReorder 537
AllowDrop 415
Anchor 415
AnchorStyles 415
Anweisungsblok 43
Anwendungseinstellungen 32
Appearance 532
Appearance (Aufzählung) 483
Appearance (Eigenschaft) 483
Append 316
AppendAllText 332
AppendFormat 316
AppendText 332, 491
Application
 EnableVisualStyles 410
 Run 410
 SetCompatibleTextRenderingDefault 410
ApplicationException 257
Arcus Cosinus 383
Arcus Sinus 383
Arcus Tangens 383
Array 123, 253
ArrayList 365
 AddRange 365
 BinarySearch 366

GetRange 366
 InsertRange 366
 LastIndexOf 366
 RemoveRange 367
 Reverse 367
 Sort 367
AsyncCompletedEventArgs 488
AsyncCompletedEventHandler 488
Attribut 159
 statisch 175
Aufrufbrowser 181
Aufrunden 383
Aufzählung 285
Ausnahme 219, 256
AutoArrange 537
AutoCheck 483
AutoCompleteCustomSource 493
AutoCompleteMode (Aufzählung) 494
AutoCompleteMode (Eigenschaft) 494
AutoCompleteSource (Aufzählung) 494
AutoCompleteSource (Eigenschaft) 494
AutoEllipsis 478
AutoScroll 436
AutoScrollMargin 436
AutoScrollMinSize 437
AutoScrollPosition 437
AutoSize 416
AutoSizeMode (Aufzählung) 480
AutoSizeMode (Eigenschaft) 480
Autovervollständigung 493
AvailableFreeSpace 326

B

back 128
BackColor 417
BackColorchanged 428
BackgroundImage 417
BackgroundImageChanged 428
BackgroundImageLayout 417
BackgroundImageLayoutchanged 428
bad_alloc 217
Bedingtes Kompilieren 118
Bedingung 74
BeforeLabelEdit 543

begin 129
 BeginUpdate 500
 Bezeichner 51
 Bezeichnungsregeln 51
 Bezugsrahmenoperator 45
 BigMul 383
 BinaryFormatter 341
 Deserialize 342
 Serialize 342
 BinaryReader 338
 BaseStream 338
 Close 338
 PeekChar 338
 Read 338, 339
 ReadBytes 339
 ReadChars 339
 BinarySearch 366
 BinaryWriter 337
 BaseStream 337
 Close 337
 Flush 337
 Seek 337
 Write 338
 Bindungsstärke 63
 Bitmap 469
 Bitweise Operatoren 289
 bool 74, 241
 boolalpha 74
 BorderStyle (Aufzählung) 478
 BorderStyle (Eigenschaft) 478
 Bottom 417
 Bounds 417
 break 100
 Breakpoint 399
 BringToFront 425
 Brush 572
 Brushes 576
 Button 481
 AutoSizeMode 482
 DialogResult 482
 PerformClick 482
 ButtonBase 480
 AutoEllipsis 480
 FlatStyle 480
 Image 481
 ImageAlign 481
 ImageIndex 481
 ImageList 481
 TextAlign 481

TextImageRelation 481

C

CancelAsync 487
 CancelButton 440
 CancelEventArgs 435
 CancelEventHandler 435
 CanOverflow 558
 CanUndo 489
 case 86
 catch 223
 CausesValidation 417
 CausesValidationChanged 428
 ctype 134
 Ceiling 383
 CellBorderStyle 527
 Char 317
 IsControl 317
 IsDigit 317
 IsLetter 317
 IsLetterOrDigit 317
 IsLower 317
 IsNumber 317
 IsPunctuation 317
 IsSeparator 317
 IsSymbol 317
 IsUpper 317
 IsWhiteSpace 317
 char 133, 241
 CharacterCasing (Aufzählung) 494
 CharacterCasing (Eigenschaft) 494
 CheckAlign 483
 CheckBox 482
 Appearance 483
 AutoCheck 483
 CheckAlign 483
 Checked 483
 CheckedChanged 484
 CheckState 483
 CheckStateChanged 484
 ThreeState 484
 Checked 483
 CheckedChanged 484
 CheckState (Aufzählung) 483
 CheckState (Eigenschaft) 483
 CheckStateChanged 484
 cin 59
 getline 137

- ignore* 137
- class 155
- Clear 364, 368, 541
- ClearUndo 491
- Click 429
- ClientRectangle 418
- ClientSize 418
- ClientSizeChanged 428
- Close 337, 338, 340, 341
- Close (Form) 444
- CLR-Konsolenanwendung 245
- CollapseAll 551
- Collect 392
- Collections 357
 - ArrayList* 365
 - DictionaryEntry* 369
 - ICollection* 360
 - IComparer* 357
 - IDictionary* 367
 - IDictionaryEnumerator* 368
 - IEnumerable* 359
 - IEnumerator* 359
 - Queue* 361
 - SortedList* 370
 - Stack* 363
- Color (Klasse) 459
- ColorDialog 618
- ColumnClick 543
- ColumnClickEventArgs 543
- ColumnClickEventHandler 543
- ColumnCount 528
- ColumnHeader 544
- ColumnHeaderStyle 539
- ColumnWidth 499
- ComboBox 502
- Compare 312
- CompareTo 313
- Component 414
- Console 386
 - BackgroundColor* 387
 - Beep* 388
 - BufferHeight* 387
 - BufferWidth* 387
 - CapsLock* 387
 - Clear* 388
 - CursorLeft* 387
 - CursorSize* 387
 - CursorTop* 387
 - CursorVisible* 387
 - ForegroundColor* 387
 - LargestWindowHeight* 387
 - LargestWindowWidth* 387
 - NumberLock* 387
 - ReadLine* 255
 - ResetColor* 389
 - SetBufferSize* 389
 - SetCursorPosition* 389
 - SetWindowPosition* 389
 - SetWindowSize* 389
 - Title* 387
 - WindowHeight* 387
 - WindowLeft* 387
 - WindowTop* 387
 - WindowWidth* 387
 - Write* 249
 - WriteLine* 248
- ConsoleColor 387
- const 54
- Contains 364, 368, 425
- ContainsKey 369, 370
- ContainsValue 369, 370
- ContentAlignment 478
- ContextMenuChanged 428
- ContextMenuStrip (Eigenschaft) 418
- ContextMenuStrip (Klasse) 560
- ContextMenuStripChanged 428
- continue 101
- Control 415
 - AllowDrop* 415
 - Anchor* 415
 - AutoSize* 416
 - BackColor* 417
 - BackColorchanged* 428
 - BackgroundImage* 417
 - BackgroundImageChanged* 428
 - BackgroundImageLayout* 417
 - BackgroundImageLayoutChanged* 428
 - Bottom* 417
 - Bounds* 417
 - BringToFront* 425
 - CausesValidation* 417
 - CausesValidationChanged* 428
 - Click* 429
 - ClientRectangle* 418
 - ClientSize* 418
 - ClientSizeChanged* 428
 - Contains* 425
 - ContextMenuChanged* 428

ContextMenuStrip 418
ContextMenuStripChanged 428
Controls 418
CreateGraphics 425
Cursor 418
Cursorchanged 428
DefaultBackColor 418
DefaultFont 418
DefaultForeColor 418
DisplayRectangle 419
Dock 419
DockChanged 428
DoubleClick 429
DrawToBitmap 425
Enabled 421
EnabledChanged 428
Enter 429
Findform 426
Focus 426
Focused 421
Font 421
Fontchanged 428
ForeColor 421
ForeColorChanged 428
GetChildAtPoint 426
HasChildren 421
Height 422
HelpRequested 430
Hide 426
Invalidate 426
Invalidated 430
IsKeyLocked 427
KeyDown 430
KeyPress 431
KeyUp 430
Leave 432
Left 422
Location 422
LocationChanged 428
Margin 422
MarginChanged 428
MaximumSize 422
MinimumSize 422
ModifierKeys 422
MouseButtons 422
MouseClicked 432
MouseDoubleClick 433
MouseDown 433
MouseEnter 433
MouseHover 433
MouseLeave 433
MouseMove 433
MousePosition 422
MouseUp 434
MouseWheel 434
Move 434
Name 423
Padding 423
PaddingChanged 428
Paint 434
Parent 423
ParentChanged 428
PointToClient 427
PointToScreen 427
RectangleToClient 427
RectangleToScreen 427
Refresh 427
Region 423
RegionChanged 428
Resize 435
Right 423
SendToBack 427
Show 428
Size 423
SizeChanged 428
TabIndex 423
TabIndexChanged 428
TabStop 424
TabStopChanged 428
Tag 424
Text 424
TextChanged 428
Top 424
TopLevelControl 424
UseWaitCursor 424
Validated 435
Validating 435
Visible 425
VisibleChanged 428
Width 425
ControlBox 440
ControlCollection 418
Controls 418
ControlStyles 582
Convert 255
Copy 331, 491
CopyTo 360
Cos 383

Cosh 384
 Count 360
 cout 46
 CreateDirectory 327
 CreateGraphics 425
 cstring 138
 strcpy 138
 strlen 150
 CultureInfo 307
 DateTimeFormat 308
 DisplayName 308
 EnglishName 308
 Name 308
 NumberFormat 308
 OptionalCalendars 308
 Current 360
 CurrentThread 307
 Cursor (Eigenschaft) 418
 Cursor (Klasse) 473
 Cursorchanged 428
 Cursors 474

D

Darstellungsreihenfolge 420
 DataSource 498
 Datatips 398
 Date 322
 Datenelement 159
 Datentyp 50
 char 133
 double 53
 float 53
 int 51
 long 51
 long double 53
 short 51
 unsigned int 51
 unsigned long 51
 unsigned short 51
 DateTime 322
 Date 322
 Day 322
 DayOfWeek 322
 DayOfYear 322
 DaysInMonth 323
 Hour 322
 IsDaylightSavingTime 323
 IsLeapYear 323

Millisecond 322
 Minute 322
 Month 322
 Now 322
 ParseExact 323
 Second 322
 Subtract 325
 Ticks 322
 TimeOfDay 322
 Today 322
 ToString 325
 Year 322
 Day 322
 DayOfWeek 322
 DayOfYear 322
 DaysInMonth 323
 Deactivate 446
 Debugger 397
 Decimal 241
 default 91, 271
 DefaultBackColor 418
 DefaultFont 418
 DefaultForeColor 418
 define 54
 Dekrementoperator 69
 delegate 349
 Delegaten 347
 Invoke 350
 Multicast 350
 Delete 328, 331
 delete 214
 Dequeue 362
 Dereferenzierungsoperator 145
 Deselected 533
 Deselecting 534
 Deserialize 342
 DesktopBounds 441
 DesktopLocation 441
 Destruktor 214, 273
 Dialog
 eigener 585
 DialogResult (Aufzählung) 441
 DialogResult (Eigenschaft) 441, 482
 DictionaryEntry 369
 Directory 327, 333
 CreateDirectory 327
 Delete 328
 Exists 328
 GetCreationTime 328

- GetCurrentDirectory* 328
 - GetDirectories* 328
 - GetDirectoryRoot* 329
 - GetFiles* 329
 - GetFileSystemEntries* 329
 - GetLastAccessTime* 329
 - GetLastWriteTime* 329
 - GetLogicalDrives* 330
 - GetParent* 330
 - Move* 330
 - SetCreationTime* 328
 - SetCurrentDirectory* 330
 - SetLastAccessTime* 329
 - SetLastWriteTime* 330
 - DirectoryInfo 331
 - DisplayMember 498
 - DisplayRectangle 419
 - Dispose 273
 - Division 60
 - do 96
 - Dock 419
 - Dockchanged 428
 - DockStyle 419
 - double 53, 241
 - Double Buffering 582
 - DoubleClick 429
 - Downcast 210
 - DrawArc 577
 - DrawEllipse 577
 - DrawIcon 577
 - DrawIconUnstretched 578
 - DrawImage 578
 - DrawImageUnscaled 578
 - DrawLine 578
 - DrawPie 578
 - DrawPolygon 578
 - DrawRectangle 579
 - DrawString 579
 - DrawToBitmap 425
 - DriveFormat 326
 - DriveInfo 325
 - AvailableFreeSpace* 326
 - DriveFormat* 326
 - DriveType* 326
 - GetDrives* 325
 - IsReady* 326
 - Name* 326
 - RootDirectory* 326
 - TotalFreeSpace* 326
 - TotalSize* 326
 - VolumeLabel* 326
 - DriveType 325, 326
 - dynamic_cast 210
 - Dynamische Speicherverwaltung 213
 - delete* 214
 - new* 213
 - Dynamische Typüberprüfung 199
- ## E
-
- E (Eulersche Zahl) 386
 - Eigener Dialog 585
 - Eigenschaften 265
 - Indexer* 270
 - Elementfunktion 162
 - Elementinitialisierungsliste 167
 - Elementverweis ändern 591
 - Elementzugriffsoperator 126
 - else 77
 - Enabled 421
 - EnabledChanged 428
 - endif 118
 - endl 46
 - EndsWith 313
 - EndUpdate 500
 - Enqueue 362
 - Enter 429
 - Entweder-Oder-Verzweigung 77
 - enum 286
 - Environment 389
 - CommandLine* 390
 - CurrentDirectory* 390
 - Exit* 390
 - GetCommandLineArgs* 390
 - GetEnvironmentVariable* 390
 - GetEnvironmentVariables* 391
 - GetFolderPath* 392
 - MachineName* 390
 - OSVersion* 390
 - ProcessorCount* 390
 - SetEnvironmentVariable* 392
 - SystemDirectory* 390
 - TickCount* 390
 - UserDomainName* 390
 - UserInteractive* 390
 - UserName* 390
 - Version* 390
 - EnvironmentVariableTarget 391

Equals 317
 erase 129
 Ereignis-Handler 351
 Ereignisse 351
 ErrorImage 486
 Escape-Sequenz 47
 EVA-Prinzip 49
 event 351
 Event handler 351
 EventArgs 428
 EventHandler 428
 Events 351
 Exception 256
 Exists 328, 330
 Exit 390
 Exklusives Oder 84
 Exp 384
 ExpandAll 551
 Explizite Typumwandlung 65
 Extension 330

F

Fallunterscheidung 86
 false 74
 File 331

- AppendAllText* 332
- AppendText* 332
- Copy* 331
- Delete* 331
- Move* 331
- ReadAllBytes* 332
- ReadAllLines* 332
- ReadAllText* 333
- WriteAllBytes* 333
- WriteAllLines* 333
- WriteAllText* 333

 FileAccess 336
 FileDialog 513
 FileInfo 331

- Directory* 333
- DirectoryName* 333
- IsReadOnly* 333
- Length* 333

 FileMode 335
 FileShare 336
 FileStream 335

- CanRead* 336
- CanSeek* 336
- CanWrite* 336
- Length* 336
- Position* 336

 FileSystemInfo 330

- Attributes* 330
- CreationTime* 330
- Exists* 330
- Extension* 330
- FullName* 330
- LastAccessTime* 330
- LastWriteTime* 330

 FillEllipse 579
 FillPie 579
 FillPolygon 580
 FillRectangle 580
 Finalizer 275
 finally 259
 find 139
 FindForm 426
 FixedPanel (Aufzählung) 529
 FixedPanel (Eigenschaft) 529
 Flag 288
 FlatStyle (Aufzählung) 478
 FlatStyle (Eigenschaft) 478
 float 53, 241
 Floor 384
 FlowDirection (Aufzählung) 527
 FlowDirection (Eigenschaft) 526
 FlowLayoutPanel 526

- FlowDirection* 526
- WrapContents* 527

 Flush 337, 340
 Focus 426
 Focused 421
 FolderBrowserDialog 616
 Font (Eigenschaft) 421
 Font (Klasse) 464
 FontChanged 428
 FontDialog 619
 FontStyle 464
 for 98
 for each 254
 ForeColor 421
 ForeColorChanged 428
 Form

- AcceptButton* 440
- Activate* 444
- Activated* 445
- ActiveForm* 440

- AddOwnedForm* 444
 - CancelButton* 440
 - Close* 444
 - ControlBox* 440
 - Deactivate* 446
 - DesktopBounds* 441
 - DesktopLocation* 441
 - DialogResult* 441
 - FormBorderStyle* 441
 - FormClosed* 446
 - FormClosing* 446
 - HelpButton* 442
 - HelpButtonClicked* 447
 - Icon* 442
 - Load* 447
 - MainMenuStrip* 442
 - MaximizeBox* 443
 - MinimizeBox* 443
 - Modal* 443
 - Opacity* 443
 - OwneForms* 443
 - RemoveOwnedForm* 445
 - ResizeBegin* 447
 - ResizeEnd* 447
 - SetDesktopBounds* 445
 - SetDesktopLocation* 445
 - Show* 445
 - ShowDialog* 445
 - ShowIcon* 443
 - ShowInTaskbar* 444
 - StartPosition* 444
 - TopMost* 444
 - Format 315
 - FormatException 256
 - FormBorderStyle (Aufzählung) 442
 - FormBorderStyle (Eigenschaft) 441
 - FormClosed 446
 - FormClosedEventArgs 446
 - FormClosedEventHandler 446
 - FormClosing 446
 - FormClosingEventArgs 446
 - FormClosingEventHandler 446
 - Forms
 - ButtonBase* 480
 - LabelEditEventArgs* 542
 - ToolStripDropDownButton* 563
 - ToolStripSplitButton* 567
 - FormStartPosition 444
 - FromImage 580
 - FullRowSelect 538
 - Funktionen 103
 - Funktionskopf 43
- ## G
-
- Garbage Collection 392
 - GC 392
 - Collect* 392
 - GetTotalMemory* 393
 - KeepAlive* 393
 - ReRegisterForFinalize* 393
 - SuppressFinalize* 277, 393
 - gcnew 253
 - Generic 372
 - List* 372
 - generic 373
 - get 266
 - GetByIndex 370
 - GetChildAtPoint 426
 - GetCommandLineArgs 390
 - GetCreationTime 328
 - GetCurrentDirectory 328
 - GetDirectories 328
 - GetDirectoryRoot 329
 - GetDrives 325
 - GetEnumerator 359, 368
 - GetEnvironmentVariable 390
 - GetFiles 329
 - GetFileSystemEntries 329
 - GetFolderPath 392
 - GetKey 370
 - GetKeyList 371
 - GetLastAccessTime 329
 - GetLastWriteTime 329
 - getline 137, 138
 - GetLogicalDrives 330
 - GetParent 330
 - GetRange 366
 - GetTotalMemory 393
 - GetValueList 371
 - Globalization
 - CultureInfo* 307
 - Graphics 577
 - DrawArc* 577
 - DrawEllipse* 577
 - DrawIcon* 577
 - DrawIconUnstretched* 578
 - DrawImage* 578

DrawImageUnscaled 578
DrawLine 578
DrawPie 578
DrawPolygon 578
DrawRectangle 579
DrawString 579
FillEllipse 579
FillPie 579
FillPolygon 580
FillRectangle 580
FromImage 580
MeasureString 580
 GridLines 538
 GripMargin 558
 GripStyle 558
 Groß-/Kleinschreibung 43
 GroupBox 479
 AutoSizeMode 480
 Groups 538
 GrowStyle 528
 Grundrechenarten 60

H

Haltepunkt 399
 HasChildren 421
 Hashtable
 ContainsKey 369
 ContainsValue 369
 HatchBrush 575
 Hauptfunktion 43, 248
 Header-Datei
 cctype 134
 iostream 48
 vector 126
 HeaderStyle 538
 Height 422
 HelpButton 442
 HelpButtonClicked 447
 HelpEventArgs 430
 HelpEventHandler 430
 HelpRequested 430
 Hide 426
 HideSelection 489, 539
 HorizontalAlignment (Aufzählung) 495
 HorizontalScroll 437
 HotTracking 539
 Hour 322
 HoverSelection 539

HScrollProperties 437

I

ICollection 360
 CopyTo 360
 Count 360
 IComparable 358
 IComparer 357
 Icon (Eigenschaft) 442
 Icon (Klasse) 470
 IDictionary 367
 Add 368
 Clear 368
 Contains 368
 GetEnumerator 368
 IsFixedSize 368
 IsReadOnly 368
 Item 368
 Keys 368
 Remove 368
 Values 368
 IDictionaryEnumerator 368
 IDisposable 273
 IEnumerable 359
 GetEnumerator 359
 IEnumerator 359
 Current 360
 MoveNext 360
 Reset 360
 if 75
 ifdef 119
 ifndef 118
 ignore 137
 IList
 Add 364
 Clear 364
 Contains 364
 IndexOf 364
 Insert 364
 IsFixedSize 364
 IsReadOnly 364
 Item 364
 Remove 364
 RemoveAt 364
 Image 468
 Image (Eigenschaft) 478
 ImageAlign 478
 ImageIndex 479

ImageLayout 417
 ImageList 479
 ImageList (Klasse) 472
 ImageLocation 486
 ImageScalingSize 558
 in 254
 include 48, 115
 Indexer 270
 IndexOf 313, 364
 IndexOfAny 314
 IndexOfKey 371
 IndexOfValue 371
 Indexoperator 123, 128
 InitialImage 487
 InitializeComponents 412
 Inklusives Oder 80
 Inkrementoperator 69
 inline 163
 Insert 310, 316, 364
 InsertRange 366
 int 51, 241
 IntegralHeight 499
 IntelliSense 127
 interface 301
 Invalidate 426
 Invalidated 430
 InvalidateEventArgs 430
 InvalidateEventHandler 430
 Invoke 350
 IO 321
 iostream 48
 isalnum 134
 isalpha 134
 IsControl 317
 IsDaylightSavingTime 323
 IsDigit 317
 isdigit 134
 IsFixedSize 364, 368
 IsKeyLocked 427
 IsLeapYear 323
 IsLetter 317
 IsLetterOrDigit 317
 IsLower 317
 islower 134
 IsNumber 317
 IsPunctuation 317
 IsReadOnly 333, 364, 368
 IsReady 326
 IsSeparator 317

isspace 134
 IsSplitterFixed 529
 IsSymbol 317
 IsUpper 317
 isupper 134
 IsWhiteSpace 317
 Item 364, 368
 ItemActivate 543
 ItemActivation 536
 ItemCheck 543
 ItemChecked 544
 ItemCheckedEventArgs 544
 ItemCheckedEventHandler 544
 ItemCheckEventArgs 543
 ItemCheckEventHandler 543
 Items 499

K

KeepAlive 393
 KeyDown 430
 KeyEventArgs 431
 KeyEventHandler 430
 KeyPress 431
 KeyPressEventArgs 431
 KeyPressEventHandler 431
 Keys 368
 KeyUp 430
 Klasse 155
 abstrakt 205, 297
 Attribut 159
 Datenelement 159
 Destruktor 214, 273
 Elementfunktion 162
 Finalizer 275
 Konstruktor 165
 Methode 162
 rein abstrakt 209
 versiegelt 299
 versiegelte Methode 299
 Verweisklasse 261
 Wertklasse 278
 Zugriffsrecht 160
 Zugriffsspezifizierer 160
 Klassenansicht 178
 Aufrufbrowser 181
 Memberbereich 179
 Objektbereich 179
 Objektbrowser 180

Verweissuche 179
 Kommentar
 einzeilig 55
 mehrzeilig 55
 Kompilationsvorgang 115
 Konfigurations-Manager 41
 Konstante 54
 Konstanzwahrende Methode 168
 Konstruktor 165
 Elementinitialisierungsliste 167
 Kosinus 383
 Kosinus Hyperbolicus 384
 Kurzschlusseigenschaft 82

L

Label 477
 AutoEllipsis 478
 BorderStyle 478
 FlatStyle 478
 Image 478
 ImageAlign 478
 ImageIndex 479
 ImageList 479
 TextAlign 479
 LabelEdit 539
 LabelEditEventArgs 542
 LabelEditEventHandler 542
 Labelwrap 539
 LargeImageList 539
 LastIndexOf 314, 366
 LastIndexOfAny 314
 LayoutStyle 558
 Leave 432
 Left 422
 Length 309, 315, 333
 LinearGradientBrush 574
 Lines 489
 Linker 114
 LinkLabel 479
 ListBox 498
 BeginUpdate 500
 BorderStyle 499
 ClearSelected 501
 ColumnWidth 499
 EndUpdate 500
 FindString 501
 FindStringExact 501
 GetSelected 501
 HorizontalScrollbar 499
 IntegralHeight 499
 Items 499
 MultiColumn 499
 ScrollAlwaysVisible 500
 SelectedIndexChanged 502
 SelectedIndices 500
 SelectedItem 500
 SelectedItems 500
 SelectionMode 500
 SetSelected 501
 Sorted 500
 ListControl 497
 DataSource 498
 DisplayMember 498
 SelectedIndex 498
 SelectedValue 498
 ValueMember 498
 ListView 535
 Activation 536
 AfterLabelEdit 542
 Alignment 537
 AllowColumnReorder 537
 AutoArrange 537
 BeforeLabelEdit 543
 BeginUpdate 541
 BorderStyle 537
 CheckBoxes 538
 CheckedIndices 538
 CheckedItems 538
 Clear 541
 ColumnClick 543
 Columns 538
 EndUpdate 541
 FindItemWithText 542
 FullRowSelect 538
 GridLines 538
 Groups 538
 HeaderStyle 538
 HideSelection 539
 HotTracking 539
 HoverSelection 539
 ItemActivate 543
 ItemCheck 543
 ItemChecked 544
 Items 539
 LabelEdit 539
 LabelWrap 539
 LargeImageList 539

- ListViewItemSorter* 540
 - MultiSelect* 540
 - Scrollable* 540
 - SelectedIndexChanged* 544
 - SelectedIndices* 540
 - SelectedItems* 540
 - Showgroups* 540
 - ShowItemToolTips* 540
 - SmallImageList* 539
 - Sort* 542
 - Sorting* 540
 - StateImageList* 541
 - TopItem* 541
 - View* 541
 - ListViewAlignment 537
 - ListViewColumns 538
 - ListViewGroup 545
 - ListViewItem 545
 - ListViewItemSorter 540
 - ListViewSubItem 547
 - Load 447, 487
 - LoadAsync 488
 - LoadCompleted 488
 - LoadProgressChanged 488
 - Location 422
 - LocationChanged 428
 - Logarithmus 384
 - Logische Operatoren 80
 - Lokale Variablen 105
 - long 51, 241
 - long double 53
 - long long 241
- M**
-
- MachineName 390
 - main 43, 248
 - MainMenuStrip 442
 - Manipulatoren
 - boolalpha* 74
 - noboolalpha* 74
 - Margin 422
 - MarginChanged 428
 - MaskedTextBox 495
 - Math 382
 - Abs* 383
 - Acos* 383
 - Asin* 383
 - Atan* 383
 - BigMul* 383
 - Ceiling* 383
 - Cos* 383
 - Cosh* 384
 - E* 386
 - Exp* 384
 - Floor* 384
 - Log* 384
 - Log10* 384
 - Max* 384
 - Min* 384
 - PI* 386
 - Pow* 384
 - Round* 385
 - Sign* 385
 - Sin* 385
 - Sinh* 385
 - Sqrt* 385
 - Tan* 386
 - Tanh* 386
 - MaximizeBox 443
 - Maximum 384
 - MaximumSize 422
 - MaxLength 489
 - MeasureString 580
 - Mehrfachdeklaration 117
 - MenuStrip 559
 - MessageBox 466
 - MessageBoxButtons 466
 - MessageBoxDefaultButton 467
 - MessageBoxIcon 467
 - Methode 162
 - abstrakt* 205, 297
 - konstanzwährend* 168
 - rein virtuell* 205, 297
 - statisch* 173
 - überladen* 170
 - versiegelt* 299
 - virtuell* 199, 295
 - Millisecond 322
 - MinimizeBox 443
 - Minimum 384
 - MinimumSize 422
 - Minute 322
 - Modal 443
 - Modified 489
 - ModifierKeys 422
 - Modulo-Operator 68
 - Month 322

MouseButton (Aufzählung) 432
 MouseButton (Eigenschaft) 422
 MouseClick 432
 MouseDoubleClick 433
 MouseDown 433
 MouseEnter 433
 MouseEventArgs 432
 MouseEventHandler 432
 MouseHover 433
 MouseLeave 433
 MouseMove 433
 MousePosition 422
 MouseUp 434
 MouseWheel 434
 Move 330, 331, 434
 MoveNext 360
 Multicast-Delegaten 350
 MultiColumn 499
 Multiline 490, 532
 Multiplikation 60
 MultiSelect 540
 mutable 170

N

Name 326, 423
 Namensbereich 45, 182
 namespace 49, 182
 Negationsoperator 84
 new 213, 296
 Next 382
 NextBytes 382
 NextDouble 382
 noboolalpha 74
 NodeLabelEventArgs 552
 NodeLabelEventHandler 552
 NodeMouseClick 553
 NodeMouseDoubleClick 553
 Nodes 550
 Now 322
 npos 140
 nullptr 250

O

Object 264
 Equals 369
 GetHashCode 369
 Objektbrowser 180

Oder
 exklusiv 84
 inklusive 80
 Opacity 443
 OpenFileDialog 516
 Operatoren
 Addition 60
 Adressoperator 143
 Bezugsrahmenoperator 45
 Bindungsstärke 63
 bitweise Operatoren 289
 Dekrement 69
 Dereferenzierungsoperator 145
 Division 60
 Elementzugriffsoperator 126
 Indexoperator 123
 Inkrement 69
 Kurzschlusseigenschaft 82
 logische Operatoren 80
 Modulo 68
 Multiplikation 60
 Negationsoperator 84
 Postdecrement 70
 Postinkrement 70
 Prädecrement 70
 Präinkrement 70
 Restwert 68
 Subtraktion 60
 Und 80
 Vergleichsoperatoren 75
 Zeigeroperator 148
 zusammengesetzte Zuweisung 66
 Zuweisung 60
 Orientation 530
 OverflowException 256
 override 295
 OwnedForms 443

P

Padding (Eigenschaft) 423
 Padding (Klasse) 475
 PaddingChanged 428
 PadLeft 311
 PadRight 311
 Paint 580
 Paint (Ereignis) 434
 PaintEventArgs 434
 PaintEventHandler 434

- Panel 525
 - BorderStyle* 525
 - TabStop* 526
 - Parameterbezeichner 251
 - Parent 423
 - ParentChanged 428
 - ParseExact 323
 - PasswordChar 495
 - Paste 492
 - PathSeparator 550
 - Peek 341, 362, 363
 - PeekChar 338
 - Pen 576
 - PerformClick 482
 - PI 386
 - PictureBox 486
 - BorderStyle* 486
 - CancelAsync* 487
 - ErrorImage* 486
 - Image* 486
 - ImageLocation* 486
 - InitialImage* 487
 - Load* 487
 - LoadAsync* 488
 - LoadCompleted* 488
 - LoadProgressChanged* 488
 - SizeMode* 487
 - WaitOnLoad* 487
 - PictureBoxSizeMode 487
 - Point (Klasse) 455
 - PointToClient 427
 - PointToScreen 427
 - Pop 363
 - pop_back 128
 - Postdekrement 70
 - Postinkrement 70
 - Potenz 384
 - Pow 384
 - Prädekrement 70
 - pragma 117
 - endregion* 412
 - once* 117
 - region* 412
 - Präinkrement 70
 - Präprozessor 48
 - Präprozessordirektive
 - define* 54
 - endif* 118
 - ifdef* 119
 - ifndef* 118
 - include* 48, 115
 - pragma* 117
 - Priorität 63
 - private 161
 - ProcessorCount 390
 - ProgressBar 502
 - Increment* 504
 - Maximum* 503
 - Minimum* 503
 - PerformStep* 504
 - Step* 503
 - Style* 503
 - Value* 503
 - ProgressChangedEventArgs 488
 - ProgressChangedEventArgsHandler 488
 - Projekt
 - Anlegen mit Projektmappe* 30
 - bereinigen* 36
 - CLR-Konsolenanwendung anlegen* 245
 - Datei hinzufügen* 40
 - Datei kopieren* 40
 - erstellen* 35, 36
 - Fehler finden* 38
 - Klasse hinzufügen* 155
 - kompilieren* 35
 - neu erstellen* 36
 - Neue Datei* 33
 - starten* 37
 - Startprojekt festlegen* 41
 - verwaltete Klasse hinzufügen* 261
 - Win32-Konsolenanwendung anlegen* 30
 - Windows Forms-Anwendung anlegen* 407
 - Zu Projektmappe hinzufügen* 40
 - Projektmappe 30
 - öffnen* 39
 - Projekt hinzufügen* 40
 - Startprojekt festlegen* 41
 - property 266
 - protected 161, 196
 - public 161
 - Push 363
 - push_back 128
- ## Q
-
- Quadratwurzel 385
 - Queue 361

Dequeue 362
Enqueue 362
Peek 362
ToArray 362

R

RadioButton 484
 Appearance 485
 AutoCheck 485
 CheckAlign 485
 Checked 485
 CheckedChanged 485
 PerformClick 485
 Random 381
 Next 382
 NextBytes 382
 NextDouble 382
 Read 338, 339, 341
 ReadAllBytes 332
 ReadAllLines 332
 ReadAllText 333
 ReadBytes 339
 ReadChars 339
 ReadLine 255, 341
 ReadOnly 490
 ReadToEnd 341
 Rectangle 456
 RectangleToClient 427
 RectangleToScreen 427
 ref 261
 Referenz 151
 Refresh 427
 Region (Eigenschaft) 423
 RegionChanged 428
 Remove 311, 317, 364, 368
 RemoveAt 364, 371
 RemoveOwnedForm 445
 RemoveRange 367
 Replace 310, 316
 ReRegisterForFinalize 393
 Reset 360
 Resize 435
 ResizeBegin 447
 ResizeEnd 447
 ResizeRedraw 582
 Restwertoperator 68
 return 110
 Reverse 367

rfind 139
 RichTextBox 496
 Right 423
 RootDirectory 326
 Round 385
 RowCount 528, 533
 Rückgabewert 110
 Runden 385

S

SaveFileDialog 516
 Schleife 93
 do 96
 for 98
 while 93
 Schnittstelle 209, 301
 Scroll 438
 ScrollableControl 435
 AutoScroll 436
 AutoScrollMargin 436
 AutoScrollMinSize 437
 AutoScrollPosition 437
 HorizontalScroll 437
 Scroll 438
 ScrollControlIntoView 438
 VerticalScroll 438
 ScrollAlwaysVisible 500
 ScrollBars (Aufzählung) 495
 ScrollBars (Eigenschaft) 495
 ScrollControlIntoView 438
 ScrollEventArgs 438
 ScrollEventHandler 438
 ScrollEventType 439
 ScrollOrientation 439
 ScrollProperties 437
 sealed 299
 Second 322
 Selected 534
 SelectedIndex 498
 SelectedIndexChanged 502, 544
 SelectedIndices 500
 SelectedItem 500
 SelectedNode 550
 SelectedText 490
 SelectedValue 498
 Selecting 534
 SelectionLength 490
 SelectionMode (Aufzählung) 500

- SelectionMode (Eigenschaft) 500
- SelectionStart 490
- SendToBack 427
- SerializableAttribute 342
- Serialize 342
- set 266
- SetByIndex 371
- SetCurrentDirectory 330
- SetDesktopBounds 445
- SetDesktopLocation 445
- SetStyle 582
- short 51, 241
- ShortcutsEnabled 490
- Show (Control) 428
- Show (Form) 445
- ShowDialog 445
- ShowGroups 540
- ShowIcon 443
- ShowInTaskbar 444
- ShowItemToolTips 540
- Sign 385
- Sinus 385
- Sinus Hyperbolicus 385
- size 127, 139
- Size (Eigenschaft) 423
- Size (Klasse) 454
- SizeChanged 428
- SizeMode 487
- SizingGrip 560
- SmallImageList 539
- SolidBrush 572
- Sort 367
- Sorted 500
- SortedList 370
 - ContainsKey* 370
 - ContainsValue* 370
 - GetByIndex* 370
 - GetKey* 370
 - GetKeyList* 371
 - GetValueList* 371
 - IndexOfKey* 371
 - IndexOfValue* 371
 - RemoveAt* 371
 - SetByIndex* 371
- SortOrder 540
- Split 309
- SplitContainer 528
 - BorderStyle* 529
 - FixedPanel* 529
 - IsSplitterFixed* 529
 - Orientation* 530
 - Panel1* 530
 - Panel1Collapsed* 530
 - Panel1MinSize* 530
 - Panel2* 530
 - Panel2Collapsed* 530
 - Panel2MinSize* 530
 - SplitterWidth* 530
 - SplitterDistance* 530
 - SplitterIncrement* 530
 - SplitterMoved* 531
 - SplitterMoving* 531
- SplitterCancelEventArgs 531
- SplitterCancelEventHandler 531
- SplitterDistance 530
- SplitterEventArgs 531
- SplitterEventHandler 531
- SplitterIncrement 530
- SplitterMoved 531
- SplitterMoving 531
- SplitterWidth 530
- Sqrt 385
- Stack 363
 - Peek* 363
 - Pop* 363
 - Push* 363
 - ToArray* 363
- Standardkonstruktor 166
- StartsWith 313
- StateImageList 541
- static 174
- static_cast 65
- statische Typüberprüfung 199
- StatusStrip 560
 - SizingGrip* 560
- std
 - cin* 59
 - cout* 46
 - endl* 46
- stdafx.h 248
- strcpy 138
- StreamReader 340
 - Close* 341
 - Peek* 341
 - Read* 341
 - ReadLine* 341
 - ReadToEnd* 341
- StreamWriter 339

- AutoFlush* 339
 - BaseStream* 339
 - Close* 340
 - Flush* 340
 - Write* 340
 - WriteLine* 340
 - String 308
 - Compare* 312
 - CompareTo* 313
 - EndsWith* 313
 - Format* 315
 - IndexOf* 313
 - IndexOfAny* 314
 - Insert* 310
 - LastIndexOf* 314
 - LastIndexOfAny* 314
 - Length* 309
 - PadLeft* 311
 - PadRight* 311
 - Remove* 311
 - Replace* 310
 - Split* 309
 - StartsWith* 313
 - Substring* 309
 - ToLower* 310
 - ToUpper* 310
 - Trim* 312
 - string 138
 - find* 139
 - npos* 140
 - rfind* 139
 - size* 139
 - substr* 139
 - StringBuilder 315
 - Append* 316
 - AppendFormat* 316
 - Equals* 317
 - Insert* 316
 - Length* 315
 - Remove* 317
 - Replace* 316
 - substr 139
 - Substring 309
 - Subtract 325
 - Subtraktion 60
 - Suffix 53
 - SuppressFinalize 393
 - switch 86
 - System 248
 - Console* 386
 - Environment* 389
 - GC* 392
 - SystemDirectory 390
 - SystemException 257
 - SystemIcons 471
- ## T
-
- TabAppearance 532
 - TabControl 532
 - Appearance* 532
 - Deselected* 533
 - Deselecting* 534
 - ImageList* 532
 - Multiline* 532
 - RowCount* 533
 - Selected* 534
 - SelectedIndex* 533
 - SelectedTab* 533
 - Selecting* 534
 - ShowToolTips* 533
 - TabCount* 533
 - TabPage* 533
 - TabControlAction 534
 - TabControlCancelEventArgs 534
 - TabControlCancelEventHandler 534
 - TabControlEventArgs 533
 - TabControlEventHandler 533
 - TabIndex 423
 - TabIndexChanged 428
 - TableLayoutPanel 527
 - CellBorderStyle* 527
 - ColumnCount* 528
 - GrowStyle* 528
 - RowCount* 528
 - TableLayoutPanelCellBorderStyle 527
 - TableLayoutPanelGrowStyle 528
 - TabPage 534
 - ImageIndex* 535
 - ToolTipText* 535
 - TabStop 424
 - TabStopChanged 428
 - Tag 424
 - Tangens 386
 - Template 227
 - Text 315, 424
 - TextAlign 479
 - TextBox 493

- AcceptsReturn* 493
- AutoCompleteCustomSource* 493
- AutoCompleteMode* 494
- AutoCompleteSource* 494
- CharacterCasing* 494
- PasswordChar* 495
- ScrollBars* 495
- TextAlign* 495
- TextBoxBase
 - AcceptsTab* 489
 - AppendText* 491
 - BorderStyle* 489
 - CanUndo* 489
 - Clear* 491
 - ClearUndo* 491
 - Copy* 491
 - Cut* 491
 - DeselectAll* 491
 - GetCharIndexFromPosition* 491
 - GetFirstCharIndexFromLine* 491
 - GetFirstCharIndexOfCurrentLine* 492
 - GetLineFromCharIndex* 492
 - HideSelection* 489
 - Lines* 489
 - MaxLength* 489
 - Modified* 489
 - Multiline* 490
 - Paste* 492
 - ReadOnly* 490
 - Select* 492
 - SelectAll* 492
 - SelectedText* 490
 - SelectionLength* 490
 - SelectionStart* 490
 - ShortcutsEnabled* 490
 - TextLength* 490
 - Undo* 492
 - WordWrap* 490
- TextChanged 428
- TextImageRelation (Aufzählung) 481
- TextImageRelation (Eigenschaft) 481
- TextLength 490
- TextureBrush 573
- this 164
- Thread
 - CurrentThread* 307
- Threading 307
 - Timer* 394
 - TimerCallback* 394
- ThreeState 484
- throw 220
- Tick 322
- TickCount 390
- Ticks 322
- TimeOfDay 322
- Timer 394
- TimerCallback 394
- ToArray 362, 363
- Today 322
- ToLower 310
- tolower 135
- ToolStrip 557
 - CanOverflow* 558
 - GripMargin* 558
 - GripStyle* 558
 - ImageList* 558
 - ImageScalingSize* 558
 - Items* 558
 - LayoutStyle* 558
 - ShowItemToolTips* 559
 - Stretch* 559
- ToolStripButton 563
- ToolStripComboBox 563
- ToolStripContainer 568
- ToolStripDropDownButton 563
- ToolStripGripStyle 558
- ToolStripItem 561
- ToolStripLabel 564
- ToolStripLayoutStyle 559
- ToolStripMenuItem 565
- ToolStripProgressBar 566
- ToolStripSeparator 567
- ToolStripStatusLabel 564
- ToolStripTextBox 567
- Top 424
- TopItem 541
- TopLevelControl 424
- TopMost 444
- TotalFreeSpace 326
- TotalSize 326
- ToUpper 310
- toupper 135
- Trackinghandle 249
- Trackingreferenz 250
- TreeNode 553
- TreeNodeMouseClickEventArgs 553
- TreeNodeMouseClickEventHandler 553
- TreeView 548

AfterCheck 551
AfterCollapse 551
AfterExpand 551
AfterLabelEdit 552
AfterSelect 551
BeforeCheck 552
BeforeCollapse 552
BeforeExpand 552
BeforeLabelEdit 553
BeforeSelect 552
BeginUpdate 551
CollapseAll 551
EndUpdate 551
ExpandAll 551
GetNodeCount 551
ImageIndex 549
Indent 549
ItemHeight 550
LineColor 550
NodeMouseClicked 553
NodeMouseDoubleClick 553
Nodes 550
PathSeparator 550
SelectedNode 550
ShowLines 550
ShowPlusMinus 550
ShowRootLines 550
Sort 551
TreeViewAction 552
TreeViewCancelEventArgs 553
TreeViewCancelEventHandler 552
TreeViewEventArgs 552
TreeViewEventHandler 551
 Trigonometrische Funktionen 382
 Trim 312
 true 74
 try 223
 typename 228, 373
 Typüberprüfung 199
 Typumwandlung
 explizit 65

U

Überladen
 Methode 170
 UML
 Aktivitätsdiagramm 201
 Klassendiagramm 201

Umwandlungsoperatoren 284
 Und 80
 Undo 492
 unsigned char 241
 unsigned int 51, 241
 unsigned long 51, 241
 unsigned long long 241
 unsigned short 51, 241
 UserName 390
 UseWaitCursor 424
 using 49

V

Validated 435
 Validating 435
 value 278
 ValueMember 498
 Values 368
 Variablen 49
 vector 126
 back 128
 begin 129
 erase 129
 Indexoperator 128
 pop_back 128
 push_back 128
 size 127
 Vererbung 187
 Vergleichsoperatoren 75
 VerticalScroll 438
 Verweisklasse 261
 Verzweigung 73
 View (Aufzählung) 541
 View (Eigenschaft) 541
 virtual 199, 295
 Visible 425
 VisibleChanged 428
 Visual C++
 Aufrufbrowser 181
 CLR-Konsolenanwendung anlegen 245
 Erststart 28
 Installation 25
 Klasse hinzufügen 155
 Klassenansicht 178
 kompilieren 35
 Neue Datei hinzufügen 33
 Objektbrowser 180
 Projekt anlegen 30

Projekt erstellen 35
Projekt hinzufügen 40
Projektmappe öffnen 39
Startprojekt festlegen 41
verwaltete Klasse hinzufügen 261
Win32-Konsolenanwendung anlegen 30
Windows Forms-Anwendung anlegen

407
void 103
VolumeLabel 326
VScrollProperties 438

W

WaitOnLoad 487
wchar_t 241
wenn-dann 75
Wertklasse 278
while 93
Width 425
Win32-Konsolenanwendung 30
Windows Forms-Anwendung 407
WordWrap 490

Write 249, 338, 340
WriteAllBytes 333
WriteAllLines 333
WriteAllText 333
WriteLine 248, 340
Wurzel 385

Y

Year 322

Z

Zeiger 144
Zeigerarithmetik 150
Zeigeroperator 148
z-Reihenfolge 420
Zufallszahlen 381
Zugriffsrecht 160
Zugriffsspezifizierer 160
Zugriffstaste 424
Zuweisungsoperator 60
zusammengesetzt 66