Einführung in C++

ladedu.com/cpp

Einführung in C++

Heinz Tschabitscher

et tu es toujours là

Inhaltsverzeichnis

Vorwort **xiii**

1 Einführung **1**
1.1 Die Wurzeln von C++ 1
1.2 Wie fange ich an? 1
1.3 Wie verwende ich diese Einführung? 2
1.4 Programmieraufgaben 3

2 Aller Anfang ist leicht **5**
2.1 Sogar die Kommentare sind in C++ verbessert 5
2.2 Die Schlüsselwörter const und volatile 5
2.3 Der Gültigkeitsbereichsoperator 6
2.4 Die iostream Bibliothek 7
2.5 Mehr über die stream Bibliothek 8
2.6 Der cin Operator 9
2.7 Ein- und Ausgabe mit Dateien 9
2.8 Variablendefinition 10
2.9 Die C++ Referenz 11
2.10 Definitionen sind ausführbar 11
2.11 Defintion und Deklaration 12
2.12 Eine bessere for-Schleife 12
2.13 Die Priorität der Operatoren 13
2.14 Programmieraufgaben 13

3 Zusammengesetzte Datentypen **15**
3.1 Aufzählungstypen 15
3.2 Eine einfache Struktur 15
3.3 Eine ganz einfache Klasse 16
3.4 Die freie Union von C++ 17
3.5 C++ Typenumwandlung 17
3.6 Programmieraufgaben 18

4 Zeiger **19**
4.1 Repetitorium 19
4.2 Konstante Zeiger und Zeiger auf Konstanten 20

4.3 Ein Zeiger auf void 20
4.4 Dynamische Speicherverwaltung 21
4.5 Zeiger auf Funktionen 22
4.6 Programmieraufgaben 23

5 Funktionen 25
5.1 Prototypen 25
5.2 Kompatible Datentypen 26
5.3 Wie funktioniert die Prototypisierung? 26
5.4 Ein bisschen mehr über Prototypen 27
5.4.1 Hat die Protoypisierung Nachteile? 27
5.5 Übergabe per Referenz 27
5.6 Standardparameter 28
5.7 Warum ist die Ausgabe so wirr? 29
5.8 Variable Parameter-Anzahl 30
5.9 Überladen von Funktionsnamen 31
5.10 Programmieraufgaben 32

6 Zugriffsschutz 33
6.1 Wozu das alles? 33
6.2 Kein Zugriffsschutz 33
6.3 Zugriffsschutz 34
6.4 Was sind private Elemente? 35
6.5 Was sind öffentliche Elemente? 35
6.6 Mehr neue Termninologie 37
6.7 Das Senden einer Nachricht 38
6.8 Eine normale Variable verwenden 39
6.9 Ein Programm mit Problemen 39
6.10 Objekte schützen Daten 40
6.11 Hast Du diese Technik schon einmal benutzt? 41
6.12 Welche Nachteile gibt es? 42
6.13 Konstruktoren und Destruktoren 43
6.14 Modulare Gestaltung 44
6.15 inline-Elementfunktionen 44
6.16 Die header-Datei der Klasse 44
6.17 Die Implementation der Klasse 44
6.18 Wir verwenden das Box-Objekt 45
6.19 Ein Versteckspiel 46
6.20 Abstrakte Datentypen 47
6.21 Freundfunktionen 47
6.22 Die Struktur in C++ 48
6.23 Eine sehr praktische Klasse 48
6.24 Programmieraufgaben 49

7 Mehr über Zugriffsschutz 51
7.1 Wozu das ganze? 51
7.2 Ein Array von Objekten 51
7.3 Deklaration und Definition einer Variabe 52
7.4 Eine Zeichenkette innerhalb eines Objektes 53
7.5 Ein Objekt mit einem internen Zeiger 54
7.6 Ein dynamisch beschafftes Objekt 55
7.7 Ein Objekt mit einem Zeiger auf ein anderes Objekt 56
7.8 Noch ein neues Schlüsselwort: this 57
7.9 Eine verbundene Liste von Objekten 57
7.10 Das Schachteln von Objekten 58
7.11 Operatorüberladung 60
7.12 Nachteile des Überladens von Operatoren 61
7.13 Funktionsüberladung in einer Klasse 61
7.14 Getrennte Kompilation 62
7.15 Einige Methoden kommen von selbst 62
7.16 Der Kopierkonstruktor 63
7.17 Der Zuweisungsoperator 64
7.18 Ein praktisches Beispiel 64
7.19 Ein weiteres praktisches Beispiel 65
7.20 Was sollte der nächste Schritt sein? 65
7.21 Programmieraufgaben 66

8 Vererbung 67
8.1 Eine einfache Klasse als Hors d'oeuvre 67
8.2 Die Implementierung useres Vehikels 68
8.3 Wir verwenden die Vehikel Klasse 68
8.4 Unsere erste abgeleitete Klasse 69
8.5 Wie deklarieren wir eine abgeleitete Klasse? 70
8.6 Die Implementierung der Klasse 70
8.7 Eine weitere abgeleitete Klasse 71
8.8 Die Implementierung des Lasters 71
8.9 Wir verwenden alle drei Klassen 71
8.10 Wozu #ifndef VEHIKEL_H? 72
8.11 Unsere erste praktische Vererbung 73
8.12 Programmieraufgaben 74

9 Mehr Vererbungslehre 75
9.1 Neuorganisierte Dateistruktur 75
9.2 Der scope Operator 76
9.3 Was sind geschützte Daten? 76
9.4 Was sind private Daten? 77
9.5 Versteckte Methoden 77
9.6 Wir verwenden die Laster-Klasse 78

9.7 Alle Daten initialisieren 78
9.8 Reihenfolge der Konstruktion 79
9.9 Wie werden die Destruktoren abgearbeitet? 79
9.10 Vererbung, wenn wir Konstruktoren verwenden 80
9.11 Wie sieht es mit der Reihenfolge der Ausführung aus? 81
9.12 Zeiger auf ein Objekt und ein Array von Objekten 81
9.13 Die neue Zeit-Klasse 82
9.14 Programmieraufgaben 82

10 Mehrfachvererbung und Ausblick 83

10.1 Mehrfachvererbung 83
10.2 Simple Mehrfachvererbung 83
10.3 Doppelte Methodennamen 84
10.4 Mehr über doppelte Methodennamen 84
10.5 Wir vergehen uns an einigen Prinzipien des OOP 85
10.6 Doppelte Variablennamen 85
10.7 Praktische Mehrfachvererbung 86
10.8 Reihenfolge der Elementinitialisierung 86
10.9 Wir verwenden die neue Klasse 87
10.10Klassenschablonen 87
10.11Die erste Schablone 88
10.12Eine Klassenschablone 88
10.13Wiederverwertung der Stapelklasse 89
10.14Was soll mein nächster Schritt sein? 89

11 Virtuelle Funktionen 91

11.1 Ein einfaches Programm mit Vererbung 91
11.2 Das Schlüsselwort virtual 92
11.3 Wir verwenden Zeiger auf Objekte 92
11.4 Ein Zeiger und eine virtuelle Funktion 93
11.5 Ein einfacher Zeiger auf die Elternklasse 93
11.6 Die C++ Zeiger Regel 94
11.7 Eine tatsächlich virtuelle Funktion 94
11.8 Ist das wirklich wichtig? 95
11.9 Programmieraufgaben 96

12 Mehr virtuelle Funktionen 97

12.1 Wie beginne ich ein OOP-Projekt? 97
12.2 Die Person Header-Datei 97
12.3 Die Implementierung der Klasse „Person“ 97
12.4 Die Aufseherin Header-Datei 98
12.5 Die Implementierung der drei abgeleiteten Klassen 98
12.6 Das erste Programm 99
12.7 Die Klasse der verbundenen Liste 99

12.8 Die Implementation der verbundenen Liste 100
12.9 Wir verwenden die verbundene Liste 100
12.10 Wozu ist das alles gut [Zawosbrauchides]? 101
12.11 Ein Anwendungsgerüst 101
12.12 Eine rein virtuelle Funktion 102
12.13 Programmieraufgaben 103

13 Abflug – Das Spiel 105
13.1 Spielispieli 105
13.2 Der Spielablauf 105
13.3 Einige spezielle Konstanten 106
13.4 Die erste Klasse – Uhr 106
13.5 Kommandos 107
13.6 Die zweite Klasse - Gegenstaende 107
13.7 Die Klasse der Flüge und Gates – Plan 108
13.8 Die meistbenutzte Klasse – Ort 109
13.9 Die Nachrichten 110
13.10 Das Hauptprogramm 110
13.11 Die Arbeitsmethode 110
13.12 Schlussbemerkungen zu Abflug 111
13.13 Dein Projekt 111

Vorwort

Diese Einführung in C++ ist eine Übersetzung des wunderbaren C++ Language Tutorial von Gordon Dodrill.

Dieses Dokument ist die eine Einführung in die Programmiersprache C++, setzt aber eine gewisse Erfahrung im Programmieren mit C voraus. Die Einführung besteht aus einer Einführung und 12 Kapiteln. Die Kapitel sollten in der vorgegebenen Reihenfolge *durch*(ein- & aus-)gelesen werden, da die Themen in einer logischen Folge vorgestellt werden und aufeinander aufbauen. Es hilft Dir sicherlich, den Quellcode der Programme herunterzuladen, ihn zu kompilieren und jedes Programm auszuführen, wenn es im Text behandelt wird. Die fleißige Leserin wird das Beispielprogramm leicht (oder auch stärker) verändern, es kompilieren und wieder ausführen, um zu sehen, ob sie das im Text Behandelte verstanden hat. Außerdem sammelt sie dabei wertvolle Erfahrung im Umgang mit seinem (Freud!) Compiler.

Um diese Einführung möglichst effizient zu nutzen, empfiehlt es sich, den Text eines oder zweier Kapitel auszudrucken, die Beispielprogramme herunterzuladen und das Material anhand des Quellcodes (der in einen beliebigen Editor geladen werden kann) durchzugehen. Nach erfolgreichem Abschluß eines Kapitels können weitere heruntergeladen werden.

1 Einführung

1.1 Die Wurzeln von C++

Um ein Betriebssystem für die Computer der Serie PDP-11 (die schlussendlich zum UNIX Betriebssystem führten) zu schreiben, wurde von AT&T die Programmiersprache C entwickelt. Das Ziel der Entwickler war eine möglichst effiziente Programmiersprache. Bjarne Stroustrup, auch Mitarbeiter von AT&T, entwickelte C++, um C um die Objektorientierung zu erweitern. Da das Konzept der Objektorientierung zu diesem Zeitpunkt noch neu und alle Implementationen von objektorientierten Programmiersprachen sehr langsam waren, war es oberstes Ziel, die Effizients von C in C++ beizubehalten.

C++ kann als traditionelle prozedurale Programmiersprache mit einigen zusätzlichen Funktionen gesehen werden. C wird um einige Konstrukte für objektorientiertes Programmieren erweitert und um einige Konstrukte, die die allgemeine Syntax verbessern sollen. Jedes gut geschriebene C++ Programm wird sowohl Elemente des objektorientierten Programmierens als auch solche klassischen prozeduralen Programmierens enthalten. C++ ist eine erweiterbare Sprache, da wir neue Typen definieren können, die sich wie die vordefinierten Typen verhalten, die Teil der Programmiersprache selbst sind. C++ ist somit für größere Programmieraufgaben geeignet.

1.2 Wie fange ich an?

Die Programmiersprache C war ursprünglich durch die klassische Publikation von Kernighan und Ritchie, „The C Programming language", definiert, und dies war bis vor wenigen Jahren der Standard für alle C Programmiererinnen. Der ANSI Standard für C wurde schließlich im Dezember 1989 als offizieller Standard für das Programmieren in C festgelegt. Der ANSI-C Standard erweitert die ursprüngliche Sprache der Kernigan und Ritchie-Definition um Vieles und ändert sie in einigen wenigen Punkten ab. Die beiden Definitionen sind nicht zu hundert Prozent kompatibel und einige erfahrene C Programmiererinnen haben die neueren Konstrukte des ANSI-C Standards möglicherweise noch nicht genau studiert.

Diese Einführung setzt ein solides Grundwissen der Programmiersprache C voraus und wir werden nur sehr wenig Zeit mit den Grundlagen verbringen. Um denjenigen, die den C-Dialekt nach Kernigan und Ritchie gelernt haben, ein wenig unter die Arme zu greifen, werden sich einige Abschnitte den Neuerungen des ANSI Standards widmen. Während der ANSI-C Standard entwickelt wurde, übernahm man viele der neuen Konstrukte von C++ in die Definition von C selbst. Zwar ist also C++ eine Erweiterung von C, es ist aber nur fair zu sagen, daß ANSI-C einige seiner Wurzeln in C++ hat. Ein Beispiel

dafür wären etwa die Prototypen, die für C++ entwickelt und später zu C hinzugefügt wurden.

Der beste und einfachste Weg, C++ zu erlernen, ist, es zu verwenden. So gut wie jedes korrekte C Programm ist auch in C++ korrekt und die ungefähr 12 neuen Schlüsselwörter sind der einzige Grund, warum einige C Programme nicht als C++ Programme compiliert und ausgeführt werden können. Es gibt einige weitere Unterschiede, deren Diskussion wir uns aber für später aufheben. Somit ist der beste Weg, C++ zu erlernen, das vorhandene Wissen (C) zu erweitern und neue Konstrukte zu verwenden, wenn sie in einem Programm gerade nützlich sind. Es wäre ein großer Fehler, zu versuchen, alle neuen Konstrukte und Konzepte in Deinem ersten C++ Programm anzuwenden. Am Ende hättest Du möglicherweise einen unverständlichen Mischmasch an Quellcode, ineffizienter als dasselbe Programm in C alleine. Es wäre viel besser, nach und nach neue Konstrukte in Deinen Talon aufzunehmen und sie anzuwenden, wenn sie gebraucht werden während Du Erfahrung im Umgang mit ihnen sammelst.

Als Illustration der Portabilität von C nach C++ kann wohl der Hinweis dienen, daß alle Beispielprogramme des Coronado Enterprises C Tutorial einwandfrei als C++ Programme kompiliert und ausgeführt werden konnten. Dabei handelte es sich um Version 2.6, die 1994 erschien. Durch die Neuerungen und Änderungen der C++ Compiler seit damals mag dies heute nicht mehr stimmen. Keines der C++ Programme wird sich allerdings als C Programm kompilieren lassen, aus dem ganz einfachen Grund, daß der neue C++ Kommentarstil verwendet wird (da die meisten neueren C Compiler auch C++ Compiler sind, tolerieren sie aber in vielen Fällen die Verwendung dieser Kommentare und geben schlimmstenfalls eine Warnung aus).

1.3 Wie verwende ich diese Einführung?

Wenn Du diese Einführung verwendest, sitzt Du idealerweise vor Deinem Computer. Es soll Dir helfen, neben der richtigen Verwendung der Programmiersprache C++ auch die Verwendung Deines Compilers zu erlernen. Lade ein Beispielprogramm in Deinen bevorzugten Text Editor und lies den dazugehörigen Text, der alle neuen Konstrukte, die im Beispielprogramm vorkommen, erläutert. Wenn Du das Programm und die neu eingeführten Konstrukte verstanden hast, kompiliere das Programm mit Deinem C++ Compiler.

Hast Du das Programm erfolgreich kompiliert und ausgeführt, baust Du einige Fehler ein und achtest darauf, welche Fehlermeldungen der Compiler ausgibt. Wenn Du schon einige Erfahrung im Programmieren hast, wird es Dich nicht erstaunen, Fehlermeldungen zu erhalten, die anscheinend überhaupt nichts mit dem Fehler, den Du eingefügt hast, zu tun haben. Das ist darauf zurückzuführen, daß die Fehleranalyse für jede moderne Programmiersprache ein sehr schwieriges Problem darstellt. Die wertvollste Erfahrung, die Du dabei sammeln sollst, ist diejenige im Umgang mit Deinem Compiler und seinen Eigenheiten. Dann solltest Du versuchen, das Programm mit Hilfe der neu vorgestellten Techniken zu erweitern, um weitere Erfahrungen zu sammeln.

Es wird nicht notwendig sein, jedes einzelne Beispielprogramm zu kompilieren und

auszuführen. Am Ende jedes Programmes findest Du in Kommentaren das Resultat, wenn das kompilierte Programm ausgeführt wird. Einige Konstrukte sind sehr einfach und leicht zu verstehen, sodaß Du auf die Schritte der Kompilation und Programmausführung verzichten wirst und Dich an die Kommentare hältst.

Im Text dieser Einführung werden **Schlüsselwörter**, *Variablennamen* und *Funktionsnamen* gesondert hervorgehoben, um das Studium der Beispielprogramme zu erleichtern.

1.4 Programmieraufgaben

Am Ende jedes Kapitels finden sich Programmieraufgaben, die es Dir ermöglichen sollen, einige der neuen Konstrukte des Kapitels auszuprobieren. Du wirst sicherlich sehr davon profitieren, wenn Du versuchen, die Aufgaben zu lösen. Nur durch das Lesen dieser Einführung wirst Du zwar ein recht ansehnliches theoretisches Wissen um die Programmiersprache C++ erworben haben, zur C++ Programmiererin wirst Du aber nur durch Schreiben von C++ Programmen. Die Programmieraufgaben sollen als Anregungen für eigene Programme dienen.

2 Aller Anfang ist leicht

Am Beginn unseres Studiums der Programmiersprache C++ und objektorientierten Programmierens (OOP) sollen einige Anmerkungen stehen, um Dir den Start zu erleichtern. Da objektorientiertes Programmieren wahrscheinlich neu für Dich ist, wirst Du Dich mit einem Schwall neuer Ausdrücke und neuer Terminologie konfrontiert sehen. Das ist bei etwas Neuem immer der Fall, also laß Dich von all den neuen Dingen nicht einschüchtern!. In jedem Kapitel werden wir einige neue Konzepte einführen und so wirst Du langsam die gesamte Sprache lernen.

Die Kapitel eins bis vier dieser Einführung konzentrieren sich auf die Neuerungen von C++, die nicht objektorientiert sind. Objektorientierte Programmiertechniken werden erst ab Kapitel fünf abgehandelt.

2.1 Sogar die Kommentare sind in C++ verbessert

Beispielprogramm: KOMM.CPP

Die Datei KOMM.CPP dient als Beispiel einiger Neuerungen in C++. Wir werden die einzelnen neuen Konzepte separat, beginnend mit den Kommentaren, besprechen.

Ein C++ Kommentar beginnt mit dem Doppelschrägstrich „//“ irgendwo in einer Zeile und kommentiert den Rest der Zeile aus. Das Ende der Zeile ist auch das Ende des Kommentars. Die alte Methode des Kommentierens nach ANSI-C Standard kann in C++ auch verwendet werden, wie unter anderem aus den Zeilen 11 bis 14 ersichtlich ist. Die neue Definition des Kommentars ist die bevorzugte Methode, da es nunmehr unmöglich ist, unabsichtlich mehrere Zeilen Code auszukommentieren. Das kann passieren, wenn man das Ende des Kommentarblocks („*/“) beim alten C Kommentar vergißt. Guter Programmierstil ist es daher, die neue Art für alle Kommentare zu verwenden und die alte Methode nur zu verwenden, um bei der Fehlersuche eine größere Codesequenz auszukommentieren. Die beiden Arten des Kommentars können nebeneinander verwendet und auch ineinander verschachtelt verwendet werden.

Ich möchte Dich jedoch davor warnen, Kommentare zu verwenden, wenn dasselbe auch damit erreicht werden kann, sinnvolle Namen für Variablen, Konstanten und Funktionen zu finden. Durch die sinnvolle und sinngebende Auswahl von Variablen- und Funktionsnamen wird fast jeder Code selbsterklärend. Du solltest versuchen, das zu erreichen.

2.2 Die Schlüsselwörter const und volatile

In den Zeilen 9 bis 11 werden zwei neue Schlüsselwörter verwendet, die nicht zum ursprünglichen Sprachumfang nach der K&R Definition gehörten, aber Teil des ANSI-C

Standards sind. Das Schlüsselwort **const** wird verwendet, um eine Konstante zu definieren. In Zeile 9 ist diese Konstante vom Typ **int**, ihr Name ist START und sie wird mit dem Wert 3 initialisiert. Der Compiler wird es Dir nicht gestatten, den Wert von START absichtlich oder unabsichtlich zu ändern, weil START als Konstante deklariert wurde. Gäbe es in Deinem Programm eine Variable STARTS, wäre es Dir nicht möglich, STARTS irrtümlich als START zu schreiben und zu verändern. Der Compiler würde einen Fehler ausgeben, aufgrunddessen Du den Fehler beheben könntest. Da es nicht erlaubt ist, den Wert einer Konstanten nachträglich zu ändern, ist es unbedingt notwendig, daß die Konstante mit einem sinnvollen Wert initialisiert wird.

Du hast sicher bemerkt, daß das Schlüsselwort **const** auch im Funktionskopf in Zeile 23 verwendet wird, um anzuzeigen, daß der formale Parameter *Wert* innerhalb der Funktion eine Konstante ist. Jeder Versuch, dieser Variable einen neuen Wert zuzuweisen führt zwangsläufig zu einem Compilationsfehler. Dies ist ein kleiner Schritt, um die Fähigkeit des Compilers, für Dich Fehler zu finden, zu verbessern.

Das Schlüsselwort **volatile** ist Teil des ANSI-C Standards, aber in der ursprünglichen K&R Definition nicht enthalten. Der Wert einer **volatile** Variable kann zwar durch die Programmiererin verändert werden, die Änderung kann allerdings auch von einer anderen Quelle ausgehen, wie zum Beispiel durch einen Interrupt Zeitgeber, der den Wert der Variable hochzählt. Um den Code optimieren zu können, muß dem Compiler bekannt sein, daß die Variable durch derartige andere Einflüsse verändert werden kann. Das Studium von Codeoptimierungsverfahren ist sehr interessant, führt aber weit über den Rahmen dieser Einführung hinaus. Beachte, daß auch eine Konstante Schlvolatile sein kann, was darauf hinausläuft, daß Du die Variable nicht ändern kannst, sehr wohl aber eine Hardware-Funktion.

Der Ausgabebefehl in Zeile 25 soll uns im Moment noch nicht interessieren. Ihm werden wir uns später in einiger Ausführlichkeit widmen. Wenn Du einige Erfahrung mit dem K&R Standard der Programmierung hast, wirst Du Dich möglicherweise an den Zeilen 5 und 23 etwas stoßen. Hier werden die Verwendung von Prototypen und die neuere Funktionsdefinition nach dem ANSI-C Standard illustriert. Die Verwendung von Prototypen ist in C nicht verbindlich vorgeschrieben (aber empfohlen), in C++ aber absolut unumgänglich. Aus diesem Grund widmet sich Kapitel 5 ausschließlich diesem Thema.

Es ist empfehlenswert, dieses Programm jetzt mit Deinem C++ Compiler zu kompilieren und laufen zu lassen, um zu sehen, ob Du dieselben Ergebnisse erhältst wie jene, die am Ende des Programmes in den Kommentaren stehen. Einer der Hauptgründe für das Kompilieren ist, festzustellen, ob Dein Compiler richtig geladen wird und läuft.

2.3 Der Gültigkeitsbereichsoperator

Beispielprogramm: GLTOP.CPP

Das Beispielprogramm GLTOP.CPP illustriert ein weiteres Element, das mit C++ neu eingeführt wird. Es gibt nichts Vergleichbares in K&R oder ANSI-C. Dieser Operator erlaubt den Zugriff auf die globale Variable mit dem Namen *index*, obwohl es eine lokale

Variable mit demselben Namen innerhalb der Funktion **main()** gibt. Die Verwendung des zweifachen Doppelpunktes vor dem Variablennamen in den Zeilen 11, 13 und 16 teilt dem System mit, daß wir die globale Variable mit dem Namen *index*, die in Zeile 4 definiert wurde, verwenden wollen und nicht die lokale Variable, die wir in Zeile 8 definiert haben.

Diese Technik erlaubt überall im Programm den Zugriff auf globale Variablen. Sie können in Berechnungen, als Parameter einer Funktion oder für einen anderen Zweck verwendet werden. Es zeugt aber nicht von gutem Programmierstil, diese Möglichkeit des Zugriffs auf sonst versteckte Variablen zu mißbrauchen, da der Code dadurch oft schwer verständlich wird. Es ist empfehlenswert, in so einem Fall einen anderen Variablennamen zu wählen, aber die Möglichkeit existiert für den Fall, daß Du sie einmal benötigen solltest.

Bevor Du zum nächsten Beispiel weitergehst solltest Du dieses Programm kompilieren und laufen lassen. Unser nächstes Beispielprogramm widmet sich dem **cout** Operator in den Zeilen 10, 11, 15 und 16.

2.4 Die iostream Bibliothek

Beispielprogramm: NACHR.CPP

Wirf einen Blick auf das Beispielprogramm NACHR.CPP als einem ersten Beispiel objektorientierten Programmierens, einem sehr einfachen allerdings. In diesem Programm definieren wir einige Variablen und weisen ihnen Werte zu. In den Ausgaberoutinen in den Zeilen 17 bis 20 und 23 bis 26 finden diese Variablen dann Verwendung. **cout** wird verwendet, um Daten mittels der Standardausgabe (üblicherweise der Bildschirm) auszugeben, aber es bestehen gewisse Unterschiede zur altbekannten *printf()* Funktion, da wir dem System nicht mitteilen müssen, welchen Typs die auszugebenden Daten sind. Beachte bitte, daß **cout** nicht direkt eine Ausgabefunktion ist. Wir können es aber als solche betrachten, bis wir das später in dieser Einführung genauer definieren. Bis dahin soll es uns genügen, festzuhalten, daß **cout** ein Objekt ist, das wir zur Ausgabe von Daten auf dem Bildschirm verwenden.

In C++ sind wie auch in C keine Ausgabefunktionen als Bestandteil der Sprache selbst enthalten, sondern es wird die stream Bibliothek definiert, um auf elegante Weise Ein- und Ausgabefunktionen hinzuzufügen. Die stream Bibliothek wird in Zeile 2 in unser Programm eingebunden.

Der Operator << bewirkt, daß die Variable oder Konstante, die direkt auf den Operator folgt, ausgegeben wird, überläßt es aber dem System zu entscheiden, wie dies geschehen soll. In Zeile 17 weisen wir das System zuerst an, die Zeichenkette auszugeben. Dies passiert, indem die einzelnen Buchstaben auf den Bildschirm kopiert werden. Danach soll das System den Wert von *index* ausgeben. Beachte aber, daß wir dem System keinerlei Angaben darüber machen, welchen Typs *index* ist oder wie der Wert ausgegeben werden soll. Daher liegt es am System, festzustellen, was für eine Variable bzw. Konstante *index* ist und den Wert dementsprechend auszugeben. Nachdem das System den Typ herausgefunden hat, überlassen wir es ihm auch, den Standardwert dafür zu

verwenden, wieviele Zeichen ausgegeben werden sollen. In diesem Fall wählt das System genau so viele Zeichen wie für die Ausgabe notwendig sind ohne etwaige voran- oder nachgestellte Leerzeichen, also genau, was wir für diese Ausgabe wollen. Schlußendlich wird noch der Zeilensprung als eine Zeichenkette mit nur einem Zeichen ausgegeben, und die Zeile mit einem Strichpunkt abgeschlossen.

Beim Aufruf der **cout** Ausgabefunktion in Zeile 17 haben wir eigentlich zwei verschiedene Funktionen aufgerufen, da wir sie sowohl für die zwei Zeichenketten als auch für die Variable vom Typ **int** verwendet haben. Dies ist der erste Hinweis auf objektorientiertes Programmieren, wir fordern vom System einfach, einen Wert auszugeben und überlassen dem System die Suche nach der passenden Funktion. Es ist nicht notwendig, dem System genau mitzuteilen, wie die Daten auszugeben sind, sondern nur, daß sie auszugeben sind. Das ist ein schwaches Beispiel für objektorientiertes Programmieren und wir werden das an anderer Stelle in dieser Einführung mit wesentlich mehr Tiefgang behandeln.

In Zeile 18 lassen wir das System eine andere Zeichenkette ausgeben, gefolgt von einer Fließkommazahl und einer weiteren Zeichenkette, die aus einem Zeichen besteht, nämlich dem Zeilenvorschub. In diesem Fall haben wir das System angewiesen, eine Fließkommazahl auszugeben ohne zusätzliche Angaben über den Variablentyp zu machen und lassen das System wiederum über die richtige Ausgabefunktion, die sich am Veriablentyp orientiert, entscheiden. Mit diesem Prozeß haben wir ein wenig an Kontrolle verloren, da wir nicht entscheiden konnten, wie viele Zeichen vor oder nach dem Komma auszugeben sind. Wir haben uns dafür entschieden, das System entscheiden zu lassen, wie die Ausgabe erfolgen soll.

Die Variable mit dem Namen *Buchstabe* hat den Variablentyp **char** und ihr wird in Zeile 14 der Wert X zugewiesen. In Zeile 19 wird sie dann als Buchstabe ausgegeben, da das **cout** Objekt weiß, daß es sich um einen Buchstaben handelt und die Variable entsprechend ausgibt.

Da C++ weitere Operatoren und Funktionen für Zeichenketten bereithält, sind die Ausgabefunktionen für Zeichenketten sehr flexibel. Bitte schlag in der Dokumentation Deines Compilers die Details weiterer Formatierungsmöglichkeiten nach. Die **cout** und *printf()* Funktionen können nebeneinander verwendet werden, was höchstwahrscheinlich auch Teil des ANSI-C++ Standard sein wird. Allerdings halten sich noch nicht alle Compiler an diesen Standard und manche verwenden verschiedene Formen der Zwischenspeicherung für diese zwei Möglichkeiten der Ausgabe. Das führt unter Umständen zu „wirren“ Ausgaben, aber Du kannst dies einfach dadurch korrigieren, daß Du Dich für eine der beiden Formen, entweder **cout** oder *printf()* entscheidest.

2.5 Mehr über die stream Bibliothek

Die stream Bibliothek wurde für die Verwendung mit C++ definiert, um die Effizienz der Sprache zu steigern. Die *printf()* Funktion wurde schon früh in der Geschichte von C entwickelt und soll alle Möglichkeiten für alle Programmiererinnen bieten. So wurde sie eine umfangreiche Funktion mit viel zusätzlichem Ballast, der zum größten Teil nur von sehr wenigen Programmiererinnen genutzt wird. Die Definition der schlanken stream

Bibliothek erlaubt es der Programmiererin, auf alle Möglichkeiten der Formatierung zuzugreifen, aber nur das zu laden, was gerade benötigt wird. Wir gehen hier zwar nicht auf alle Einzelheiten ein, die C++ stream Bibliothek bietet aber eine große Bandbreite an Formatierungsfunktionen. In der Dokumentation zu Deinem Compiler sollte sich eine komplette Liste aller Möglichkeiten finden.

In den Zeilen 23 bis 26 werden einige der zusätzlichen Funktionen der stream Bibliothek illustriert, die Du verwenden kannst, um Daten sehr flexibel und doch kontrolliert darzustellen. Der Wert von index wird in den Zeilen 23 bis 25 in dezimaler, oktaler und hexadezimaler Notation ausgegeben. Sobald einer der speziellen Operatoren für Zeichenketten, **dec**, **oct** oder **hex** ausgegeben wird, erfolgt jede weitere Ausgabe in dieser Notation. So wird zum Beispiel in Zeile 30 der Wert von index in hexadezimaler Notation ausgegeben, da in Zeile 25 diese Form gewählt wurde. Wird keiner der speziellen Operatoren ausgegeben, ist das Standardformat dezimal.

2.6 Der cin Operator

Neben dem durch die Bibliothek vordefinierten Datenstrom **cout** gibt es **cin**, welcher zum Einlesen von Daten über das Standardeingabegerät (im Regelfall die Tastatur) verwendet wird. Der cin stream verwendet den Operator >>. Ein Großteil der Flexibilität des Datenstromes **cout** findet sich bei **cin** wieder. Ein kurzes Beispiel für die Anwendung von **cin** siehst Du in den Zeilen 28 bis 30. Die speziellen Operatoren für Datenströme, **dec**, **oct** und **hex** legen gleichfalls die Notation für **cin** fest, unabhängig von **cout**. Der Standardwert ist wieder dezimal.

Zusätzlich zu den Datenstromobjekten **cout** und **cin** gibt es noch einen Standarddatenstrom, **cerr**. Über **cerr** werden standardmäßig Fehler ausgegeben. Diese Ausgabe kann im Gegensatz zu **cout** nicht in eine Datei umgeleitet werden. Die drei Datenstromobjekte, **cout**, **cin** und **cerr** entsprechen den Zeigern auf Datenströme **stdout**, **stdin** und **stderr** in C. Beispiele für ihre Verwendung werden uns durch die gesamte Einführung begleiten.

Die stream Bibliothek beinhaltet auch Ein- und Ausgabe in und von Dateien. Dies werden wir kurz im nächsten Beispiel vorstellen.

Kompiliere dieses Programm und führe es aus bevor Du zum nächsten weitergehst. Denke daran, daß das System die Eingabe einer ganzen Zahl verlangt, die wieder am Bildschirm ausgegeben wird, allerding in hexadezimaler Notation.

2.7 Ein- und Ausgabe mit Dateien

Beispielprogramm: DSTROM.CPP

Das Programm DSTROM.CPP ist ein Beispiel für die Verwendung von Datenströmen mit Dateien. Wir werden in diesem Programm einige C++ Objekte verwenden, die wir noch nicht kennen, Du wirst sie also jetzt noch nicht vollkommen verstehen.

In diesem Programm öffnen wir eine Datei zum Lesen, eine andere zum Schreiben und ein dritter Datenstrom wird zu einem Drucker geöffnet, um das Hantieren mit Dateien

zu demonstrieren. In C haben Dateien sowohl zur Eingabe als auch zur Ausgabe den Typ FILE, allerdings wird **ifstream** für Einlesen aus Dateien verwendet, **ofstream** zum Schreiben. Das wird in den Zeilen 8 bis 10 dieses Beispielprogrammes demonstriert. Eigentlich ist **ifstream** eine C++ Klasse und *EingDatei* ein Objekt dieser Klasse, wie wir später sehen werden.

Der einzige Unterschied zwischen den Datenströmen des letzten Programmes und jenen dieses Beispielprogrammes besteht darin, daß die Datenströme im vorigen Programm schon vom System für uns geöffnet wurden. Du wirst sicherlich bemerken, daß der Datenstrom *Drucker* genau in derselben Weise verwendet wird wie **cout** im letzten Programm. Natürlich schließen wir alle Dateien, die wir geöffnet haben, wieder. Schließlich wollen wir guten Programmierstil praktizieren.

In diesem Programm verwenden wir Objekte, also ist es das erste Beispiel für objektorientiertes Programmieren. Das Objekt mit dem Namen *EingDatei* weisen wir in Zeile 17 an, sich selbst zu öffnen. In Zeile 41 soll es dann ein Zeichen nach dem anderen einlesen, in Zeile 48 steht die Anweisung, sich selbst wieder zu schließen. Die „Punkt"-Notation wird für Objekte ähnlich verwendet wie für Strukturen in C. Man gibt den Namen des Objektes an, gefolgt von einem „Punkt" und schließlich von der Aktion, die das Objekt ausführen soll. Mit den Objekten *AusgDatei* und *Drucker* wird in genau derselben Weise verfahren.

Bjarne Stroustrups Buch, das in der Einleitung zu diesem Tutorial aufgelistet ist [nicht wirklich], bietet mehr und tiefergreifende Informationen über die stream Bibliothek, ebenso wie die Dokumentation zu Deinem Compiler. Mach Dir aber jetzt noch nicht zu viele Gedanken darüber. Wenn Du nämlich die Terminologie der Sprache C++ im Laufe dieser Einführung einemal gelernt hast, kannst Du Dich einem eingehenderen Studium der stream Bibliothek widmen, von dem Du dann sicherlich sehr profitierst.

Kompiliere dieses Programm und führe es aus. Es wird eine Datei verlangen, die es kopieren soll. Du kannst den Namen jeder ASCII Datei angeben, die sich in demselben Verzeichnis befindet wie das Programm.

2.8 Variablendefinition

Beispielprogramm: VARDEF.CPP

Das Beispielprogramm VARDEF.CPP ist ein Beispiel für einige weitere Neuerungen von C++, die helfen, klare und einfach zu verstehende Programme zu schreiben. In C++ werden globale und statische Variablen ebenso wie in ANSI-C automatisch mit 0 initialisiert. Die Variablen *index* in Zeile 4 und *Goofy* in Zeile 26 werden also automatisch mit 0 initialisiert. Natürlich kannst Du beide mit irgendeinem anderen Wert initialisieren, wenn Du willst.

Variablen, die innerhalb einer Funktion deklariert werden, werden nicht automatisch mit 0 initialisiert, sondern nehmen den Wert an, der sich gerade an dem Ort im Speicher befindet, wo die Variable definiert wird. Dieser Wert ist nichts wert. Demnach hat die Variable *Etwas* in Zeile 8 keinen gültigen, sondern irgendeinen Wert, der nicht für etwas Sinnvolles verwendet werden sollte (bzw. kann?). In Zeile 11 wird ihr ein Wert gemäß dem

initialisierten Wert von *index* zugewiesen. Dieser Wert wird zur Kontrolle am Bildschirm ausgegeben.

2.9 Die C++ Referenz

Bitte beachte das „&“ in Zeile 9. Es definiert *EtwasAnderes* als Referenz, eine Neuerung von C++. In diesem Zusammenhang sollte eine Referenz allerdings nur sehr selten, wenn überhaupt, verwendet werden. Das ist ein sehr einfaches Beispiel für eine Referenz und soll ihre Verwendung verdeutlichen. Es gibt nichts Vergleichbares in C, da sich eine Referenz wie ein sich selbst dereferenzierender Zeiger verhält. Nach ihrer Initialisierung wird die Referenz zu einem Synonym für die Variable *Etwas*. Eine Änderung des Wertes von *Etwas* ändert notwendigerweise auch den Wert von *EtwasAnderes*, da beide sich auf dieselbe Variable beziehen. Das Synonym kann im Rahmen der C++ Programmiersprache verwendet werden, um auf den Wert der Variablen zuzugreifen. Eine Referenz muß zum Zeitpunkt der Deklaration auf eine Variable initialisiert werden oder der Compiler gibt eine Fehlermeldung aus. Auf welche Variable eine Referenz verweist kann später im Programm nicht mehr geändert werden. Die Referenz ist etwas schwierig zu behandeln, da wir sie Referenzvariable nennen wollen, sie aber keine Variable ist, da sie ja nicht geändert werden kann. Ob uns der Name nun gefällt oder nicht, die Referenz heißt einfach so.

Wird eine Referenz so verwendet wie in diesem Beispiel, kann dies zu sehr unübersichtlichem und verwirrendem Code führen, aber sie kann – anders verwendet – den Code auch sehr klar und einfach machen. Wir werden uns der richtigen Verwendung von Referenzen in Kapitel 5 dieser Einführung widmen.

2.10 Definitionen sind ausführbar

Mit einem Grundwissen in C wird Dir Zeile 16 sehr eigenartig anmuten, sie ist aber in C++ erlaubt. Wo immer eine ausführbare Anweisung erlaubt ist, ist es auch erlaubt, eine neue Variable zu deklarieren, da die Deklaration in C++ als ausführbare Anweisung deklariert ist. Im vorliegenden Fall definieren wir die neue Variable *WiederAnderes* und initialisieren sie mit dem Wert 13. Die Lebensdauer dieser Variable erstreckt sich vom Punkt ihrer Definition bis zum Ende des Blocks in dem sie definiert wurde, im Falle von *WiederAnderes* also bis zum Ende des Programmes **main()**. Die Variable *Goofy* wird noch später in Zeile 27 definiert.

Es ist wesentlich, eine Variable nahe dem Punkt ihrer Verwendung zu deklarieren. Damit kann man einfach ersehen, wofür eine Variable genutzt wird, da der Bereich, wo sie verwendet werden kann, wesentlich enger eingeschränkt ist. Bei der Fehlersuche ist es sehr hilfreich, wenn die Deklaration der Variablen und der zu überprüfende Code nahe beieinander liegen.

2.11 Defintion und Deklaration

Definition und Deklaration sind zwei verschiedene Dinge in C++, wie übrigens auch in ANSI-C. Wir wollen nun eindeutig festlegen, was diese beiden Begriffe in C++ bedeuten. Eine Deklaration informiert den Compiler über die Eigenschaften etwa von einem Typen oder einer Funktion, definiert aber keinen Code, der im ausführbaren Programm verwendet wird. Eine Definition andererseits, definiert etwas, das im ausführbaren Programm tatsächlich existiert, etwa eine Variable oder Code-Zeilen. Jede Einheit im Programm muß genau eine Definition haben. Kurz, eine Deklaration führt einen Namen im Programm ein und eine Definition Code und dieser braucht Platz im Speicher.

Wenn wir ein **struct** deklarieren, deklarieren wir im Grunde nur ein Muster, nach dem der Compiler später im Programm, wenn wir dann Variablen dieses Typs deklarieren, diese speichern soll. Definieren wir aber Variablen dieses Typs, deklarieren wir die Namen dieser Variablen und definieren einen Platz im Speicher für ihre Werte. Deshalb führen wir beim Definieren einer Variablen eigentlich zwei Vorgänge aus: wir deklarieren sie und definieren sie gleichzeitig.

In C sind mehrere Definitionen einer Variable erlaubt, solange die Definition dieselben sind und es sich um ein und dieselbe Variable handelt. C++ verbietet eine solche Vorgangsweise aus einem guten Grund, wie wir später sehen werden.

Wir werden uns im Laufe dieser Einführung noch sehr oft auf diese Definitionen beziehen und auf sie eingehen, sollte also jetzt noch etwas unklar sein, wird das später klarer werden.

2.12 Eine bessere for-Schleife

Schau Dir die **for**-Schleife, die wir in Zeile 20 definieren, genau an. Diese Form der **for**-Schleife ist etwas klarer als jene von ANSI-C, da der Schleifenindex erst in der **for**-Schleife selbst definiert wird. Die Lebensspanne des Index hängt davon ab, wie neu (oder alt) Dein Compiler ist.

Hast Du einen etwas älteren Compiler, reicht die Lebensspanne des Index von der Definition bis zum Ende des umschließenden Blocks. In diesem Fall reicht sie dann bis Zeile 32, da die schließende geschwungene Klammer in Zeile 32 zu der letzten geöffneten geschwungenen Klammer vor der Definition der Variablen gehört. Da die Variable noch verfügbar ist, kann sie für eine weiter Schleife oder jeden anderen Zweck, für den eine Variable vom Typ **int** eingesetzt werden kann, verwendet werden.

Ist Dein Compiler aber relativ neu, endet die Lebensspanne des Index mit dem Ende der Schleife. In diesem Fall reicht sie also bis Zeile 25, wo die Schleife endet. Dies wurde erst vor kurzem vom Komitee, das die Standards festsetzt, eingeführt und ist nur bei den neuesten Compilern umgesetzt.

Egal wie alt Dein Compiler ist, die Variable mit dem Name *Zaehler2* wird bei jedem Schleifendurchlauf definiert und initialisiert, da sie innerhalb des von der **for**-Schleife kontrollierten Blocks definiert wird. Ihre Lebensspanne erstreckt sich nur auf diese Schleife, das heißt die Zeilen 23 bis 25, sodaß der von ihr eingenommene Speicherplatz bei jedem

Schleifendurchlauf automatisch freigegeben wird.

Dir wird aufgefallen sein, daß der Variablen *Zaehler2* in Zeile 23 eine Zahl zugewiesen wird, bei der Ausgabe aber ein Buchstabe ausgegeben wird. Dies ist der Fall, weil C++ sehr bedacht darauf ist, den richtigen Ausgabetypen zu verwenden. Hast Du einen sehr alten Compiler, gibt er hier möglicherweise eine Zahl aus.

Schließlich wird in Zeile 27 die statische Variable *Goofy* definiert und automatisch mit 0 initialisiert, wie wir schon zuvor bemerkt haben. Ihre Lebensspanne reicht vom Punkt der Definition bis zum Ende des Blockes, in dem die Initialisierung geschieht, in diesem Fall Zeile 32.

Kompiliere das Programm und führe es aus.

2.13 Die Priorität der Operatoren

Die Priorität der Operatoren ist identisch mit jener in ANSI-C, wir werden sie hier also nicht definieren. Es gibt einen kleinen Unterschied bei einigen Operatoren, wenn sie überladen werden, eine Technik, die wir später in dieser Einführung erlernen werden. Einige Operatoren funktionieren ein wenig anders, wenn sie überladen werden.

Mach Dir keine Gedanken über den vorigen Absatz, all das wird klarer werden, wenn wir uns einigen weiteren Punkten gewidmet haben.

2.14 Programmieraufgaben

1. Schreib ein Programm, das Deinen Namen und Dein Geburtsdatum mit Hilfe des Datenstromobjektes cout drei Mal am Bildschirm ausgibt. Definiere alle Variablen, die Du verwendest möglichst nahe ihrer Verwendung.

2. Schreib ein Programm mit einigen konstanten (const) Werten und von außen veränderbaren (volatile) Variablen und versuche, den Wert der Konstanten zu verändern, um zu sehen, welche Fehlermeldung Dein Compiler ausgibt.

3. Schreib ein Programm, das Datenstromobjekte verwendet, um Dein Geburtsdatum mit drei verschiedenen cin Anweisungen entgegenzunehmen. Gib Dein Geburtsdatum in oktaler, dezimaler und hexadezimaler Notation aus.

3 Zusammengesetzte Datentypen

3.1 Aufzählungstypen

Beispielprogramm: AUFZ.CPP

Die Datei AUFZ.CPP gibt ein Beispiel für ein Programm, das eine Variable eines Aufzählungstypen verwendet. Der Aufzählungstyp wird in C++ ähnlich verwendet wie in ANSI-C, es gibt aber doch beträchtliche Unterschiede. Das Schlüsselwort **enum** muß bei der Definition einer Variablen dieses Typs nicht noch einmal gebraucht werden, man kann dies aber tun. Die Variable mit dem Namen *Spielausgang* wird als ein Aufzählungstyp definiert. Wie Du siehst ist **enum** bei der Definition nicht vonnöten. Es kann aber für Dich eindeutiger sein, das Schlüsselwort auch bei der Variablendefintion zu verwenden, so wie das in C verlangt wird.

Um zu zeigen, daß der Gebrauch von **enum** wirklich optional ist, steht das Schlüsselwort in Zeile 9, in Zeile 8 lassen wir es aber weg. Neben diesem trivialen Unterschied gibt es aber noch einen wichtigeren in der Verwendung von Aufzählungstypen zwischen C und C++. In C ist ein Aufzählungstyp einfach eine Variable vom Typ **int**, in C++ aber ein eigener Typ. Mit Variablen dieses Typs können keine mathematischen Operationen durchgeführt werden, es können ihnen auch keine ganzen Zahlen zugewiesen werden. Eine solche Variable kann auch nicht hinauf- oder heruntergezählt werden wie in C. In unserem Beispielprogramm verwenden wir eine ganze Zahl als Index der **for**-Schleife, da wir diese inkrementieren können. Dann weisen wir der Aufzählungsvariablen den Wert dieses Indizes (*Zaehler*) mithilfe einer Typenumwandlung zu. Die Typenumwandlung ist unbedingt erforderlich, oder der Compiler wird einen Fehler melden. Die mathematischen Operationen sowie die Operatoren zum In- und Dekerementieren können für den Aufzählungstypen definiert werden, sie sind aber nicht automatisch verfügbar. Dem Überladen von Operatoren werden wir uns später widmen, dann wird auch der letzte Satz etwas mehr Sinn machen.

Wenn Du mit einem älteren Compiler arbeitest, kannst Du möglicherweise die Aufzählungsvariable *Spielausgang* als Schleifenindex verwenden, Dein Code ist dann aber nicht mit dem eines neueren Compilers kompatibel.

Den Rest des Programmes verstehst Du sicherlich. Nachdem Du Dir alles genau angeschaut hast, kompiliere das Beispielprogramm und führe es aus.

3.2 Eine einfache Struktur

Beispielprogramm: STRUKT.CPP

Das Programm STRUKT.CPP illustriert die Verwendung einer einfachen Struktur.

Diese unterscheidet sich von der in C üblichen nur dadurch, dass das Schlüsselwort **struct** bei der Definition einer Variable dieses Typs nicht notwendig ist. In den Zeilen 12 und 13 sehen wir die Definition einer Variable ohne das Schlüsselwort, Zeile 14 zeigt, daß wir es aber verwenden können, wenn wir wollen. Es bleibt Dir überlassen, welchem Stil Du in Deinen C++ Programmen den Vorzug gibst.

Kompiliere auch dieses Programm, nachdem Du es genau studiert hast. Das nächste Beispiel ist diesem sehr ähnlich, es stellt aber etwas ganz Neues vor, eine Klasse.

3.3 Eine ganz einfache Klasse

Beispielprogramm: KLASSE1.CPP

Das Programm KLASSE1.CPP ist unser erstes Beispiel für eine Klasse in C++. Das ist das erste, mit Sicherheit aber nicht das letzte Beispiel, da die Klasse der Hauptgrund ist, warum wir C++ ANSI-C oder einer anderen (prozeduralen) Programmiersprache vorziehen. Wir verwenden das Schlüsselwort **class** in Zeile 4 in genau derselben Weise wie **struct** im letzten Beispiel. Diese beiden Konstrukte sind in der Tat auch sehr ähnlich. Es gibt einen wesentlichen Unterschied, einstweilen wollen wir uns aber mit den Ähnlichkeiten beschäftigen.

Tier in Zeile 4 ist der Name der Klasse. Wenn wir Variablen dieses Typs definieren, können wir das Schlüsselwort **class** noch einmal verwenden wie in Zeile 15 oder es auch weglassen wie in den Zeilen 13 bis 15. In unserem letzten Programm haben wir fünf Variablen einer Struktur deklariert, in diesem Programm deklarieren wir aber fünf Objekte. Sie heißen Objekte, da ihr Typ eine Klasse ist. Die Unterschiede sind klein, aber fein, und wir werden später in dieser Einführung sehen, daß das Konzept der Klasse wirklich sehr wichtig und wertvoll ist. Hier haben wir die Klasse nur vorgestellt, um eine kleine Vorschau auf das zu geben, was Dich noch erwartet.

Eine Klasse ist ein Typ, den man verwendet, um Objekte zu definieren gerade so wie man eine Struktur verwendet, um Variablen zu definieren. Dein Hund Rex ist eine spezielle Instanz der umfassenden Klasse der Hunde. Ähnlich verhält es sich mit Objekten und Klassen: ein Objekt ist eine spezielle Instanz einer Klasse. Die Klasse ist ein so generalisiertes Konzept, daß Bibliotheken mit vorgefertigten Klassen erhältlich sind. Du kannst Klassen kaufen, die solche Operationen durchführen wie etwa die Verwaltung von Stapeln (stacks), Warteschlangen (queues) oder Listen, das Sortieren von Daten, das Fenster-Management usw. Das ist durch die allgemeine Anwendbarkeit und Flexibilität der Klasse möglich.

Das Schlüsselwort **public** in Zeile 6 ist notwendig, da die Variablen einer Klasse standardmäßig privat, das heißt von außen nicht zugänglich sind. Indem wir sie mit dem Schlüsselwort **public** öffentlich machen, können wir auf sie zugreifen. Mach Dir über dieses Programm jetzt nicht zu viele Gedanken, wir werden all das später relativ genau behandeln. Kompiliere das Programm und führe es aus, um zu sehen, ob es auch das tut, was wir wollen. Bedenke, daß das Dein erster Kontakt mit einer Klasse überhaupt ist und im Grund gar nichts von dem illustriert, was dieses so mächtige Element von C++ ausmacht.

3.4 Die freie Union von C++

Beispielprogramm: UNION.CPP

Das Beispielprogramm UNION.CPP zeigt eine freie Union. Während in ANSI-C alle Unionen einen Namen haben müssen, ist dies in C++ nicht notwendig. In C++ können wir eine freie Union, eine namenlose Union verwenden. Die Union ist in eine einfache Struktur eingebettet und Du hast sicher bemerkt, daß auf die Deklaration der Union in Zeile 13 kein Variablenname folgt. In ANSI-C hätten wir die Union benennen und dreifache Namen (drei Namen mit Punkten dazwischen) vergeben müssen, um auf die Mitglieder der Union Zugriff zu haben. Da es sich hier aber um eine freie Union handelt gibt es keinen Namen der Union und daher genügt auch ein doppelter Name, um auf die Mitglieder zuzugreifen, wie unter anderem die Zeilen 20, 24 und 28 zeigen.

Du wirst Dich daran erinnern, daß eine Union alle ihre Daten auf derselben Stelle im Speicher ablegt, sodaß effektiv immer nur eine Variable verfügbar ist. Genau das geschieht auch hier. Die Variablen mit den Namen *Tankinhalt*, *Bomben* und *Paletten* werden an derselben Stelle gespeichert und es obliegt der Programmiererin, Buch zu führen, welche der Variablen denn nun gerade dort gespeichert wird. Dem *Transporter* wird in Zeile 28 ein Wert für *Paletten* zugewiesen, in Zeile 30 dann ein Wert für *Tankinhalt*. Wenn wir den Wert für *Tankinhalt* zuweisen, geht der Wert für *Paletten* verloren und ist nicht mehr verfügbar, da er ja dort gespeichert war, wo jetzt *Tankinhalt* abgelegt ist. Unionen werden also in C++ genauso verwendet wie in C, die Benennung der Komponenten ausgenommen.

Den Rest des Programmes verstehst Du sicherlich mit Leichtigkeit. Kompiliere es und führe es aus.

3.5 C++ Typenumwandlung

Beispielprogramm: TYPENUW.CPP

Das Programm TYPENUW.CPP gibt einige Beispiele für die Typenumwandlung in C++. Du kannst die Typenumwandlungen in C++ genauso wie in C machen, C++ eröffnet Dir aber auch noch eine andere Möglichkeit.

In den Zeilen 10 bis 17 verwenden wir die bekannte Form des „cast“ aus ANSI-C, nichts Neues für die erfahrene C Programmiererin. Bei etwas genauerem Hinsehen stellt sich heraus, daß die Zeilen 10 bis 13 alle dasselbe bewirken. Der einzige Unterschied ist, daß wir den Compiler in einigen Anweisungen zwingen, die Typenumwandlung vor der Addition und der Zuweisung zu machen. In Zeile 13 wird die Variable vom Typ **int** vor der Addition in den Typ **float** konvertiert, die resultierende Variable vom Typ **float** wird dann nach **char** konvertiert bevor sie der Variablen c zugewiesen wird.

Weitere Beispiele für Typenumwandlungen findest Du in den Zeilen 15 bis 17. Alle drei Anweisungen sind im Grunde gleich.

Die Beispiele in den Zeilen 19 bis 26 sind C++-spezifisch und in ANSI-C nicht erlaubt. In diesen Zeilen werden die Typenumwandlungen notiert als seien sie Funktionsaufrufe anstatt der bekannteren „cast“ Methode. Die Zeilen 19 bis 26 sind identisch mit den

Zeilen 10 bis 17.

Du wirst diese Methode der Typenumwandlung möglicherweise klarer und einfacher zu verstehen finden als die Methode des „cast“ und in C++ darfst Du auch beide anwenden. Du darfst die beiden Typenumwandlungen auch mischen, der Code kann dadurch aber sehr schwer verständlich werden.

Kompiliere das Programm und führe es aus.

3.6 Programmieraufgaben

1. Vom Programm AUFZ.CPP ausgehend füge dem Aufzählungstyp Spielausgang den Wert STRAFE hinzu. Erweitere das Programm auch um eine Nachricht und die Funktionalität, diese Nachricht auszugeben.

2. Erweitere das Programm KLASSE1.CPP um die Variable Hoehe vom Typ float und speichere beliebige Werte in dieser Variable. Gib einige der Werte aus. Setze die neue Variable dann vor das Schlüsselwort public:. Was für Fehlermeldungen erhältst Du? Wir werden uns diesem Fehler in Kapitel 6 dieser Einführung widmen.

4 Zeiger

Aufgrund der Bedeutung, die Zeigern in C und C++ zukommt, werden wir in diesem Kapitel auf die wichtigsten Fragen eingehen. Selbst wenn Du in der Verwendung von Zeigern absolut firm bist, solltest Du dieses Kapitel nicht links (respektive rechts) liegen lassen, da auch einige Neuerung von C++ hier präsentiert werden.

4.1 Repetitorium

Beispielprogramm: ZEIGER.CPP

Das Programm ZEIGER.CPP ist ein einfaches Beispiel für die Verwendung von Zeigern. Hier wiederholen wir die Grundzüge des Umgangs mit Zeigern, das heißt, wenn Du Dich auf diesem Gebiet sicher fühlst, kannst Du dieses Beispielprogramm ohne weiteres überspringen.

In ANSI-C ebenso wie in C++ wird ein Zeiger deklariert, indem man dem Variablennamen ein Sternchen voransetzt. Der Zeiger zeigt dann auf eine Variable dieses spezifischen Typs und sollte nicht für Variablen anderen Typs verwendet werden. Daher ist *ZgInt* ein Zeiger auf Variablen vom Typ **int** und sollte nur für solche verwendet werden. Natürlich weiß die erfahrene C Programmiererin, daß sie den Zeiger auch für alle anderen Typen einsetzen kann, einfach mit Hilfe einer „cast“ Typenumwandlung, sie ist dann aber auch für die korrekte Verwendung verantwortlich.

In Zeile 12 wird dem Zeiger *ZgInt* die Adresse der Variablen *Schwein* zugewiesen, und in Zeile 13 verwenden wir den Namen des Zeigers *ZgInt*, um den Wert der Variablen *Hund* zum Wert von *Schwein* zu addieren. Das Sternchen dereferenziert den Zeiger genauso wie es auch in C geschieht. Abbildung 4.1 stellt den Speicherinhalt nach der Zeile 13 graphisch dar. Eine Box mit Sternchen ist ein Zeiger. Wir verwenden die Adresse, um in Zeile 14 den Wert der Variable *Schwein* auszugeben und zu zeigen, wie man einen Zeiger mit dem Ausgabestromobjekt **cout** verwendet. Genauso wird dem Zeiger auf

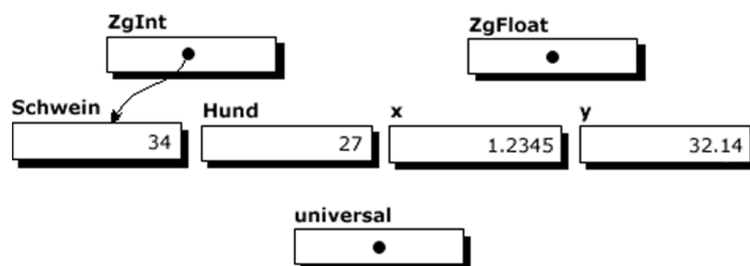

Abbildung 4.1: Speicherinhalt nach Zeile 13

eine Variable vom Typ **float**, *ZgFloat*, die Adresse von x zugewiesen, um dann in einer einfachen Rechnung in Zeile 18 Verwendung zu finden.

4.2 Konstante Zeiger und Zeiger auf Konstanten

Die Definition von C++ erlaubt es, einen Zeiger auf eine Konstante so zu definieren, daß der Wert, auf den der Zeiger zeigt, nicht verändert werden kann, sehr wohl aber die Adresse des Zeigers, sodaß er auf eine andere Variable oder Konstante zeigen kann. Diese Methode, einen Zeiger auf eine Konstante zu definieren, illustriert Zeile 22. Neben Zeigern auf Konstante sind auch konstante Zeiger möglich, solche die nicht verändert werden können. Dies zeigen wir in Zeile 23. Beachte, daß wir diese Zeiger im Beispielcode nicht verwenden.

Diese beiden Methoden können dazu dienen, beim Kompilieren eine zusätzliche Kontrolle durchzuführen und so die Qualität des Code zu heben. Wenn Du sicher bist, daß ein Zeiger immer auf dieselbe Variable oder Konstante zeigen wird, solltest Du ihn als konstanten Zeiger definieren. Bist Du sicher, daß ein Wert nicht verändert wird, definierst Du ihn als Konstante und der Compiler wird Dich darauf aufmerksam machen, wenn Du doch versuchen solltest, den Wert zu ändern.

4.3 Ein Zeiger auf void

Der Zeiger auf **void** ist zwar Teil des ANSI-C Standards, aber doch relativ neu, sodass wir hier kurz darauf eingehen. Einem Zeiger auf **void** kann der Wert jedes anderen Zeigertyps zugewiesen werden. Wie Du siehst, wird dem Zeiger auf **void**, *universal* in Zeile 15 die Adresse einer Variable vom Typ **int** zugewiesen wird, in Zeile 20 aber die Adresse einer Variable vom Typ **float**, ohne „cast“ und ohne Compiler-Fehler. Das ist ein relativ neues Konzept in C und C++. Es gibt der Programmiererin die Möglichkeit, einen Zeiger zu definieren, der auf eine Vielfalt von Dingen zeigen kann um die Informationen innerhalb eines Programmes so richtig zum Laufen bringt. Ein gutes Beispiel wäre etwa die *malloc()* Funktion, die eine Zeiger auf **void** zurückgibt. Dieser Zeiger kann auf nahezu alles zeigen, sodaß man den zurückgegebenen Zeiger auf den richtigen Typ zeigen läßt.

Für einen Zeiger auf **void** wird im Speicher so viel Platz zur Verfügung gestellt, daß er mit allen vordefinierten einfachen Typen, die in C++ oder ANSI-C verfügbar sind, verwendet werden kann. Er kann aber auch mit allen zusammengesetzten Typen, die die Programmiererin definieren kann, verwendet werden, da sich zusammengesetzte Typen aus einfachen zusammensetzen.

Wenn bei diesem trivialen Programm auch nur die kleinste Unklarheit auftritt, solltest Du noch einmal ein gutes C Programmierbuch zur Hand nehmen und die Verwendung von Zeigern nachschlagen, bevor Du in dieser Einführung weitergehst. Im weiteren Verlauf wird nämlich ein profundes Wissen um Zeiger und ihre Verwendung vorausgesetzt. Es ist unmöglich, ein etwas komplexeres Programm zu schreiben, ohne Zeiger zu verwenden.

Kompiliere das Programm und führe es aus.

4.4 Dynamische Speicherverwaltung

Beispielprogramm: NEWDEL.CPP

Das Programm NEWDEL.CPP ist ein erstes Beispiel für die Verwendung der Operatoren **new** und **delete**. Die Operatoren **new** und **delete** bewerkstelligen die dynamische Speicherbelegung und -freigabe in sehr ähnlicher Weise wie *malloc()* und *free()* in C.

Da die dynamische Speicherverwaltung in C sehr oft verwendet wird und dem ergo auch in C++ so sein würde, entschlossen sich die Entwickler von C++, dies als Teil der Sprache selbst zu implementieren, anstatt diese Funktionalität einer Bibliothek zu überlassen. Die Operatoren **new** und **delete** sind also ein Teil der Programmiersprache, genauso wie etwa der Additionsoperator. Deshalb sind diese Operatoren sehr effizient und einfach in der Anwendung wie wir in diesem Beispielprogramm sehen werden.

In den Zeilen 15 und 16 verwenden wir Zeiger wie wir es in C immerzu getan haben, Zeile 17 illustriert die Verwendung des **new** Operators. Dieser Operator verlangt ein Argument, das ein Typ sein muß, wie hier illustriert. Der Zeiger *Zeiger2* zeigt nun auf die Variable vom Typ **int**, für die wir dynamisch Speicherplatz bereitgestellt haben. Die Ganzzahl kann genauso verwendet werden, wie jede dynamisch bereitgestellte Variable in ANSI-C verwendet wird. Die Zeilen 19 und 20 zeigen die Ausgabe des Wertes, der der Variable in Zeile 18 zugewiesen wurde.

In Zeile 21 führen wir eine weitere Variable dynamisch ein und Zeile 22 läßt *Zeiger2* auf dieselbe dynamisch bereitgestellte Variable zeigen wie *Zeiger1*. In diesem Fall geht jede Referenz auf die Variable, auf die Zeiger2 zuerst zeigte, verloren und diese Variable kann nicht wieder verwendet oder freigegeben werden. Sie ist verloren im Speicher bis unser Programm die Kontrolle wieder an das Betriebssystem übergibt und der Speicherplatz neu belegt wird. Dies ist offensichtlich kein guter Programmierstil. Beachte, daß *Zeiger1* in Zeile 26 mit dem Operator **delete** wieder freigegeben wird und *Zeiger2* nun nicht mehr freigegeben werden kann, da er auf nichts zeigt. Da der Zeiger *Zeiger1* selbst nicht verändert wird, zeigt er noch immer auf dieselbe Speicheradresse. Möglicherweise könnten wir diese Adresse noch einmal verwenden, aber das wäre schrecklicher Programmierstil, da wir keine Garantie haben, was das System mit dem Zeiger oder den Daten macht. Die Speicheradresse wird an die Liste der freien Adressen zurückgegeben und wohl bald wieder von einem Programm verwendet werden.

Da der **delete** Operator seiner Definition gemäß nichts tut, wenn ihm der Wert NULL übergeben wird, kannst Du das System zwar anweisen, die Daten, auf die ein Zeiger mit dem Wert NULL zeigt, zu löschen, allein es wird nichts passieren. Das ist also nur unnötiger Code. Der **delete** Operator kann nur solche Daten freigeben, die mit dem **new** Operator angefordert wurden. Wird der **delete** Operator mit irgendeiner anderen Art von Daten verwendet, ist die Operation nicht definiert und es kann eigentlich alles passieren. Nach dem ANSI Standard ist auch ein Systemabsturz ein erlaubtes Ergebnis dieser undefinierten Aktion und kann vom Autor des Compilers als solches definiert werden.

In Zeile 28 deklarieren wir einige Variablen vom Typ **float**. Du erinnerst Dich sicherlich, daß Du in C++ Variablen nicht am Anfang eines Blocks deklarieren mußt. Eine Deklaration ist eine ausführbare Anweisung und kann daher überall in einer Liste von

solchen Anweisungen stehen. Eine der Variablen vom Typ **float** wird bei der Deklaration bereitgestellt, um zu zeigen, daß dies möglich ist. Wir wenden auf diese Variablen einige der Funktionen an, die wir zuerst auch auf die ganzzahligen Variablen verwendet haben.

In den Zeilen 36 bis 44 findest Du einige Beispiele für die Verwendung von Strukturen. Diese sollten eigentlich selbsterklärend sein.

Schließlich wirst Du Dich wundern, wie es möglich ist, einen Block von willkürlicher Größe festzulegen, wo doch der **new** Operator einen Typen als Argument verlangt, um die Größe des Blocks festzulegen. Um dies zu erreichen, verwenden wir das Konstrukt in Zeile 48. Hier belegen wir einen Speicher der Größe von 37 Variablen des Typs **char**, das heißt 37 Bytes. In Zeile 50 belegen wir einen Block, der um 133 Bytes größer ist als eine *Datum* Struktur. Dadurch wird klar, daß der **new** Operator mit all der Flexibilität von *malloc()* verwendet werden kann. Die eckigen Klammern in den Zeilen 49 und 51 sind notwendig, um dem Compiler zu sagen, daß er einen Array freigibt.

Die Standardfunktionen in C für dynamisches Speichermanagment, *malloc()*, *calloc()* und *free()* sind auch in C++ verfügbar und werden genauso verwendet, wie Du es von C kennst. Wenn Du alten Code verwendest, der die alten Funktionen enthält, verwende auch weiterhin diese Funktionen. Schreibst Du allerdings ein neues Programm, solltest Du die neuen Funktionen verwenden, da sie ein Teil der Sprache selbst sind und damit wesentlich effizienter. Es ist ein Fehler, **delete** auf eine Variable, die Du mit *malloc()* bereitgestellt hast, anzuwenden, wie es auch ein Fehler ist, *free()* mit einer Variable zu verwenden, die **new** bereitgestellt hat.

Kompiliere das Programm und führe es aus.

4.5 Zeiger auf Funktionen

Beispielprogramm: FUNKTZG.CPP

Das Programm FUNKTZG.CPP gibt ein Beispiel für die Verwendung eines Zeigers auf eine Funktion. Es muß gesagt werden, daß es sich hier um nichts Neues handelt, Zeiger auf eine Funktion sind in ANSI-C genauso verfügbar wie in C++ und funktionieren für beide Sprachen in derselben, hier beschriebenen Weise. In C Programmen wird dieses Konstrukt allerdings seltener verwendet, weshalb wir es hier erwähnen. Wenn Du in Zeigern auf Funktionen firm bist, kannst Du dieses Beispielprogramm getrost überspringen.

Das einzig Erwähnenswerte an diesem Programm ist der Zeiger auf eine Funktion, den wir in Zeile 7 kreieren. Wir deklarieren einen Zeiger auf eine Funktion, die nichts (**void**) zurückgibt und einen Parameter verlangt, eine Variable vom Typ **float**. Dir wird nicht entgangen sein, daß alle drei Funktionen, die wir in den Zeilen 4 bis 6 deklarieren, diese Vorgaben erfüllen und damit von diesem Zeiger aufgerufen werden können. Wenn Du in C keine Prototypen verwendet hast, werden Dir diese Zeilen etwas eigenartig vorkommen. Laß Dich aber jetzt nicht verwirren, wir werden auf Prototypen im nächsten Kapitel eingehen.

In Zeile 14 rufen wir die Funktion *DruckeEtwas()* mit dem Parameter *Pi* auf. In Zeile 15 weisen wir dem Funktionszeiger *Funktionszeiger* den Wert *DruckeEtwas* zu, um in

Zeile 16 mittels des Funktionszeigers dieselbe Funktion noch einmal aufzurufen. Aufgrund der Zuweisung in Zeile 15 sind die Zeilen 14 und 16 also in ihrem Resultat absolut identisch. Die Zeilen 17 bis 22 zeigen noch einige weitere Beispiele für die Verwendung von Funktionszeigern. Schau Dir diese Beispiele in Ruhe an, ich lasse Dich mit ihnen alleine.

Da wir einem Zeiger auf eine Funktion den Namen einer Funktion zuweisen konnten, ohne einen Zuweisungsfehler zu produzieren, muß der Name einer Funktion ein Zeiger auf genau diese Funktion sein. Exakt das ist auch der Fall. Ein Funktionsname ist nichts Anderes als ein Zeiger auf die Funktion, allerdings ein konstanter Zeiger und damit unveränderbar. Dasselbe ist uns auch beim Studium der Arrays in ANSI-C untergekommen. Der Name eines Array ist ein konstanter Zeiger auf das erste Element des Array.

Da es sich beim Namen einer Funktion um einen Zeiger auf die Funktion handelt, können wir diesen Namen einem Funktionszeiger zuweisen und den Funktionszeiger verwenden, um die Funktion aufzurufen. Die einzige Bedingung ist, daß der Rückgabewert sowie die Zahl und die Art der Parameter übereinstimmen müssen. Die meisten C und C++ Compiler werden Dich nicht warnen, wenn die Parameterlisten bei der Zuweisung nicht übereinstimmen. Das wäre auch gar nicht möglich, weil die Zuweisung zur Laufzeit erfolgt, wenn keine Typeninformationen verfügbar sind.

Kompiliere das Programm und führe es aus.

4.6 Programmieraufgaben

1. Wenn Daten, für die der Speicherplatz dynamisch angefordert wurde, gelöscht werden, sind sie eigentlich noch immer im Speicher vorhanden. Wiederhole die Ausgabeanweisung in Zeile 24 und 25 vom Programm NEWDEL.CPP gleich nach dem **delete** in Zeile 25, um festzustellen, ob die Werte noch immer gespeichert sind. Wiederhole die Ausgabe noch einmal kurz vor Ende des Programmes, wenn die Daten schon überschrieben sein sollten, um zu sehen, was ausgegeben wird. Selbst wenn Du die richtigen Daten bekommst, ist es schrecklicher Programmierstil, sich darauf zu verlassen, daß die Daten nicht überschrieben wurden, was in einem größeren dynamischen Programm sehr wahrscheinlich ist.

2. Schreibe eine neue Funktion für das Programm FUNKTZG.CPP, die als einzigen Parameter eine Variable vom Typ **int** verlangt und versuche, diese Funktion mittels des Funktionszeigers aufzurufen, um zu sehen, ob Du der Funktion die richtigen Daten übergeben kannst.

5 Funktionen

In diesem Kapitel wollen wir uns den Verbesserungen widmen, die C++ bei den Funktionen gebracht hat. Diese Änderungen machen das Programmieren ein wenig einfacher und erlauben es dem Compiler, mehr Fehler zu finden. Ein großer Teil dieses Kapitels wird sich den modernen Funktionsprototypen widmen.

Prototypisierung erlaubt es dem Compiler, zusätzliche Typenkontrollen durchzuführen, was so manchen Programmierfehler aufdeckt. Die ersten beiden Beispiele dieses Kapitels sollen Dir die Verwendung von Prototypen begreiflich machen und ihren Nutzen erklären. Die Prototypisierung ist ein relativ neues Konzept in C, sodaß auch so manche erfahrene C Programmiererin sich darin noch nicht zuhause fühlt. Wenn Du allerdings schon Erfahrung mit Prototypen gesammelt hast, kannst Du gleich zum Abschnitt Übergabe mittels Referenz weitergehen.

5.1 Prototypen

Beispielprogramm: PROTYP1.CPP

Das Beispielprogramm PROTYP1.CPP stellt uns Prototypen einmal vor und illustriert ihre Verwendung. Es besteht kein Unterschied zwischen der Prototypisierung, die C++ verwendet und jener von ANSI-C. Allerdings sind viele C Programmiererinnen etwas mißtrauisch, wenn es um Prototypen geht und anscheinend nicht wirklich willens, sie auch zu verwenden. In C++ ist die Prototypisierung wesentlich wichtiger und wird auch viel öfter verwendet. In manchen Fällen führt in C++ kein Weg an den Prototypen vorbei.

Ein Prototyp ist ein vereinfachtes Modell eines komplexen Objektes. In unserem Beispiel ist eine komplette Funktion das Objekt und den Prototyp dazu finden wir in Zeile 4. Der Prototyp gibt ein Modell der Schnittstelle der Funktion, sodaß Funktionsaufrufe auf die Richtigkeit der Anzahl und der Art der Argumente überprüft werden können. Jeder Aufruf der Funktion *MachWas()* muß genau drei Parameter haben, oder der Compiler wird einen Fehler melden. Neben der Anzahl müssen auch die Typen der Parameter übereinstimmen, oder wir bekommen eine Fehlermeldung. Beachte, daß der Compiler in den Zeilen 12 und 13 schon eine Typenüberprüfung, die auf dem Prototyp in Zeile 4 basiert durchführen kann, obwohl die Funktion selbst noch nicht definiert ist. Wenn kein Prototyp verfügbar wäre, könnte weder die Anzahl noch die Art der Parameter überprüft werden. Solltest Du da einen Fehler machen und keinen Prototypen angegeben haben, wird sich das Programm zwar wunderbar kompilieren und linken lassen, bei der Ausführung können dann aber eigenartige Dinge passieren.

Um einen Prototypen zu schreiben, mußt Du nur den Kopf der Funktion an den Beginn des Programmes kopieren und am Ende einen Strichpunkt anhängen als Indikator,

daß es sich um einen Prototypen und nicht um die Funktion selbst handelt. Die Variablennamen, die im Prototypen angegeben werden, sind nicht notwendig und dienen eigentlich nur als Kommentare, da sie vom Compiler komplett ignoriert werden. Du könntest den Variablennamen *Fluegel* in Zeile 4 mit Deinem Vornamen ersetzen und es würde in der Kompilation keinen Unterschied machen. Die Gedanken dessen, der Dein Programm dann liest, werden sich dann womöglich auch mehr um Dich denn um das Programm drehen (was ja beileibe nichts Schlechtes ist).

In diesem Fall sind die beiden Funktionsaufrufe in den Zeilen 12 und 13 korrekt, sodaß die Kompilation ohne Fehler über die Bühne gehen sollte.

Obwohl wir in unserem Programm den Typ **char** für die Variable *Augen* verwenden wollen, möchten wir ihn doch als Zahl und nicht als Zeichen gebrauchen. Die Typenumwandlung zum Typ **int** in Zeile 22 brauchen wir, um die richtige Ausgabe (die einer Zahl anstatt eines ASCII Zeichens) zu erhalten. Das nächste Beispielprogramm ist ähnlich, aber ohne diese Umwandlung zu **int**.

5.2 Kompatible Datentypen

Wir haben die Kompatibilität von Typen schon erwähnt, wollen aber hier noch einmal kurz darauf eingehen, um die Diskussion der Prototypen zu komplettieren. Kompatibel sind alle einfachen Typen, die untereinander sinnvoll umgewandelt werden können. Wenn Du zum Beispiel eine ganze Zahl als eigentlichen Parameter verwendest, die Funktion allerdings eine Variable vom Typ **float** als formellen Parameter verlangt, führt das System die Umwandlung automatisch durch, ohne daß Du etwas davon bemerkst. Dasselbe gilt auch für **float** nach **char** und **char** zu **int**. Es gibt strikte Regeln für die Typenumwandlung, die Du zum Beispiel in Kapitel 3.2 des ANSI-C Standards findest.

Wenn wir der Funktion als eigentlichen Parameter einen Zeiger auf eine Variable vom Typ **int** übergeben, die Funktion aber als formellen Parameter eine Variable vom Typ **int** erwartet, erfolgt keine automatische Umwandlung, da es sich um zwei komplett verschiedene Arten von Werten handelt. So würde auch eine Struktur nicht automatisch in einen **long float**, einen **array** oder auch nur in irgendeine andere Struktur umgewandelt werden, da alle diese inkompatibel sind und nicht sinnvoll umgewandelt werden können. Alles in Kapitel 3 über Typenkompatibilität gesagte gilt auch für die Kompatibilität beim Funktionsaufruf. Genauso muß der Typ des Rückgabewertes, in diesem Fall **void**, mit dem erwarteten Typ im Funktionsaufruf kompatibel sein. Andernfalls gibt der Compiler eine Warnung aus.

5.3 Wie funktioniert die Prototypisierung?

Hier hast Du die Gelegenheit, die Prototypisierung selbst auszuprobieren und festzustellen, welche Fehlermeldungen Du erhältst, wenn Du einen Fehler machst. Ändere die Parameter in Zeile 12 so ab, daß dort (12.2, 13, 12345) steht und sieh Dir an, was der Compiler dazu sagt. Wahrscheinlich wird er gar nichts sagen, weil alle Typen kompatibel sind. Änderst Du die Parameter aber zu (12.0, 13) wirst Du eine Fehlermeldung

oder zumindest eine Warnung erhalten, weil Du zu wenig Argumente angegeben hast. So sollte sich der Compiler auch melden, wenn Du einen der Parameter in Zeile 13 in eine Adresse verwandelst, indem Du ein & davor setzt. Schließlich änderst Du noch das erste Wort in Zeile 4 von **void** in **int** und schaust, welchen Fehler der Compiler ausgibt. Zunächst wirst Du den Funktionskopf in Zeile 18 so abändern müssen, daß er mit dem Prototypen übereinstimmt, dann wirst Du festestellen, daß die Funktion keinen Wert zurückgibt. Wenn Du diese Änderungen vorgenommen hast, solltest Du gesehen haben, daß Prototypen wirklich etwas Sinnvolles für Dich tun können.

Kompiliere dieses Programm und führe es aus, dann ändere es wie oben besprochen und versuche nach jeder Änderung, es zu kompilieren.

5.4 Ein bisschen mehr über Prototypen

Beispielprogramm: PROTYP2.CPP

Das nächste Programm, PROTYP2.CPP hält einige weitere Informationen über Prototypen für Dich bereit. Es ist identisch mit dem letzten, von einigen kleineren Änderungen abgesehen. Wir haben die Variablennamen im Prototypen in Zeile 4 weggelassen, nur um zu zeigen, daß sie der C++ Compiler wirklich als Kommentare interpretiert. Der Funktionskopf wurde anders formatiert, um jeden Parameter kommentieren zu können. Dies soll beim Lesen des Programmes eine Hilfe bieten. Du solltest aber bedenken, daß Kommentare die sorgfältige Auswahl der Variablennamen nicht ersetzen sollen und können.

5.4.1 Hat die Protoypisierung Nachteile?

In Hinblick auf die Größe des Programmes oder die Geschwindigkeit bringt Prototypisierung keinerlei Nachteile mit sich. Die Prototypen werden beim Kompilieren zur Kontrolle herangezogen und verlängern damit die Dauer eines Kompilierungsvorganges unwesentlich. Wenn Du nur einen Fehler durch die Prototypisierung findest, den Du andernfalls mit einem Debugger suchen hättest müssen, hat es sich schon ausgezahlt.

Prototypisierung wirkt sich also lediglich auf die Größe der Quelldatei und auf die Dauer der Kompilation aus, beides ist aber vernachlässigbar.

Kompiliere das Programm und führe es aus. Du wirst feststellen, daß es dem letzten gleicht, die Änderungen im Prototypen und die Entfernung der Typenumwandlung in der letzten Zeile der Funktion ausgenommen.

5.5 Übergabe per Referenz

Beispielprogramm: UEBREF.CPP

Das Programm UEBREF.CPP ist ein Beispiel für die Übergabe per Referenz, ein Konzept, das in ANSI-C nicht verfügbar ist. Wir haben die Referenzvariable in Kapitel 2 schon erwähnt und gleichzeitig festgehalten, daß Du es vermeiden solltest, sie so einzusetzen, wie wir es dort getan haben. Dieses Programm zeigt eine Situation, wo wir

sie sehr zu unserem Vorteil verwenden. Die Übergabe per Referenz erlaubt es uns, einer Funktion eine Variable zu übergeben, wobei alle Änderungen an der Variable innerhalb der Funktion an das Hauptprogramm Übergeben werden. In ANSI-C wird derselbe Effekt durch die Übergabe eines Zeigers auf die Variable erreicht, aber die Verwendung einer Referenz ist ein wenig sauberer.

Beachte, daß die zweite Variable im Prototypen in Zeile 4 ein & vor dem Variablennamen hat. Dies weist den Compiler an, diese Variable wie einen Zeiger auf die eigentliche Variable zu behandeln. So wird praktisch die eigentliche Variable vom Hauptprogramm in der Funktion verwendet. In der Funktion selbst wird die Variable *Ein2* in den Zeilen 24 bis 27 genauso verwendet wie jede andere auch, aber sie verhält sich, als würden wir die eigentliche vom Hauptprogramm übergebene Variable verwenden und nicht eine Kopie davon. Die andere Variable, *Ein1*, wird wie eine ganz normale ANSI-C Variable verwendet. Der Variablenname *Ein2* ist also ein Synonym für die Variable *Index* im Hauptprogramm, der Name *Ein1* hingegen bezieht sich auf eine Kopie der Variable *Zaehler* im Hauptprogramm. Im Gebrauch wird ein Zeiger an die Funktion übergeben und automatisch dereferenziert, wenn er in der Funktion verwendet wird. Das ist Dir, der Programmiererin, natürlich klar.

Wenn Du es vorziehst, die Variablennamen im Prototypen wegzulassen, sähe Dein Prototyp folgendermaßen aus: void Pfusche(int, int &);

Als Pascal-Programmiererin wirst Du bemerkt haben, daß die Variable *Ein1* wie ein normaler Parameter in einem Pascal-Funktionsaufruf behandelt wird, ein Aufruf per Wert. Die Variable *Ein2* allerdings wird behandelt wie eine Variable mit dem reservierten Wort VAR davor, was üblicherweise Aufruf per Referenz genannt wird. Wie wir schon festgestellt haben, ist eine Referenzvariable in C++ eigentlich ein sich selbst dereferenzierender Zeiger, der sich auf den eigentlichen Wert bezieht, beziehungsweise darauf zeigt.

Wenn Du dieses Programm kompilierst und dann ausführst, wirst Du bemerken, daß die erste Variable in der Funktion zwar geändert wurde, aber wieder ihren alten Wert erhält, wenn wir zum Hauptprogramm zurückkehren. Auch die zweite Variable wurde in der Funktion geändert, diese Änderung spiegelt sich aber auch im Hauptprogramm wieder, was wir erkennen, wenn die Werte der Variablen am Bildschirm ausgegeben werden. Es sollte jetzt klar sein, daß eine Referenz es Dir erlaubt, einen Parameter per Referenz an eine Funktion zu übergeben. Wenn Du erst einmal ein wenig Erfahrung mit Referenzen gesammelt hast, wirst Du sie einsetzen, um die Effizienz einiger Deiner Programme zu steigern. Die Übergabe einer großen Struktur kann mit einer Referenz sehr effizient gestaltet werden.

5.6 Standardparameter

Beispielprogramm: STANDARD.CPP

Das Programm STANDARD.CPP ist ein Beispiel für die Verwendung von Standardparametern in C++. Dieses Programm erweckt sicherlich einen etwas eigenartigen Eindruck, weil es für einige Parameter im Prototypen Standardwerte festlegt. Diese Stan-

dardwerte können aber sehr nützlich sein, wie wir gleich sehen werden.

Der Prototyp sagt uns, daß der erste Parameter, *Laenge*, bei jedem Funktionsaufruf angegeben werden muß, da für ihn kein Standardparameter festgelegt wurde. Den zweiten Parameter, *Weite*, müssen wir nicht bei jedem einzelnen Aufruf angeben, und wenn wir ihn nicht explizit angeben, verwendet das Programm den Wert 2 für die Variable *Weite* innerhalb der Funktion. So ist auch der dritte Parameter optional, und wird er nicht angegeben, wird in der Funktion der Wert 3 für die Variable *Hoehe* verwendet.

In Zeile 11 dieses Programmes geben wir alle drei Parameter explizit an, es ist also nichts Außergewöhnliches an diesem Funktionsaufruf zu bemerken. In Zeile 12 geben wir allerdings nur zwei Werte an und verwenden also den Standardparameter für die dritte Variable. Das System reagiert genau so, als hätten wir die Funktion mit Volumen(x, y, 3) aufgerufen, da der Standardwert für den dritten Parameter 3 ist. In Zeile 13 geben wir beim Funktionsaufruf nur mehr einen Wert an, der für den ersten formellen Parameter verwendet wird, die beiden anderen nehmen die Standardwerte an. Das System wird verhält sich, als hätten wir die Funktion mit Volumen(x, 2, 3) aufgerufen. Beachte, daß die Ausgabe dieser drei Zeilen umgekehrt erfolgt. Darauf werden wir in kürze eingehen.

Es gibt einige Regeln, die zwar offensichtlich sind, aber dennoch hier aufgeführt werden sollen. Sobald ein Parameter in der Liste der formellen Parameter einen Standardwert erhält, müssen auch die folgenden Parameter Standardwerte haben. Es ist nicht möglich, in der Mitte der Liste ein Loch zu lassen, die Vergabe von Standardwerten erfolgt von hinten nach vorne. Selbstredend müssen die Standardwerte die richtigen Variablentypen haben, andernfalls gibt der Compiler eine Fehlermeldung aus. Die Standardwerte können entweder im Prototypen oder im Funktionskopf angegeben werden, nicht jedoch in beiden. Würden die Standardwerte an beiden Orten angegeben, müßte der Compiler nämlich nicht nur diese Standardwerte verwenden, er müßte auch genau überprüfen, ob die beiden Werte auch wirklich identisch sind. Dies würde ein schon sehr schwieriges Unterfangen, das der Entwicklung eines C++ Compilers, noch weiter komplizieren.

Es sei hier wärmstens empfohlen, die Standardparameter im Prototypen anzugeben und nicht im Funktionskopf. Der Grund dafür wird klar werden, sobald wir objektorientierte Programmiertechniken anwenden.

5.7 Warum ist die Ausgabe so wirr?

Wenn der Compiler auf eine **cout** Anweisung stößt, durchsucht er zuerst die komplette Zeile von rechts nach links auf etwaige Funktionen, dann werden die Daten nacheinander ausgegeben, von links nach rechts. Deshalb wird in Zeile 11 zuerst die Funktion *Volumen()* berechnet und deren interne Ergebnisse zuerst ausgegeben. Dann werden die einzelnen Felder von **cout** von links nach rechts ausgegeben, „Einige Daten sind“ also als nächstes. Schließlich wird der Rückgabewert von *Volumen()* im Format **int**, dem Typ des Rückgabewertes, ausgegeben. Das Resultat ist, daß die Ausgabe nicht in der erwarteten Reihenfolge erfolgt, wenn die Zeilen 11 bis 13 abgearbeitet werden. (Da die Ausgabe nicht so erfolgt, wie man das intuitiv erwarten würde, denkt man natürlich an eine Schwäche in der Sprache. Eine Nachfrage bei Borland International, Autoren von

Turbo C++ und Borland C++ ergab, daß der Code ordnungsgemäß abgearbeitet wird.) Dein Compiler wird dieses Programm womöglich nicht richtig ausführen. Dann müßt Du entweder eine Möglichkeit finden, den Compiler dazu zu bringen, Ausgaben mit *printf()* und **cout** nebeneinander zu akzeptieren oder Du änderst die *printf()* Anweisungen in äquivalente **cout** Anweisungen um.

Zusätzlich haben wir noch immer das Problem, das auftritt, wenn wir **cout** und *printf()* vermischen, wie in Kapitel 2 beim Programm NACHR.CPP besprochen. Schlußendlich werden aber alle standard(?)konformen Compiler dieses Problem überwinden.

Die Zeilen 15 bis 18 entsprechen den Zeilen 11 bis 13, nur sind die Anweisungen auf zwei Zeilen aufgeteilt, sodaß die Ausgabe in der erwarteten Reihenfolge erfolgt.

Kompiliere das Programm und führe es aus, wenn Du es verstanden hast. Die eigenartige Ausgabereihenfolge wird uns übrigens im Laufe dieser Einführung noch einmal begegnen.

5.8 Variable Parameter-Anzahl

Beispielprogramm: VARPAR.CPP

Das Programm VARPAR.CPP gibt ein Beispiel für die Verwendung von einer veränderbaren Anzahl an Argumenten in einem Funktionsaufruf.

Wir haben einige Mühen auf uns genommen, um den Compiler dazu zu bringen, für uns zu kontrollieren, ob wir in Funktionsaufrufen die richtige Anzahl und die richtigen Typen von Parametern verwenden. Ab und an mag es aber vorkommen, daß wir eine Funktion schreiben wollen, die unterschiedliche Anzahlen von Parametern verwenden kann. Die *printf()* Funktion ist ein gutes Beispiel dafür. ANSI-C stellt uns in der Datei „stdarg.h“ drei Makros zur Verfügung, mit denen sich das bewerkstelligen läßt. Diese können wir auch in C++ verwenden, müssen aber einen Weg finden, die strengere Typenkontrolle zu umgehen, die bei jeder Funktion in C++ durchgeführt wird. Dies bewirken die drei Punkte in Zeile 6. Dieser Prototyp sagt aus, daß als erstes ein Argument des Typs **int** notwendig ist, dann aber wird vom Compiler keine weiter Typenkontrolle durchgeführt.

Das Hauptprogramm besteht aus drei Aufrufen der Funktion, jeder mit einer anderen Anzahl von Parametern, und das System ignoriert dies einfach. Du könntest so viele verschiedene Variablentypen in den Funktionsaufruf einbauen wie Du willst. Solange der erste Parameter eine Variable vom Typ **int** ist, wird sich das System jede erdenkliche Mühe geben, Dein Programm zu kompilieren und es auszuführen. Natürlich erfolgt auch keinerlei Kontrolle nach dem ersten Argument, es obliegt also allein Dir, die richtigen Parametertypen im Funktionsaufruf zu verwenden.

In diesem Fall zeigt der erste Parameter die Anzahl der zusätzlichen Argumente an. In diesem einfachen Beispiel geben wir nur die Zahlen auf dem Bildschirm aus, um zu demonstrieren, daß sie wirklich richtig abgearbeitet wurden.

Du hast sicherlich festgestellt, daß die Verwendung von einer variablen Anzahl an Parametern zu sehr obskurem Quellcode führen kann und deshalb nur sehr selten angewendet werden sollte. Die Möglichkeit besteht jedenfalls, solltest Du etwas derartiges jemals benötigen. Kompiliere das Programm und lass’ es laufen.

5.9 Überladen von Funktionsnamen

Beispielprogramm: UEBERLAD.CPP

Das Programm UEBERLAD.CPP ist ein Beispiel für das Überladen von Funktionsnamen. Diese Technik ist in ANSI-C nicht verfügbar, sehr wohl aber in C++ und wird auch regelmäßig angewendet. Auf den ersten Blick wird sie etwas eigenartig anmuten, ist aber einer der Grundbausteine objektorientierten Programmierens. Die Nützlichkeit und den Sinn dieses Konstrukts werden wir im Laufe dieser Einführung noch sehr zu schätzen lernen.

Dieses Programm hat außer der Funktion *main()* noch drei zusätzliche Funktionen, und alle drei tragen denselben Namen. Deine erste Frage wird nun sein: „Welche Funktion rufe ich mit *MachWas()* eigentlich auf?“ Die Antwort auf diese Frage ist: Du rufst die Funktion mit der richtigen Anzahl an formellen Variablen des richtigen Typs auf. Rufst Du *MachWas()* mit einer ganzen Zahl oder einer Variable vom Typ **int** als eigentlichem Parameter auf, wird die Funktion, die in Zeile 25 beginnt, ausgeführt. Wenn der eigentliche Parameter vom Typ **float** ist, wird die Funktion in Zeile 30 aufgerufen, hat der Funktionsaufruf zwei **float** Werte oder Variablen als Argumente, wird die Funktion in Zeile 36 ausgeführt.

Der Rückgabewert wird zur Feststellung der passenden Funktion nicht herangezogen. Allein die Typen der formellen Parameter werden verwendet, um festzustellen, welche der überladenen Funktionen aufgerufen wird.

Das Schlüsselwort **overload** in Zeile 4 sagt dem System, daß es wirklich Deine ernste Absicht ist, den Namen *MachWas* zu überladen und dies nicht aus Versehen geschieht. Dies wird aber nur von sehr frühen Versionen von C++ verlangt. Neuere Versionen verlangen dieses Schlüsselwort nicht mehr, es ist aber aus Kompatibilitätsgründen nach wie vor (noch) erlaubt. Es ist nicht notwendig, **overload** zu verwenden, da das Überladen in C++ generell in einem Zusammenhang geschieht, wo es offensichtlich ist, daß der Funktionsname überladen wird.

Die eigentliche Auswahl, welche Funktion aufgerufen wird, wird beim Kompilieren getroffen und nicht bei der Ausführung des Programmes, sodaß es zu keinen Geschwindigkeitseinbußen kommt. Würden wir jeder der Funktionen mit dem überladenem Namen einen einzigartigen Namen verleihen, wäre kein Unterschied in der Größe des Programmes oder in seiner Geschwindigkeit festzustellen.

Das Überladen von Funktionsnamen mag Dir eigentümlich vorkommen und das ist es auch, wenn man an die Regeln von K&R oder ANSI-C gewöhnt ist. Hast Du aber einmal etwas Erfahrung mit C++ gesammelt, wirst Du das Überladen ganz selbstverständlich und oft in Deinen Programmen verwenden.

Beachte die Verwendung des Schlüsselwortes **const** in einigen Funktionsprototypen und -köpfen. Dies bewahrt die Programmiererin, wie gesagt, davor, den formellen Parameter innerhalb der Funktion zu ändern. In einer Funktion, die so kurz ist wie die Funktionen dieses Programmes, besteht keine wirkliche Gefahr einer irrtümlichen Zuweisung. In einer wirklichen Funktion kannst Du aber ganz schnell auf den ursprünglichen Zweck einer Variable vergessen, wenn Du die Funktion später einmal modifizierst.

5.10 Programmieraufgaben

1. Ändere den Typ der Variable Fluegel im Prototypen von PROTYP1.CPP zu float, sodaß eine Diskrepanz mit der Funktionsdefinition besteht. Gibt der Compiler eine Fehlermeldung aus?

2. Ändere die Funktionsdefinition in PROTYP1.CPP, sodaß sie mit dem geänderten Prototypen wieder übereinstimmt. Kompiliere das Programm und führe es aus, ohne die Funktionsaufrufe in den Zeilen 12 und 13 zu ändern. Erkläre das Ergebnis.

3. Lösche in STANDARD.CPP den Standardwert für Hoehe, nur um zu sehen, welchen Fehler der Compiler meldet. Nur die letzten Werte der Liste können Standardwerte annehmen.

4. Ändere die Funktionsnamen in UEBERLAD.CPP so ab, daß jeder Name einzigartig ist und vergleiche die Größe des Programmes mit jener des ursprünglichen Programmes.

6 Zugriffsschutz

Wie wir schon im ersten Kapitel festgestellt haben, erscheint einer Programmiererin, die viel Erfahrung in der prozeduralen Programmierung hat, des objektorientierte Programmieren reichlich unnatürlich. Mit diesem Kapitel beginnen wir unsere Definition objektorientierten Programmierens und widmen uns dem Zugriffsschtz, der dem Motto „divide et impera“ (teile und herrsche) folgt. Wie wir auch schon festgestellt haben, bringt das objektorientierte Programmieren natürlich wieder eine Reihe neuer Begriffe und Namen aufs Tableau. Lass Dich von der neuen Terminologie aber nicht abschrecken, es wird alles halb so schlimm sein, wenn wir uns einen Begriff nach dem anderen in einer sinnvollen Reihenfolge vornehmen.

Der Zugriffsschutz ist eigentlich das Erzeugen von Objekten, dem wir uns während des gesamten Kapitels widmen werden. Ein Objekt mit Zugriffsschutz wird oft auch ein abstrakter Datentyp genannt und um diesen dreht sich das gesamte objektorientierte Programmieren. Ohne Zugriffsschutz, der nur unter Verwendung einer oder mehrerer Klassen möglich ist, gibt es kein objektorientiertes Programmieren. Natürlich hat die Objektorientierung noch andere Aspekte, der Zugriffsschutz ist aber der Kern der Sache.

6.1 Wozu das alles?

Wir brauchen einen Zugriffsschutz weil wir Menschen sind und Menschen nun einmal Fehler machen. Wenn wir Quellcode richtig vor (falschem) Zugriff schützen, bauen wir eine Mauer rundherum, die den Code vor zufälligen Änderungen durch all die dummen Fehler, die wir so gerne machen, schützt. Außerdem können wir Fehler auf ein relativ kleines Gebiet zurückführen und sie so schneller finden und beheben. Einiges mehr noch wird über die Vorteile des Zugriffsschutzes zu sagen sein, wenn wir in diesem Kapitel voranschreiten.

6.2 Kein Zugriffsschutz

Beispielprogramm: OFFEN.CPP

Das Programm mit dem Namen OFFEN.CPP ist ein wirklich dummes Programm, es macht nämlich so gut wie gar nichts. Es soll aber unser Ausgangspunkt für die Diskussion des Zugriffsschutzes (auch als das Vorenthalten von Information bekannt) sein. Der Zugriffsschutz ist ein wichtiger Teil des objektorientierten Programmierens und Du solltest am Ende dieses Kapitels mit Fug und Recht von sich behaupten können, ihn zu verstehen.

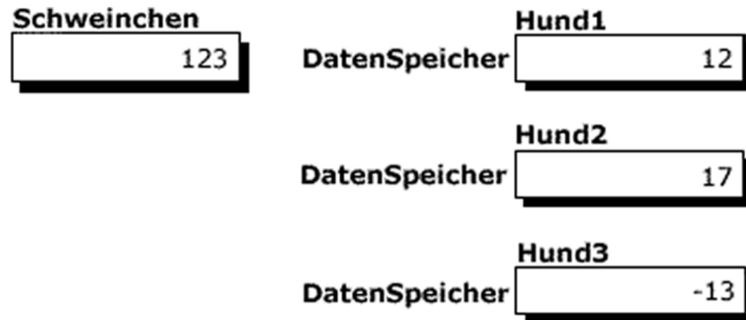

Abbildung 6.1: Speicherzustand nach Zeile 17

In den Zeilen 4 bis 7 definieren wir eine sehr einfache Struktur, die aus einer einzigen Variable vom Typ **int** besteht. Das ist nicht unbedingt sinnvoll, soll aber das Problem illustrieren, das wir in diesem Kapitel lösen wollen. In Zeile 11 deklarieren wir drei Variablen, die alle aus einer einzelnen Variable vom Typ **int** bestehen und alle drei überall innerhalb der Funktion *main()* verfügbar sind. Wir können jeder Variable Werte zuweisen, sie inkrementieren, lesen, ändern oder was auch immer mit ihr anstellen. Einige der möglichen Operationen werden in den Zeilen 14 bis 22 gezeigt und sollten mit ein wenig Erfahrung in C klar sein.

Eine einzelne lokale Variable mit dem Namen *Schweinchen* deklarieren und verwenden wir mehr oder weniger parallel, um zu zeigen, daß es sich bei diesem Code um nichts Außergewöhnliches handelt. Abbildung 6.1 stellt den Speicherzustand nach der Abarbeitung von Zeile 17 bildhaft dar.

Schau Die dieses Beispielprogramm genau an, es ist nämlich die Basis unseres Studiums des Zugriffsschutzes. Kompiliere das Programm und führe es aus, dann gehen wir weiter zu unserem nächsten Programm, dem ersten mit richtigem Zugriffsschutz.

6.3 Zugriffsschutz

Beispielprogramm: KLAS.CPP

Das Programm KLAS.CPP ist unser erstes Beispiel für ein Programm mit ein wenig Zugriffsschutz. Das Programm ist identisch mit dem letzten, außer daß es einige Operationen etwas anders durchführt. Wir werden uns die Unterschiede der Reihe nach ansehen und erklären, was passiert. Bedenke, daß es sich hier um ein wirklich triviales Beispiel handelt, wo die Sicherheitsmaßnahmen nicht von Nöten sind, sondern nur dazu dienen sollen, ihre Verwendung (in einem komplizierteren Programm) zu illustrieren.

Der erste Unterschied besteht darin, daß wir anstatt einer Struktur in Zeile 4 eine Klasse haben. Der einzige Unterschied zwischen Struktur und Klasse ist der, daß die Klasse mit einer privaten Sektion beginnt, die Struktur mit einer öffentlichen. Das Schlüsselwort **class** wird verwendet, um eine Klasse zu deklarieren, wie hier gezeigt.

Die Klasse mit dem Namen *Daten* beseht aus einer einzelnen Variable mit dem Namen *DatenSpeicher* und zwei Funktionen, eine mit dem Namen *SetzeWert()*, die andere mit

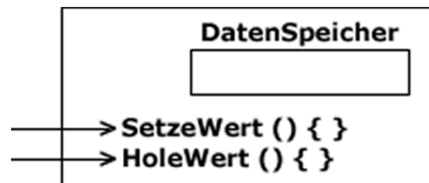

Abbildung 6.2: Eine Mauer schützt die Daten

dem Namen *HoleWert()*. Eine etwas komplettere Definition einer Klasse wäre etwa die einer Gruppe von Variablen und einer oder mehrerer Funktionen, die mit diesen Variablen arbeiten. Wir werden all das sehr bald in eine sinnvolle und nützliche Ordnung bringen.

6.4 Was sind private Elemente?

Alle Daten am Anfang einer Klasse sind automatisch private Daten. Deshalb kann auf Daten, die am Beginn einer Klasse stehen, von außen nicht zugegriffen werden, sie sind sozusagen versteckt. Darum kann auf die Variable mit dem Namen *DatenSpeicher*, die ein Teil des in Zeile 24 definierten Objektes (was genau ein Objekt ist werden wir später definieren) mit dem Namen *Hund1* ist, nirgends innerhalb der Funktion *main()* zugegriffen werden. Es ist, als hätten wir um die Variablen eine Mauer errichtet, um sie vor äußeren Programmeinflüssen zu schützen. Es erscheint eigentlich dumm, in der Funktion *main()* eine Variable zu definieren, die wir gar nicht nutzen können, genau das haben wir aber soeben getan.

Abbildung 6.2 illustriert die Klasse mit ihrer Mauer rund um die Daten, um diese zu schützen. Dir sind sicherlich die kleinen Löcher aufgefallen, die wir offen gelassen haben, um der Benutzerin den Zugriff auf die Funktionen *SetzeWert()* und *HoleWert()* zu ermöglichen. Diese Löcher haben wir geöffnet, indem wir die Funktionen als öffentliche Elemente der Klasse definiert haben.

6.5 Was sind öffentliche Elemente?

In Zeile 7 führen wir ein neues Schlüsselwort ein: **public**. Dieses Schlüsselwort sagt aus, daß alles darauf folgende auch außerhalb der Klasse verfügbar ist. Da wir unsere zwei Funktionen nach dem Schlüsselwort **public** definiert haben, sind sie öffentliche Funktionen und können von jeder Funktion aus aufgerufen werden, die im Gültigkeitsbereich dieses Objektes liegt. Das öffnet zwei Löcher in der Mauer, die wir um unsere Daten gebaut haben, um sie zu schützen. Denke aber immer daran, daß die private Variable für die aufrufende Funktion nicht verfügbar ist. Deshalb können wir die Variable nur verwenden, indem wir eine der zwei Funktionen aufrufen, die wir als öffentliche Funktionen definiert haben. Diese Funktionen werden Elementfunktionen genannt, da sie Elemente der Klasse sind.

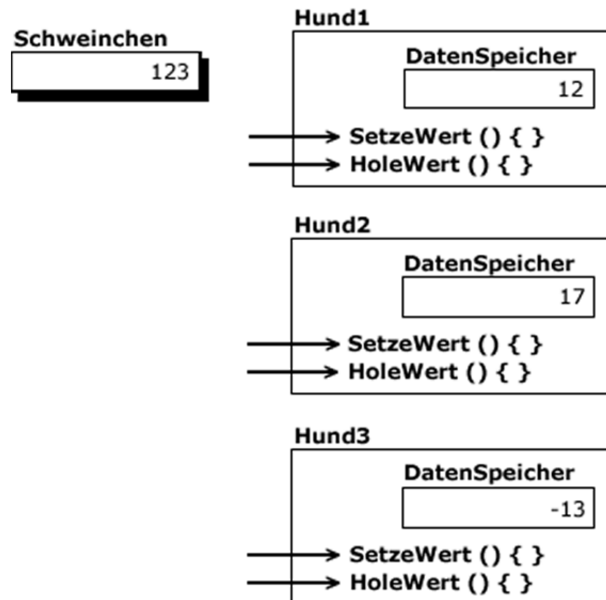

Abbildung 6.3: Speicherinhalt nach Zeile 30

Nachdem wir zwei Funktionen deklariert haben, müssen wir sie nun auch definieren und angeben, was sie eigentlich tun sollen. In den Zeilen 12 bis 20 tun wir das ganz normal, mit dem einzigen Unterschied, daß den Funktionsnamen der Klassenname, getrennt durch einen zweifachen Doppelpunkt, vorausgeht. Die beiden Funktionsdefinitionen werden die Implementation der Funktionen genannt. Wir müssen den Klassennamen angeben, da es uns freisteht, denselben Funktionsnamen innerhalb mehrerer Klassen zu verwenden und der Compiler wissen muß, welche Funktionsimplementation welcher Klasse zuzuordnen ist.

Eine der wichtigsten Erkenntnisse, die wir hier gewinnen sollen, ist jene, daß all die privaten Daten der Klasse innerhalb der Implementationen der Elementfunktionen der Klasse ganz normal für Änderungen zur Verfügung stehen. Innerhalb der Funktionsimplementationen, die ja zur Klasse gehören, kannst Du mit den privaten Daten der Klasse alles anstellen, private Daten anderer Klassen sind allerdings nicht verfügbar. Das ist auch der Grund, warum wir dem Funktionsnamen den Klassennamen voranstellen müssen, wenn wir sie definieren. Abbildung 6.3 zeigt den Speicher nach dem Abarbeiten der Zeile 30.

Es soll nicht unerwähnt bleiben, daß es natürlich erlaubt ist, private Variablen und Funktionen zu deklarieren und dann im öffentlichen Teil zusätzliche Variablen und Funktionen. In den meisten Fällen werden die Variablen ausschließlich im privaten Teil deklariert und die Funktionen nur im öffentlichen. Manchmal wird man aber Variablen oder Funktionen auch im jeweils anderen Teil deklarieren. Das führt oft zu sehr praktischen Lösungen für spezielle Probleme, im Allgemeinen werden die Elemente aber nur in den erwähnten Teilen verwendet.

In C++ gibt es vier verschiedene Gültigkeitsbereiche für Variablen: global, lokal, in

einer Datei und in einer Klasse. Globale Variablen sind überall in der Datei, in der sie definiert werden, und auch in anderen Dateien verfügbar. Lokale Variablen sind auf eine einzelne Funktion beschränkt. Dateien, die außerhalb jedweder Funktion definiert werden, sind im auf ihre Definition folgenden Rest der Datei verfügbar. Eine Variable mit dem Gültigkeitsbereich einer Klasse schließlich ist überall innerhalb des Gültigkeitsbereiches der Klasse, der die Implementationen mit einschließt, verfügbar und nirgends sonst. Die Variable *DatenSpeicher* ist eine solche Variable.

Du wirst jetzt einigermaßen verwirrt sein, da wir eine Reihe von Regeln aufgestellt haben, ohne je Gründe für diese anzuführen. Lies weiter und Du wirst sehen, daß es doch recht vernünftige und brauchbare Gründe für all das gibt.

6.6 Mehr neue Termninologie

Wann immer Menschen etwas Neues entwickeln, sind sie der festen Überzeugung, daß das Neue durch einen neuen Namen auch würdig gefeiert werden muß. Da bildet die Objektorientierung keine Ausnahme, also müssen wir uns an einige neue Bezeichnungen für gute alte Bekannte gewöhnen, wenn wir lernen, sie effektiv einzusetzen. Um Dir dabei behilflich zu sein, werden wir hier einige dieser Begriffe aufzählen und beginnen hiermit, sie auch im Text zu benutzen, damit Du Dich an sie gewöhnen kannst. Du wirst möglicherweise nicht alle gleich verstehen, es ist aber sicherlich sinnvoll, sie möglichst früh einzuführen.

- Eine *Klasse* ist eine Gruppierung von Daten und Methoden (Funktionen). Eine Klasse ist einer Struktur in ANSI-C sehr ähnlich, ist sie doch nur ein Muster, das verwendet wird, eine Variable zu schaffen, die dann in einem Programm verwendet wird.

- Ein *Objekt* ist eine Instanz einer Klasse, was man sich so ähnlich verstellen kann wie eine Variable eine Instanz eines Variablentyps ist. Im Programm selbst verwendest Du Objekte, da diese Werte einthalten und verändert werden können.

- Eine *Methode* ist eine Funktion innerhalb einer Klasse. In Programmierbüchern werden Funktionen, die innerhalb einer Klasse verwendet werden, oft Methoden genannt.

- Eine *Nachricht* ist dasselbe wie ein Funktionsaufruf. Beim objektorientierten Programmieren senden wir Nachrichten anstatt Funktionen aufzurufen. Einsteilen kannst Du die beiden als identisch ansehen. Später in dieser Einführung werden wir sehen, daß es doch feine Unterschiede gibt.

Mit all dieser neuen Terminologie werden wir nun unser Studium des Programmes KLAS.CPP fortsetzen und zeigen, wie die Klasse verwendet wird. Wir halten einmal fest, daß wir eine Klasse mit einer Variablen und zwei Methoden haben. Die Methoden operieren mit den Variablen, wenn eine Nachricht sie anweist, das zu tun. In dieser

Einführung werden wir die Bezeichnungen Objekt und Variable austauschbar verwenden, da beide ganz gut wiedergeben, was ein Objekt wirklich ist.

Das folgende ist nur ein kleiner Punkt, kann daher aber auch leicht übersehen werden. Die Zeilen 8 und 9 enthalten die Prototypen für unsere zwei Methoden und sind damit unser erstes Beispiel für eine Prototypen innerhalb einer Klasse. Aus diesem Grund haben wir uns den Prototypen im letzten Kapitel so eingehend gewidmet. Was sagt Zeile 8 aus? Die Methode mit dem Namen *SetzeWert()* verlangt einen Parameter vom Typ **int** und gibt nichts zurück, der Rückgabetyp ist also **void**. Die Methode mit dem Namen *HoleWert()* allerdings hat keine Eingabeparameter, gibt aber einen Wert vom Typ **int** zurück an den Aufrufenden.

6.7 Das Senden einer Nachricht

Nach all den Definitionen in den Zeilen 1 bis 20 kommen wir endlich zum Programm, wo die Klasse auch wirklich zum Einsatz kommt. In Zeile 24 definieren wir drei Objekte der Klasse *Daten* und nennen sie *Hund1*, *Hund2* und *Hund3*. In dieser Zeile verwenden wir das Schlüsselwort **class** nicht noch einmal, einfach weil es nicht notwendig ist. Jedes der Objekte beinhaltet eine Information, auf die wir mittels der Methoden *SetzeWert()* und *HoleWert()* zugreifen können. Wir können diese Information aber nicht direkt setzen oder ihren Wert lesen, da diese Information innerhalb unserer Mauer rund um die Klasse versteckt ist. In Zeile 27 senden wir an das Objekt mit dem Namen *Hund1* eine Nachricht, die es anweist, den Wert der internen Variable auf 12 zu setzen. Obwohl dies wie ein Funktionsaufruf aussieht, wollen wir es richtiger das Senden einer Nachricht an das Objekt nennen. Du erinnerst Dich, daß das Objekt mit dem Namen *Hund1* eine zu ihm gehörige Methode mit dem Namen *SetzeWert()* hat, die den Wert der internen Variable auf den Parameter der Methode setzt. Sicherlich ist Dir aufgefallen, daß die Form sehr dem Zugriff auf ein Element einer Struktur ähnelt. Du gibst den Namen des Objektes an, gefolgt von einem Punkt und dann dem Namen der Methode. Genauso senden wir auch an die anderen zwei Objekte, *Hund2* und *Hund3* Nachrichten, ihre Werte auf die jeweils angegebenen Parameter zu setzen.

Die Zeilen 32 und 33 haben wir auskommentiert, da diese Aktionen nicht erlaubt sind. Die Variable mit dem Namen *DatenSpeicher* ist privat und damit außerhalb des Objektes selbst nicht verfügbar. Es sollte klar sein, daß die Daten, die das Objekt *Hund1* beinhaltet, in den Methoden von *Hund2* und *Hund3* nicht verfügbar sind, da es sich um verschiedene Objekte handelt. All diese Regeln sollen Dir dabei helfen, schneller besseren Code zu schreiben und wir werden bald sehen wie sie das tun.

Die andere für jedes Objekt definierte Methode kommt in den Zeilen 35 bis 37 zum Einsatz, um zu illustrieren, wie sie verwendet wird. Es wird jeweils eine weitere Nachricht an das jeweilige Objekt gesendet und der zurückgegebene Wert mittels der Strömebibliothek am Bildschirm ausgegeben.

Box
Hoehe 12
Breite 10

Quadrat
Hoehe 8
Breite 8

Fahnenmast
Hoehe 50
Breite 6

Abbildung 6.4: Speicherinhalt nach Zeile 34

6.8 Eine normale Variable verwenden

Wir deklarieren eine andere Variable mit dem Namen *Schweinchen* und verwenden sie in diesem Beispielprogramm, um zu zeigen, daß eine normale Variable mit den Objekten gemischt und ganz normal verwendet werden kann. Die Verwendung dieser Variable sollte Dir keine allzu großen Rätsel aufgeben. Wenn Du dieses Programm also verstanden hast, kompiliere es und laß es laufen. Dann entferne die Kommentarzeichen aus den Zeilen 32 und 33 und achte darauf, welche Fehlermeldung der Compiler ausgibt.

6.9 Ein Programm mit Problemen

Beispielprogramm: MASTOFFN.CPP

Das Programm MASTOFFN.CPP ist ein Beispiel, das einige ernsthafte Probleme aufzeigt, die wir dann im nächsten Beispielprogramm mithilfe des Zugriffsschutzes überwinden werden.

Wir deklarieren zwei Strukturen, eine für ein *Rechteck*, die andere für einen *Masten*. Die einzelnen Datenfelder sollten klar sein, mit Ausnahme der *Tiefe* des *Fahnenmasten*, wobei es sich darum handelt, wie tief er im Boden verankert ist. Die gesamte Länge des Fahnenmasten ergibt sich demnach aus der *Laenge* plus der *Tiefe*.

Abbildung 6.4 zeigt den Speicher für dieses Programm nach dem Abarbeiten der Zeile 34. Mit Deiner Erfahrung im Programmieren in ANSI-C sollten beim Verständnis dieses Programmes keinerlei Probleme auftreten. Einzig das Ergebnis der Zeile 40, wo wir die *Hoehe* des *Quadrat* mit der *Breite* der *Box* multiplizieren, wird Dich etwas verwundern. Das ist in ANSI-C genauso wie in C++ erlaubt, hat aber keine „weltliche“ Bedeutung und keinen Sinn, da die Daten von zwei verschiedenen Elementen stammen. So ist das Ergebnis von Zeile 42 noch sinnloser, denn das Produkt der *Hoehe* des *Quadrat* und der *Tiefe* des *Fahnenmast* hat absolut keine Bedeutung in irgendeinem denkbaren physikalischen System. In einem Programm, das so einfach ist wie dieses, ist der Fehler offensichtlich, in einem größeren Programm schleichen sich solche Fehler aber sehr leicht ein und sind oft nur sehr mühsam wieder zu finden.

Wäre es nicht schön, wenn wir einen Weg fänden, solche dummen Fehler in großen Industrieprogrammen zu vermeiden? Hätten wir ein gutes Programm, das genau definiert,

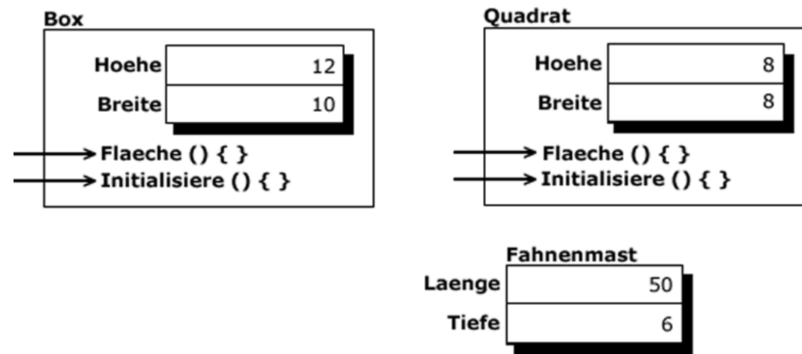

Abbildung 6.5: Die beiden Objekte von KLASMAST.CPP

welche Dinge wir mit einem *Rechteck* anstellen können und ein zweites Programm, das genau definiert, was wir mit einem *Mast* tun können und wären die Daten vor äußerem Zugriff geschützt, dann könnten wir solche Dinge verhindern. Wenn diese Elemente miteinander arbeiten sollen, können wir sie nicht in einzelnen Programmen verwirklichen, wohl aber können wir sie auf Klassen aufteilen, um unser Ziel zu erreichen.

Es wird Dich nicht überraschen, daß unser nächstes Programm genau das für uns tun wird und noch dazu in einer eleganten Art und Weise. Bevor wir aber weitergehen, solltest Du dieses Programm kompilieren und laufen lassen und Dich an den abstrusen Ergebnissen erfreuen.

6.10 Objekte schützen Daten

Beispielprogramm: KLASMAST.CPP

Das Beispielprogramm KLASMAST.CPP illustriert den Schutz von Daten in einem sehr einfachen Programm.

In diesem Programm haben wir aus dem *Rechteck* eine Klasse mit denselben zwei Variablen, die nun privat sind, und zwei Methoden, die mit diesen Variablen arbeiten können, gemacht. Eine Methode dient dazu, die Werte der erzeugten Objekte zu initialisieren und die andere Methode gibt den Flächeninhalt des Objektes zurück. Die Methoden definieren wir in den Zeilen 13 bis 22 so wie wir es oben in diesem Kapitel beschrieben haben. Der *Mast* ist weiterhin eine Struktur, um zu zeigen, daß die beiden parallel verwendet werden können und C++ tatsächlich eine Erweiterung von ANSI-C ist.

In Zeile 35 definieren wir zwei Objekte, die wir wieder *Box* und *Quadrat* nennen, diesmal können wir ihren Elementen allerdings nicht direkt Werte zuweisen, da es sich um private Elemente der Klasse handelt. Bild 5-5 zeigt die beiden Objekte, die im aufrufenden Programm verfügbar sind. Deshalb haben wir die Zeilen 38 bis 40 auskommentiert und senden anstatt dessen in den Zeilen 42 und 43 Nachrichten an die Objekte, die sie anweisen, sich selbst mit den als Parametern angegebenen Werten zu initialisieren. Der *Fahnenmast* wird in derselben Weise initialisiert wie im vorigen Programm. Die Ver-

wendung der Klasse bewahrt uns vor den unsinnigen Berechnungen, die wir im letzten Beispielprogramm angestellt haben, da wir die Fläche eines Objektes nur mit den im Objekt selbst gespeicherten Daten berechnen können. Nun verwenden wir den Compiler, um die sinnlose Rechnerei zu verhindern. Das Endergebnis ist, daß die sinnlosen Berechnungen des letzten Programmes in diesem Programm nicht mehr möglich sind, also sind die Zeilen 52 bis 55 auskommentiert. Auch hier ist es vielleicht schwierig, die Sinnhaftigkeit des ganzen in einem so einfachen Programm zu erkennen. In einem umfassenden Programm macht sich die Verwendung des Compilers, um sicherzustellen, daß die Regeln eingehalten werden, aber sehr bezahlt.

Obwohl *Quadrat* und *Box* beide Objekte der Klasse *Rechteck* sind, sind ihre privaten Daten vor dem jeweils anderen Objekt versteckt, sodaß keines absichtlich oder unabsichtlich die Daten des anderen verändern kann.

Das ist der abstrakte Datentyp, von dem wir am Beginn dieses Kapitels gesprochen haben, ein Modell mit privaten Variablen, um Daten zu speichern, und Operationen, die mit diesen Daten ausgeführt werden können. Die einzigen Operationen, die mit den Daten arbeiten können, sind die Methoden, was viele Fehler, die allzu leicht passieren, verhindern hilft. Der Zugriffsschutz bindet die Daten und die Prozeduren, oder Methoden, eng aneinander und schränkt den Gültigkeitsbereich und die Sichtbarkeit beider ein. Auch hier treffen wir auf die Taktik des Teilen und Herrschens, mit der ein Objekt vom Rest des Codes abgesondert und von diesem komplett isoliert entwickelt wird. Dann erst wird es in den Rest des Codes mit Hilfe einiger einfacher Schnittstellen eingebaut.

Eine Studie hat vor einigen Jahren gezeigt, daß Programmiererinnen viel eher bei Daten Fehler machen als bei Code. Also war es klar, daß man die Qualität von Software alleine schon dadurch würde verbessern können, daß man die Daten vor unabsichtlichen Veränderungen schützt. Das ist der Ursprung des Zugriffsschutzes und diese Idee hat sich im Laufe der Jahre sehr bewährt.

6.11 Hast Du diese Technik schon einmal benutzt?

Ein recht gutes Beispiel für diese Technik geben die Dateibearbeitungsbefehle in ANSI-C ab. Du kannst auf die Daten in einer Datei nur über den „Umweg“ der Funktionen, die Dein Compiler dafür zur Verfügung stellt, zugreifen. Du hast keinen direkten Zugriff auf die eigentlichen Daten, weil es Dir unmöglich wäre, diese zu adressieren. Daher sind diese Daten private Daten, und die Funktionen sind den Methoden von C++ sehr ähnlich.

Zwei Aspekte dieser Technologie sind in der Softwareentwicklung besonders wichtig. Erstens bekommst Du alle Daten, die sie brauchen, vom System, da die Schnittstelle komplett ist und alles abdeckt, zweitens aber bekommst Du keine Daten, die Du nicht brauchst. Du wirst davor abgehalten (oder bewahrt), in das Dateimanagement einzugreifen und möglicherweise Unfug anzustellen und dabei Daten zu verlieren. Du kannst auch nicht die falschen Daten verwenden, da die Funktionen einen seriellen Zugriff auf diese verlangen. Daß es sich bei diesem Beispiel aber um ein relativ schwaches handelt, sieht man daran, daß es für eine versierte C Programmiererin ein Leichtes ist, den Zugriffsschutz zu umgehen.

Ein weiteres Beispiel für einen schwachen Zugriffsschutz sind die Bildschirm- und Tastaturroutinen. Du kannst in deren Arbeit nicht eingreifen, hast aber über Schnittstellen auf alle Daten Zugriff, die Du benötigst, um sie effizient zu verwenden.

Stell Dir vor, Du entwickelst ein Programm, das die Charakteristika von Fahnenmasten analysiert. Du würdest nicht unbedingt irgendwelche Daten, die angeben, wo Dein Programm gespeichert ist, als die Höhe eines Fahnenmasten verwenden wollen oder die Cursorposition als des Fahnenmasten Dicke oder Farbe. Der gesamte Code für den Fahnenmasten wird separat entwickelt und erst wenn er fertig ist, kann er auch verwendet werden. Für die Verwendung genügen dann einige wenige Operationen mit der Klasse. Die Tatsache, daß die Daten vor Dir sicher verwahrt und versteckt sind, bewahrt Dich vor „Dummheiten“, wenn Du um 2 Uhr früh arbeitest, um irgendeine deadline (heißt nicht umsonst so) einzuhalten. Dies wird Zugriffsschutz genannt und ist einer der großen Vorteile, die objektorientiertes gegenüber prozeduralem Programmieren hat.

Wie aus dem oben gesagten deutlich wird, ist objektorientiertes Programmieren als solches nicht wirklich neu, in einigen wenigen und kleinen Punkten wird es verwendet, seit es Computer gibt. Die neueste Entwicklung ist es allerdings, der Programmiererin die Möglichkeit zu geben, daran teilzuhaben und ihre Programme zu partionieren, um die Fehleranfälligkeit zu verringern und die Qualität zu erhöhen.

6.12 Welche Nachteile gibt es?

Es sollte klar sein, daß diese Technik die Effizienz der Programme etwas leiden läßt, da jeder Zugriff auf ein Element der Klasse die Zeit und Ineffizienz eines Funktionsaufrufes, oder dann eher Methodenaufrufes, benötigt. Die Zeit, die beim Kompilieren verloren geht, ist aber bei der Fehlersuche schnell wettgemacht, da ein Programm, das aus Objekten besteht, wesentlich einfacher und schneller zu verstehen ist.

Dieses Programm ist so einfach, daß es unsinnig ist, nachzuforschen, wo wir möglicherweise profitiert haben. In einem wirklichen Projekt kann es ein großer Gewinn sein, wenn eine Entwicklerin all die Details des Rechtecks entwickelt, programmiert und dann zur Verfügung stellt. Genau das ist schon für Dich gemacht worden, wenn Du Dir den Monitor als Objekt vorstellst. Es gibt ein komplettes Arsenal an programmierten und fehlerbereinigten Routinen, die Du verwenden kannst, um am Bildschirm das zu tun, was Du willst. Alles, was Du tun mußt, ist, die Schnittstelle zu benutzen lernen und Dich darauf verlassen, daß die Funktionen auch funktionieren. Du mußt Dir die Implementation nicht ansehen oder sie gar verstehen, vorausgesetzt, es funktioniert alles so, wie Du es Dir vorstellst. Es ist Dir nicht möglich, die Größe Deines Bildschirms mit der Tiefe des Fahnenmasten zu multiplizieren, einfach weil Du die Informationen dazu nicht hast.

Wenn Du die Vorteile dieses Programmierstils verstanden hast kompiliere das Programm und führe es aus (Oper?).

6.13 Konstruktoren und Destruktoren

Beispielprogramm: KONSMAST.CPP

Das Programm KONSMAST.CPP soll uns jetzt mit Konstruktoren und Destruktoren bekannt machen.

Bis auf die Tatsache, daß wir einen Konstruktor und einen Destruktor hinzugefügt haben, ist dieses Beispielprogramm identisch mit dem letzten. In Zeile 9 deklarieren wir den Konstruktor, der immer denselben Namen hat wie die Klasse selbst, und in den Zeilen 15 bis 19 definieren wir ihn. Der Konstruktor wird vom C++ System automatisch aufgerufen, wenn ein Objekt deklariert wird und verhindert die Verwendung von uninitialisierten Variablen. Wenn wir also in Zeile 48 das Objekt mit dem Namen *Box* deklarieren, ruft das System automatisch den Konstruktor auf. Der Konstruktor setzt die zwei Variablen, *Hoehe* und *Breite*, im Objekt mit dem Namen *Box* beide auf den Wert 6 . Zur Kontrolle geben wir das in den Zeilen 51 und 52 aus. Genauso werden bei der Deklaration vom *Quadrat* die Werte für die *Hoehe* und die *Breite* des *Quadrat* durch den Konstruktor beide mit 6 initialisiert.

Ein Konstruktor hat der Definition nach denselben Namen wie die Klasse selbst. In diesem Fall heißen beide *Rechteck*. Die Definition von C++ verbietet eine Rückgabewert des Konstruktors. Eigentlich hat er einen vordefinierten Rückgabetyp, nämlich einen Zeiger auf das Objekt selbst, das soll uns aber bis sehr viel später in dieser Einführung nicht kümmern. Obwohl beiden Objekten durch den Konstruktor Werte zugewiesen werden, initialisieren wir sie in den Zeilen 60 und 61 mit neuen Werten. Da wir einen Konstruktor haben, der die Initialisierung übernimmt, sollten wir der Methode *Initialisiere()* vielleicht einen anderen Namen geben, das Konzept illustriert sie aber auch so.

Der Destruktor ist dem Konstruktor sehr ähnlich, nur wird er aufgerufen, wenn ein Objekt nicht mehr gültig ist. Du wirst sich erinnern, daß automatische Variablen eine beschränkte Lebensdauer haben, da ihre Tage gezählt sind, sobald das Programm den Block, innerhalb dessen sie deklariert wurden, verläßt. Gerade bevor dies geschieht wird der Destruktor aufgerufen, so einer existiert. Ein Destruktor hat denselben Namen wie die Klasse selbst mit einer Tilde davor. Ein Destruktor hat keinen Rückgabewert.

Wir deklarieren einen Destruktor in Zeile 12 und definieren ihn in den Zeilen 32 bis 36. In diesem Fall weist der Destruktor nur den Variablen jeweils den Wert 0 zu bevor ihr Speicherplatz freigegeben wird, es passiert also im Grunde gar nichts. Der Destruktor wurde nur hinzugefügt, um zu zeigen, wie er verwendet wird. Wenn innerhalb des Objektes dynamisch Speicherplatz beschafft wurde, sollte der Destruktor diesen freigeben, bevor die Zeiger darauf verloren gehen. Damit ist ihr Speicherplatz für weitere Verwendung verfügbar.

Es ist interessant zu wissen, daß ein Konstruktor für ein Objekt, das vor der Funktion *main()* deklariert wird, eine globale Variable also, auch vor der Funktion *main()* abgearbeitet wird. Genau so wird ein Destruktor für ein solches Objekt nach dem Abarbeiten der Funktion *main()* aufgerufen. Das hat zwar auf Dein Programm keinen wesentlichen Einfluß, ist aber doch recht interessant.

6.14 Modulare Gestaltung

Beispielprogramm: BOXEN1.CPP

BOXEN1.CPP gibt ein Beispiel, wie modulare Gestaltung nicht aussehen soll. Für ein sehr kleines Programm kann das ja ganz nett sein, es soll Dir aber zeigen, wie Du Dein Programm auf kleinere, leichter handhabbare Dateien aufteilst, wenn Du ein größeres Programm entwickelst oder Teil eines Teams bist, das solches tut. Die nächsten drei Beispielprogramme in diesem Kapitel werden zeigen, wie modulare Gestaltung ungefähr aussehen sollte.

Dieses Programm ähnelt dem letzten. Wir haben die Struktur *Mast* weggelassen und die Klasse heißt jetzt *Box*. Die Klasse wird in den Zeilen 4 bis 13 deklariert, in den Zeilen 16 bis 35 erfolgt ihre Implementation und wir verwenden die Klasse in den Zeilen 38 bis 52. Mit den Erklärungen zum letzten Programm sollte das Verständnis dieses Beispiels keine Schwierigkeiten bereiten.

6.15 inline-Elementfunktionen

Da die Methode in Zeile 11 sehr einfach ist (und weil wir damit einer weitere Neuerung in C++, die Du oft verwenden wirst, begegnen), ist die Implementation der Methode Teil der Deklaration. Wenn die Implementation in der Deklaration steht, erfolgt die Abarbeitung ohne Funktionsaufruf, was zu wesentlich schnellerem Code führt. In manchen Fällen führt das zu Code, der sowohl kleiner als auch schneller ist, ein weiteres Beispiel für die Effizienz von C++. Solche inline-Elementfunktionen haben dieselbe Effizienz wie Makros in C und ihnen ist bei kleinen Funktionen der Vorzug zu geben.

Kompiliere das Programm und führe es aus als Vorbereitung auf die nächsten drei Beispiele, Wiederholungen dieses Programmes in einer etwas anderen Form.

6.16 Die header-Datei der Klasse

Beispielprogramm: BOX.H

Wenn Du Dir die Datei BOX.H genau ansiehst, wirst Du feststellen, daß es sich dabei lediglich um die Klassendefinition handelt. Hier findet sich nichts über die Details der Implementation, die inline-Elementfunktion *HoleFlaeche()* natürlich ausgenommen. Wir finden in dieser Datei die komplette Definition, wie die Klasse verwendet wird, ohne mit der Implementation der Methoden in Berührung zu kommen. Drucke diese Datei aus, sie wird nützlich sein, wenn wir uns die nächsten beiden Dateien ansehen. Diese Datei enthält die Zeilen 4 bis 13 des vorigen Beispielprogrammes, BOXEN1.CPP. Es wird die header-Datei der Klasse genannt und kann weder kompiliert noch ausgeführt werden.

6.17 Die Implementation der Klasse

Beispielprogramm: BOX.CPP

Im Programm BOX.CPP findet sich die Implementation der Methoden, die wir in der header-Datei deklariert haben. In Zeile 2 importieren wir diese Datei mit den Prototypen der Methoden und den Definitionen der Variablen auch. Die Zeilen 16 bis 35 der Datei BOXEN1.CPP finden wir in dieser Datei, der Implementation der Methoden, die wir für die Klasse *Box* deklariert haben, wieder.

Zwar können wir diese Datei kompilieren, ausführen können wir sie aber nicht, da sie keine Funktion *main()* enthält, den Einstiegspunkt, den jedes ANSI-C oder C++ Programm benötigt. Nach dem Kompilieren wird der Objektcode im aktuellen Verzeichnis gespeichert und steht zur weiteren Verwendung durch andere Programme bereit. Das Ergebnis eines Kompiliervorganges wird üblicherweise Objektdatei genannt, da sie Objektcode beinhaltet. Das Wort Objekt in diesem Sinn hat aber nichts damit zu tun, was wir unter einem Objekt verstehen, wenn wir von objektorientiertem Programmieren sprechen. Es wird einfach die Bedeutung des Wortes „überladen". Die Sitte, das Resultat des Kompilierens als Objektdatei zu bezeichnen gibt es schon wesentlich länger als das Konzept des objektorientierten Programmierens.

Die Trennung von Definition und Implementation ist ein wesentlicher Schritt vorwärts in der Entwicklung von Software. Die Programmiererin benötigt lediglich die Datei mit der Definition, um die Klasse effektiv in seinen Programmen einsetzen zu können. Sie muß nicht wissen, wie die einzelnen Methoden realisiert sind. Wenn ihr die Implementation zur Verfügung stünde, könnte sie diese studieren und möglicherweise den einen oder anderen Trick finden, wie sie ihr Programm etwas effizienter gestalten kann, das würde aber zu nicht portierbarem Code und leicht zu Fehlern führen, wenn die Autorin der Klasse die Implementation modifiziert, ohne die Schnittstellen zu ändern. Der Sinn und Zweck von objektorientiertem Programmieren ist es, die Implementation so zu verstecken und abzusondern, daß sie nichts außerhalb ihres kleinen Bereiches innerhalb der Schnittstelle beeinflussen kann.

Du solltest die Implementation nun kompilieren. Wir werden das Resultat im nächsten Beispielprogramm verwenden.

6.18 Wir verwenden das Box-Objekt

Beispielprogramm: BOXEN2.CPP

Wirf einen Blick auf die Datei mit dem Namen BOXEN2.CPP und Du wirst sehen, daß wir in diesem Beispiel die zuvor definierte Klasse verwenden. Die letzten drei Dateien zusammen sind identisch mit dem Programm BOXEN1.CPP, das wir uns vorher angesehen haben.

In Zeile 3 importieren wir die header-Datei BOX.H, da wir die Definition der Klasse *Box* benötigen, um die drei Objekt deklarieren und ihre Methoden verwenden zu können. Du solltest gleich erkannt haben, daß es sich um nichts als eine Kopie des letzten Programmes handelt und auch genau dasselbe tut. Es gibt aber einen gewichtigen Unterschied zwischen BOXEN1.CPP und BOXEN2.CPP, wie wir gleich sehen werden.

Wir müssen hier eine wesentliche Unterscheidung machen. Wir rufen nicht einfach Funktionen auf und nennen daß halt jetzt das Senden von Nachrichten. Es gibt einen

wesentlichen Unterschied in den Aktionen selbst. Da die Daten eines Objektes eng an dieses gebunden sind gibt es keinen anderen Weg, an diese Daten zu kommen, als mit Hilfe der Methoden. Also senden wir eine Nachricht an das Objekt, die ihm mitteilt, daß es etwas mit den internen Daten anstellen soll. Wannimmer wir allerdings eine Funktion aufrufen, geben wir ihr die Daten, mit denen sie arbeiten soll, als Parameter mit. Eine Funktion hat keine eigenen Daten. Zugegeben, der Unterschied ist vielleicht ein geringer, wir werden aber die neue Terminologie verwenden und Dir sollte klar sein, daß eben doch ein Unterschied besteht.

Kompiliere dieses Programm und führe es aus. Wenn es ans linken geht, mußt Du dieses Programm mit dem Resultat der Kompilation der Klasse mit dem Namen *Boxen* linken. Die Datei heißt möglicherweise BOX.O. Dein Compiler hält sicherlich Informationen bereit, wie Du das anstellst.

Dieses Programm ist die erste Gelegenheit, eine Projekt-Datei oder `make` zu verwenden. Egal welchen C++ Compiler Du benutzt, es ist die Zeit wert, wenn Du Dir jetzt ansiehst, wie das funktioniert. Wir werden das im Laufe dieser Einführung noch des öfteren benötigen. Es liegt in der Natur von C++, die Programmiererin dazu anzuhalten, viele Dateien für ein Projekt zu verwenden und Du solltest Dich so bald wie möglich daran gewöhnen.

6.19 Ein Versteckspiel

Die drei Beispielprogramme, die wir uns zuletzt angesehen haben, illustrieren eine Methode des Zugriffsschutzes, die eine großen Einfluß auf die Qualität der Software haben kann, die für ein großes Projekt entwickelt wird. Da alle Information, die die Benutzerin einer Klasse benötigt, in der header-Datei zu finden ist, bekommt sie auch nicht mehr als diese zu sehen. Die Details der Implementation werden vor ihr versteckt, um sie davor abzuhalten, diese zu studieren und irgendeinen Kunstgriff zu tun, der zu unsauberem Code führt. Da sie nicht genau weiß, was diejenige, die die Klasse geschrieben hat, implementiert hat, muß sie sich an die Definitionen, die die header-Datei gibt, halten. Das kann, wie gesagt, einen großen Einfluß auf ein großes Projekt haben. Darüber hinaus wird die irrtümliche Änderung von Daten verhindert.

Ein weiterer Grund, die Implementation zu verstecken, ist ein ökonomischer. Der Hersteller Deines Compilers hat Dir viel Bibliotheksfunktionen zur Verfügung gestellt, nicht jedoch deren Quellcode, nur die Schnittstelle zu jeder Funktion. Du weißt, wie die Funktionen zum Dateizugriff zu verwenden sind, nicht jedoch deren Implementation und Du brauchst diese auch gar nicht. Es kann auch eine Bibliotheken-Industrie entstehen, die Programmiererinnen für eine Lizenzgebühr fix und fertig programmierte und geteste-te Bibliotheken von hoher Qualität zur Verfügung stellt. Da die Programmiererin nur die Definition der Schnittstelle benötigt, erhält sie diese zusammen mit dem Objektcode (dem Ergebnis des Kompiliervorganges) der Klasse und kann diese nach Herzenslust verwenden. Der Quellcode der Autorin der Klasse ist vor unabsichtlichen oder absichtlichen Modifikationen geschützt und die Autorin behält die volle Kontrolle über ihn.

Es ist wichtig, daß Du die Prinzipien dieses Kapitels verstanden hast, bevor Du zum

nächsten weitergehst. Wenn Du Dir über irgendetwas noch nicht vollkommen im Klaren bist, wirf noch einmal einen (genaueren) Blick darauf. Etwas sei noch erwähnt, falls Du es noch nicht selbst bemerkt hast: um eine Klasse effektiv zu verwenden, bedarf es einigen Weitblicks.

6.20 Abstrakte Datentypen

Wir haben den sogenannten abstrakten Datentyp am Beginn dieses Kapitels erwähnt und noch einmal etwas später. Jetzt ist es an der Zeit, etwas genauer darauf einzugehen. Ein abstrakter Datentyp ist eine Gruppe von Daten, von denen jede eine Reihe von Werten speichern kann, und einigen Methoden oder Funktionen, die mit diesen Daten arbeiten. Die Daten sind vor äußerem Zugriff geschützt. Da die Daten eine Beziehung zueinander haben, bilden Sie eine zusammenhängende Gruppe, deren Mitglieder miteinander sehr viel zu tun haben, aber wenig mit der „Außenwelt“.

Die Methoden allerdings haben eine Verbindung zu dieser Außenwelt über die Schnittstellen, die Zahl der Kontakte ist aber beschränkt und die Verbindung mit dem übrigen Programm ist eine lose. Das Objekt ist lose mit der Außenwelt verbunden. Durch die enge Verbindung untereinander einerseits und die lose Bindung zur Außenwelt andererseits wird die Wartung von Software erleichtert. Das ist vielleicht der größte Vorteil objektorientierten Programmierens.

Es macht Dir vielleicht Sorgen, daß die Programmiererin die privaten Variablen zwar außerhalb der Klasse nicht verwenden kann, sie aber doch zu Gesicht bekommt und so eventuell ziemlich genau abschätzen kann, wie die Klasse implementiert ist. Die Variablen hätten ohne weiteres komplett in einer anderen Datei versteckt werden können. Da die Autoren von C++ aber auf Effizienz großen Wert gelegt haben, blieben die Variablen in der Klassendefinition, wo sie zwar sichtbar, aber nicht brauchbar sind.

6.21 Freundfunktionen

Eine Funktion außerhalb der Klasse kann als **friend**-Funktion der Klasse definiert werden und so Zugriff auf die privaten Elemente der Klasse erhalten. Das öffnet das Schutzschild rund um die Daten der Klasse ein wenig, sollte also nur sehr selten angewendet werden. Es gibt Fälle, wo ein Programm durch die Verwendung solcher Freundfuntkionen leichter zu verstehen ist und diese einen kontrollierten Zugriff auf die Daten erlauben. Wir werden die Verwendung von Freundfunktionen in einigen Beispielprogrammen später in dieser Einführung zeigen. Wir haben sie hier aber der Vollständigkeit halber erwähnt. Du kannst einer einzelnen Funktion diesen Status verleihen, aber auch Elementen anderer Klassen oder gar ganzen Klassen. Weder ein Konstruktor noch ein Destruktor kann allerdings aus verständlichen Gründen eine Freundfuntkion sein.

6.22 Die Struktur in C++

Strukturen (Schlüsselwort **struct**) sind in C++ nach wie vor enthalten und verhalten sich genauso wie in ANSI-C. Du kannst Deiner Struktur Methoden geben, die mit den Daten genauso arbeiten wie in einer Klasse, aber Methoden wie Daten sind am Anfang einer Struktur automatisch öffentlich. Selbstverständlich kannst Du die Daten oder Methoden als privat deklarieren. Eine Struktur sollte aber nur für Konstrukte verwendet werden, die auch wirklich Strukturen sind. Wenn Du auch nur einfachste Objekte konstruierst, solltest Du eine Klasse dafür verwenden.

6.23 Eine sehr praktische Klasse

Die Beispiele, die wir in diesem Kapitel für Zugriffsschutz gegeben haben, waren all sehr einfach, um zu zeigen, wie es grundsätzlich funktioniert. Da es aber sicherlich nicht schadet, auch ein etwas größeres Beispiel gesehen zu haben, sehen wir uns als nächstes die Klasse *Datum* an. Diese Klasse ist ein komplettes und brauchbares Beispiel. Du kannst sie in jedem Programm verwenden, um das aktuelle Datum in einem von vier vordefinierten Formaten auszugeben. Du kannst sie auch verwenden, um ein Datum zu speichern und für die Ausgabe zu formatieren.

Beispielprogramm: DATUM.H

Die Datei mit dem Namen DATUM.H ist die header-Datei für unsere Klasse *Datum*. Sie ist reichhaltigst kommentiert, sodaß nicht viel zu sagen bleibt. Wenn Du im Prinzip verstanden hast, was wir in diesem Kapitel behandelt haben, solltest Du auch mit dieser Klasse keine Probleme haben. Neu für Dich ist das geschützte Wort **protected** in Zeile 13. Wir werden dieses Wort ein paar Kapitel weiter hinten definieren. Bis dahin nimm an, daß es dasselbe meint wie **private**, das genügt einstweilen fürs Verständnis dieses Beispiels. Den Code in den Zeilen 8,9 und 57 werden wir gleich erklären. Mißachte diese Zeilen einstweilen einfach. Auch dem Schlüsselwort **static** wie es in den Zeilen 18 und 19 verwendet wird, werden wir uns später widmen. Diese neuen Konstrukte haben wir deshalb eingefügt, weil wir diese Klasse später beim Studium der Vererbung verwenden wollen.

Du solltest Dich so lange mit der header-Datei befassen, bis Du bis auf die oben genannten neuen Konstrukte alles verstanden hast. Geh dann weiter zur Implementation der Klasse.

Beispielprogramm: DATUM.CPP

Die Datei mit dem Namen DATUM.CPP beinhaltet die Implementation der *Datum* Klasse. Auch hier findet sich nichts Außergewöhnliches oder Schwieriges. Sie verwendet einfache Konstrukte, um das Datum zu speichern und in einer brauchbaren Form auszugeben. Du sollten auch diesen Code so lange studieren, bis Du ihn komplett verstanden hast, bevor Du zum nächsten Beispiel weitergehst, wo wir die *Datum* Klasse in einem Programm verwenden werden.

Die Implementation des Konstruktors in den Zeilen 14 bis 25 verwendet Systemaufrufe, um das aktuelle Datum zu bekommen. Diese Aufrufe werden nur kompilieren, wenn

sie 16-Bit DOS verwenden, da sie nicht kompatibel sind. Du kannst diese Zeilen so abändern, daß sie die Aufrufe Deines Systems verwenden oder den Elementvariablen einfach beliebige Werte zuweisen. Sinn und Zweck dieses Codes ist es, die Verwendung von Zugriffsschutz und Konstruktoren zu zu illustrieren, nicht das Auslesen der Uhr und des Kalenders Deines Computers.

Beispielprogramm: VERWDAT.CPP

Im einfachen Programm VERWDAT.CPP verwenden wir die *Datum* Klasse, um das aktuelle und ein weiteres Datum am Bildschirm auszugeben. Du solltest kein Problem mit dem Verständnis dieses Programmes haben, also wollen wir auch keine weiteren Worte darüber verlieren.

Du solltest Dir genug Zeit nehmen, diese drei Dateien zu verstehen, da sie den Ausgangspunkt einer Anschauungsreise in den nächsten Kapiteln bilden. Wir werden diese Klasse in Verbindung mit anderen verwenden, um einfache und vielfache Vererbung zu demonstrieren. Auch wenn Du nicht jede Einzelheit des Programmes verstehst, solltest Du doch versuchen, die Struktur und die wesentlichen Konzepte zu verstehen.

Wir werden die Diskussion des Zugriffsschutzes im nächsten Kapitel fortsetzen.

6.24 Programmieraufgaben

1. Füge zu KLAS.CPP eine Methode hinzu, die das Quadrat des gespeicherten Wertes zurückgibt. Erweitere das Hauptprogramm um Code, der den quadrierten Wert liest und ausgibt.

2. Erweitere KLAS.CPP weiters um einen Konstruktor, der den gespeicherten Wert mit 10 initialisiert. Schreibe im Hauptprogramm einige Zeilen Code, die die Werte unmittelbar nach der Definition des Objektes ausgeben.

3. Füge dem Konstruktor von Rechteck in KONSMAST.CPP eine Bildschirmausgabe hinzu und eine weitere zum Destruktor, um zu zeigen, daß diese wirklich aufgerufen werden.

4. Schreibe eine etwas anspruchsvollere Anwendung für die Klasse Datum, die wir am Ende des Kapitels vorgestellt haben.

5. Schreibe eine Klasse mit dem Namen Name, die ähnlich der Klasse Datum jeden beliebigen Namen in drei Teilen speichern kann und den vollständigen Namen dann in einer Reihe von Formaten ausgeben kann, wie zum Beispiel
   ```
   Heinz Heinzl Tschabitscher
   H. H. Tschabitscher
   Tschabitscher, Heinz Heinzl
   ```
   oder was immer Dir zusagt.

7 Mehr über Zugriffsschutz

7.1 Wozu das ganze?

Wir haben uns diese Frage schon früher gestellt, können aber jetzt, da wir ein wenig Erfahrung gesammelt haben, eine wesentlich bessere Antwort geben. Zugriffsschutz schützt die Daten – wie der Name schon sagt – vor irrtümlicher Änderung, und Konstruktoren stellen sicher, daß alle Elemente initialisiert werden. Beide verhindern Fehler, die wir gerne machen. Während wir eine Klasse schreiben, beschäftigen wir uns nur mit den Interna der Klasse, später, wenn wir die Klasse dann wirklich anwenden, brauchen wir uns um die interne Struktur der Klasse oder ihrer Methoden nicht mehr zu kümmern und können uns auf die Lösung des Problems konzentrieren. Natürlich gibt es noch viel über die Verwendung und die Vorteile von Klassen zu sagen und zu lernen, weshalb wir uns auch gleich einigen neuen Aspekten widmen wollen.

Ziel dieses Kapitels ist es, zu illustrieren, wie einige der traditionellen Aspekte von C/C++ mit Klassen und Objekten verwendet werden. Wir werden über Zeiger auf ein Objekt und Zeiger innerhalb eines Objektes sprechen, von Arrays, die in einem Objekt liegen und Arrays von Objekten. Da Objekte nur ein weiteres „Datenkonstrukt" in C++ sind, ist all das vorher Genannte möglich und kann verwendet werden, wenn der Bedarf besteht.

Um das ganze zumindest ein bißchen systematisch zu gestalten, werden wir das Programm BOXEN1.CPP als Ausgangspunkt nehmen und mit jedem Beispielprogramm einige wenige neue Konstrukte hinzufügen. Du erinnerst Dich sicherlich, daß es sich bei BOXEN1.CPP um ein sehr einfaches Programm gehandelt hat, mit der Definition der Klasse, der Implementation derselben und dem Hauptprogramm in einer einzelnen Datei. Wir nehmen diese Datei als Basis, weil wir nach und nach alle Teile des Programmes ändern werden und es daher anschaulicher ist, all das in einer einzigen Datei zu haben. Es sollte aber klar sein, daß es richtiger wäre, die Teile auf drei verschiedene Dateien aufzuteilen, wie wir es mit BOX.H, BOX.CPP und BOXEN2.CPP im vorigen Kapitel gezeigt haben. Das erlaubt es der Autorin von *Box*, der Programmiererin nur die Schnittstelle, BOX.H, zur Verfügung zu stellen. Wie wir schon des öfteren festgestellt haben, erscheint es eigentümlich, so kleine Programme auf drei Dateien aufzuteilen und das ist es ja auch. Das letzte Kapitel dieser Einführung wird ein Programm vorstellen, das diese Aufteilung auch rechtfertigt.

7.2 Ein Array von Objekten

Beispielprogramm: OBJARRAY.CPP

Die Datei OBJARRAY.CPP ist unser erstes Beispiel eines Arrays von Objekten. Dieses Beispielprogramm ist nahezu identisch mit dem Programm BOXEN1.CPP bis zur Zeile 45, wo wir einen Array von 4 Boxen definieren.

Wenn Du daran denkst, welche Funktion ein Konstruktor hat, erinnerst Du Dich sicherlich, daß jedes der *Box* Objekte mit den im Konstruktor angegebenen Werten initialisiert wird, da der Konstruktor für jedes einzelne Objekt bei der Definition aufgerufen wird. Um einen Array von Objekten definieren zu können, muß ein Konstruktor ohne Parameter für dieses Objekt existieren. (Wir haben noch nicht gezeigt, wie ein Konstruktor mit Initialisierungsparametern aussieht, werden das aber im nächsten Programm tun.) Das ist eine Frage der Effizienz, da es möglicherweise eine Fehler ist, alle Elemente eines Arrays von Objekten mit denselben Werten zu initialisieren. Wir werden das Ergebnis, das der Aufruf des Konstruktors liefert, sehen, wenn wir das Programm kompilieren und ausführen.

In Zeile 50 starten wir eine **for**-Schleife, die mit 1 anstelle der für das Element eines Arrays typischen 0 beginnt. Damit verwendet das erste Objekt, *Gruppe[0]*, die Werte, die ihm bei der Ausführung des Konstruktors zugewiesen wurden. Das Senden einer Nachricht an eines der Objekte erfolgt genauso wie bei jedem anderen Objekt auch. Wir verwenden dazu den Namen des Arrays, gefolgt vom Index des Objektes in eckigen Klammern, wie in Zeile 51 gezeigt. Die andere Methode rufen wir in den Zeilen 58 und 59 auf, wo die Fläche der vier Boxen der *Gruppe* am Bildschirm ausgegeben wird.

7.3 Deklaration und Definition einer Variabe

Zu Illustrationszwecken haben wir in Zeile 8 noch eine Variable, *ExtraDaten* hinzugefügt. Da wir das Schlüsselwort **static** verwenden, handelt es sich dabei um eine externe Variable und es existiert nur eine Variable für alle Objekte der Klasse. Alle sieben Objekte dieser Klasse teilen sich also diese eine Variable.

Die Variable wird hier lediglich deklariert, was heißt, daß sie einmal existieren wird, sie wird aber noch nicht definiert. Eine Deklaration sagt dem Compiler, daß irgendwo im Programm eine Variable existieren wird und gibt ihr einen Namen, aber es ist die Definition, die im Speicher einen Platz für die Variable bereitstellt. Per Definitionem kann eine statische Variable im Kopf einer Klasse deklariert, nicht aber definiert werden. Sie wird also in der Implementation definiert, in unserem Fall in Zeile 17 und kann dann innerhalb der Klasse verwendet werden.

Abbildung 7.1 versinnbildlicht einige der Variablen. Die Objekte mit den Namen *Grosz*, *Gruppe[0]*, *Gruppe[1]* und *Gruppe[2]* sind auf dem Bild nicht zu sehen, aber auch sie teilen sich mit den anderen die Variable *ExtraDaten*. Sie werden im Bild nicht gezeigt, um es ein wenig übersichtlicher und klarer zu gestalten. Jedes Objekt hat seine eigene *Laenge* und *Breite*, da diese nicht als **static** deklariert sind.

Die Zeile 24 des Konstruktors setzt die globale Variable mit jedem Objekt, das wir erzeugen, auf 1. Es ist nur eine Zuweisung vonnöten, die anderen sechs sind unnötig. Es ist im Allgemeinen keine gute Idee, einer statischen Variable in einem Konstruktor einen Wert zuzuweisen, in diesem Fall aber illustriert das eine Variable, die als **static**

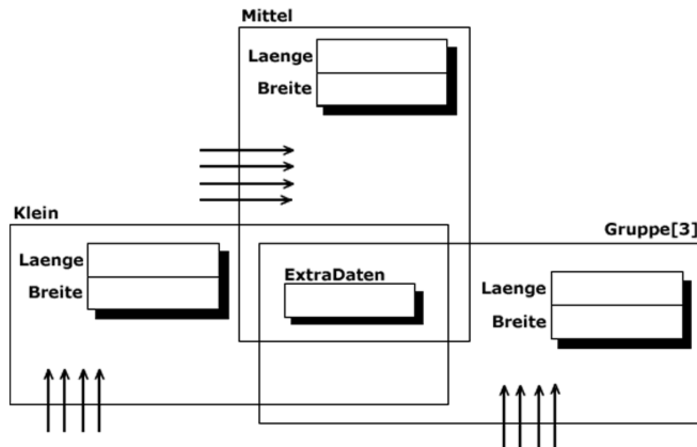

Abbildung 7.1: Variablen in OBJARRAY.CPP

deklariert ist. Um zu zeigen, das es wirklich nur eine solche Variable gibt, die sich alle Objekte der Klasse teilen, inkrementiert die Methode, die sie liest, die Variable auch. Jedes Mal, wenn wir sie in den Zeilen 61 bis 65 lesen, wird sie um 1 erweitert, und das Ergebnis beweist uns, daß es sich bei dieser Variable wirklich um ein Einzelstück handelt. Du wirst auch bemerken, daß die Methode mit dem Namen *HoleExtraDaten* innerhalb der Klassendeklaration definiert und daher als inline-Code behandelt wird.

Du wirst Dich an die beiden statischen Variablen, die wir in den Zeilen 18 und 19 der Datei DATUM.H des Kapitels 6 deklariert haben, erinnern. In Zeile 9 und 10 von DATUM.CPP haben wir sie definiert und gesagt, daß wir uns ihnen später widmen werden. Deklaration und Definition dieser Variablen sind ein gutes Beispiel dafür, wo diese Konstrukte für statische Variablen in Deinem Code stehen sollten.

Wenn Du sicher bist, daß Du dieses Beispielprogramm und vor allem Funktion und Verwendung von statischen Variablen verstanden hast, kompiliere es und führe es aus.

7.4 Eine Zeichenkette innerhalb eines Objektes

Beispielprogramm: OBJZKETT.CPP

Das Programm OBJZKETT.CPP ist unser erstes Beispiel für eine Zeichenkette innerhalb eines Objektes. Eigentlich handelt es sich dabei ja nicht um eine Zeichenkette selbst, sondern um einen Zeiger, die beiden sind aber so eng verbunden, daß wir sie zugleich abhandeln können.

In Zeile 8 deklarieren wir einen Zeiger auf eine Variable vom Typ **char** mit dem Namen *TextZeile*. Der Konstruktor nimmt einen Parameter, einen Zeiger auf eine Zeichenkette, entgegen. Die Zeichenkette wird im Konstruktor unter dem Namen *TextZeile* kopiert. Wir hätten die Variable *TextZeile* in der Klasse als Array deklarieren können, um dann mithilfe von *strcpy()* die Zeichenkette in diesen Array zu kopieren und alles hätte genauso gut funktioniert. Das soll aber Dir als Aufgabe am Ende des Kapitels vorbehalten bleiben [hehe]. Wir sollten vielleicht noch festhalten, daß wir uns keineswegs auf einen einzigen

Parameter für einen Konstruktor beschränken müssen. Wie wir noch zeigen werden, kann ein Konstruktor (fast) jede erdenkliche Anzahl von Parametern haben.

Wenn wir also dieses Mal unsere drei Boxen deklarieren geben wir jeder eine konstante Zeichenkette mit auf den Weg zum Konstruktor, damit dieser dem internen Zeiger auf eine Zeichenkette auch etwas zuweisen kann. Beim Aufruf von *HoleFlaeche()* in den Zeilen 50 bis 54 wird eine Nachricht ausgegeben und die Fläche zurückgegeben. Es wäre sicherlich sinnvoller, diese beiden Aktionen auf separate Methoden aufzuteilen, da ja kein ersichtlicher Zusammenhang zwischen ihnen besteht, wir haben es aber so gemacht wie wir es gemacht haben, um zu zeigen, daß es geht. Was wir also wirklich demonstrieren, ist, daß eine Methode einen Nebeneffekt (Ausgabe der Nachricht) haben kann und einen Wert zurückgibt, die Fläche der Box. We wir aber in Kapitel 5 beim Beispiel STANDARD.CPP gesehen haben, ist die Abfolge der Abarbeitung bisweilen eigenartig was, uns dazu bewogen hat, die Codezeile auf zwei aufzuteilen.

Hast Du dieses Programm verstanden, kompiliere es bitte und führe es aus.

7.5 Ein Objekt mit einem internen Zeiger

Beispielprogramm: OBJINTZG.CPP

Das Programm OBJINTZG.CPP ist unser erstes Beispiel für ein Programm mit eingebettetem Zeiger, mit dem wir dynamisch Speicher beschaffen.

In Zeile 8 deklarieren wir einen Zeiger auf eine Variable vom Typ **int**, aber es ist eben nur ein Zeiger, kein Speicher ist ihm tatsächlich noch zugewiesen. Deshalb stellt der Konstruktor in Zeile 22 eine Variable vom Typ **int** im Speicher bereit, die wir mit diesem Zeiger verwenden. Es ist Dir sicherlich klar, daß jedes der drei Objekte, die wir in Zeile 46 definieren, seinen eigenen Zeiger hat, der auf seine eigene Adresse im Speicher zeigt. Für jedes Objekt steht eine eigene dynamisch beschaffte Variable im Speicher zur Verfügung. Alle diese Variablen haben aber zunächst den Wert 112, da wir in Zeile 23 diesen Wert in jeder der drei Variablen speichern, je einmal pro Konstruktoraufruf.

In einem so kleinen Programm müssen wir uns schon sehr anstrengen, den verfügbaren Speicher voll auszuschöpfen, und mit dem, was wir tun, wird uns das aller Wahrscheinlichkeit nach auch nicht gelingen. Deshalb führen wir keinen Test durch, ob der Speicher, den wir anfordern, auch tatsächlich verfügbar ist. In einem „richtigen“ Programm solltest Du aber testen, ob der Wert des zurückgegebenen Zeigers nicht NULL ist, um sicherzustellen, daß die Daten auch gespeichert werden können.

Die Methode mit dem Namen *Setze()* erwartet drei Parameter, von denen letzterer den Wert der neuen, dynamisch beschafften Variable setzt. Wir senden Nachrichten, diese Methode auszuführen, an zwei Objekte, die kleine und die große Box. Die Standardwerte der mittleren Box bleiben unangetastet.

Wir zeigen die drei Flächen gefolgt von den drei in der dynamisch beschafften Variable gespeicherten Werten aus und haben schlußendlich ein Programm, das einen Destruktor braucht, um vollkommen richtig zu funktionieren. Wenn wir einfach den Gültigkeitsbereich der Objekte verlassen, wie wir es tun, wenn wir die Funktion *main()* verlassen, bleiben uns die drei dynamisch beschafften Variablen im Speicher erhalten, nur zeigt

nichts mehr auf sie. Sie sind nicht mehr adressierbar und deshalb nichts als verschwendeter Speicherplatz. Aus diesem Grund benötigen wir einen Destruktor, der die Variablen löscht (mittels **delete**), auf die die Zeiger mit dem bedeutungsschwangeren Namen *Zeiger* zeigen, je eine pro Aufruf. Bei einem solchen Aufruf (der wie wir wissen automatisch erfolgt, wenn ein Objekt zu existieren aufhört) weisen die Zeilen 38 und 39 den Variablen, die automatisch gelöscht werden, den Wert 0 zu. Zwar ist dies vollkommen sinnlos, möglich ist es aber doch.

In unserem speziellen Beispiel werden alle Variablen (auch die dynamisch beschafften) automatisch gelöscht, da wir zum Betriebssystem zurückkehren, das diese Aufgabe für uns übernimmt, wenn es das komplette Programm aus dem Speicher entfernt. Trotzdem zeugt es von gutem Programmierstil, aufzuräumen, wenn man mit dem Spielen fertig ist (dynamisch bereitgestellten Speicher nicht mehr benötigt).

Auf ein anderes Konstrukt sollten wir auch noch einmal kurz eingehen, nämlich auf die Methoden in den Zeilen 12 und 13, die wir **inline** implementieren. Wie wir in Kapitel 6 festgestellt haben, können Funktionen mit dieser Technik implementiert werden, wenn erstes und oberstes Ziel Geschwindigkeit ist, da die Funktionen im Code selbst stehen und nicht mit einem Funktionsaufruf verbunden sind. Da der Code für diese Methoden in der Deklaration steht, wird der Compiler ihn inline, also im Code, wo der Funktionsaufruf steht einfach an dessen Statt einsetzen, eine weitere Implementation des Codes der Methode ist nicht notwendig. Es handelt sich dabei allerdings nur um einen Vorschlag an den Compiler, das zu tun, er muß sich keineswegs daran halten und kann die Methode (oder Funktion) als solche implementieren.

Bedenke aber, daß wir am Verstecken unseres Codes interessiert sind, was durch die im vorigen Absatz diskutierte Vorgangsweise natürlich untergraben wird, da wir den kompletten Code offen zur Schau stellen. Oft aber wirst Du an der Geschwindigkeit mehr interessiert sein als am Verstecken von trivialem Code, denn nur solcher empfiehlt sich wirklich, **inline** implementiert zu werden. Kompiliere das Programm und führe es aus.

7.6 Ein dynamisch beschafftes Objekt

Beispielprogramm: OBJDYNAM.CPP

Im Beispielprogramm OBJDYNAM.CPP werfen wir zum ersten Mal einen Blick auf ein dynamisch beschafftes Objekt. Dieses unterscheidet sich nicht von einem „normalen“ dynamisch bereitgestellten Objekt, ein Beispiel ist aber immer recht hilfreich.

In Zeile 40 definieren wir einen Zeiger auf ein Objekt vom Typ (der Klasse) *Box* und da es sich nur um einen Zeiger ohne Objekt, auf das er zeigen könnte, handelt, stellen wir in Zeile 45 dynamisch ein solches bereit. Wenn wir das Objekt in Zeile 45 kreieren, wird der Konstruktor automatisch aufgerufen und weist den zwei internen Variablen Werte zu. Der Konstruktor wird nicht aufgerufen, wenn wir den Zeiger definieren, da es ja zu diesem Zeitpunkt nichts gibt, das initialisiert werden könnte. Er wird erst aufgerufen, wenn der Speicher für das Objekt bereitgestellt wird.

Auf die Elemente des Objektes greifen wir so zu wie auch auf die Elemente einer Struk-

tur, mithilfe des Zeigeroperators, wie etwa in den Zeilen 51 bis 53 illustriert. Natürlich kannst Du den Zeiger auch ohne den Pfeil zu verwenden dereferenzieren, wie zum Beispiel (*point).set(12, 12); anstatt des Code in Zeile 52, die Pfeilnotation ist aber einfacher und universeller, sollte also verwendet werden. Schließlich löschen wir das Objekt in Zeile 55 und das Programm ist abgearbeitet. Gäbe es für diese Klasse einen Destruktor, würde er automatisch im Zuge der **delete** Anweisung aufgerufen werden, um aufzuräumen, bevor das Objekt im Datennirvana eins mit diesem wird. (??)

Du hast mittlerweile wahrscheinlich bemerkt, daß die Verwendung von Objekten sich von der Anwendung von Strukturen nicht wesentlich unterscheidet. Kompiliere dieses Programm und führe es aus, wenn Du verstanden hast, worum es sich dreht.

7.7 Ein Objekt mit einem Zeiger auf ein anderes Objekt

Beispielprogramm: OBJLISTE.CPP

Das Programm OBJLISTE.CPP zeigt ein Objekt mit einer internen Referenz auf ein anderes Objekt derselben Klasse. Das ist die normale Struktur einer einfach verbundenen Liste und wir werden die Verwendung in diesem Beispiel sehr einfach halten.

Der Konstruktor beinhaltet in Zeile 22 die Initialisierung des Zeigers mit einer Zuweisung des Wertes NULL. Das solltest Du in allen Deinen Programmen so halten, laß keinen Zeiger ins Nichts zeigen, sondern initialisiere alle Zeiger auf irgendeinen Wert. Wenn dies im Konstruktor geschieht, hast Du gewährleistet, daß die Zeiger eines jeden Objektes automatisch initialisiert werden. Es ist dann unmöglich, es zu vergessen.

In den Zeilen 13 und 14 deklarieren wir zwei weitere Methoden, von denen die in Zeile 14 ein Konstrukt beinhaltet, das wir in unserer Einführung noch nicht betrachtet haben. Diese Methode gibt einen Zeiger auf ein Objekt der Klasse *Box* zurück. Wie Du weißt, kannst Du in Standard-C eine Zeiger auf eine Struktur zurückgeben, und was wir hier haben, ist die Umsetzung dessen in C++. Die Implementation in den Zeilen 49 bis 52 gibt den Zeiger, der als Elementvariable im Objekt selbst gespeichert ist, zurück. Auf die Verwendung werden wir eingehen, wenn wir das eigentliche Programm betrachten.

Im Hauptprogramm definieren wir einen weiteren Zeiger mit dem Namen *BoxZeiger*, um ihn später zu verwenden. In Zeile 67 lassen wir den internen Zeiger des Objektes der Klasse *Box* mit dem Namen Klein auf das Objekt *Mittel* zeigen. Zeile 68 läßt den internen Zeiger des Objektes *Mittel* auf das Objekt *Grosz* zeigen. Wir haben eine verbundene Liste mit drei Elementen erzeugt. In Zeile 70 veranlassen wir, daß der Zeiger des Hauptprogrammes auf das Objekt *Klein* zeigt. In Zeile 71 verwenden wir diesen Zeiger dann, um auf das Objekt *Klein* zu zeigen und lassen ihn auf das Objekt zeigen, auf das der Zeiger des Objektes *Klein* zeigt, also die Adresse des Objektes *Mittel*. Wir sind demnach von einem Objekt zum nächsten fortgeschritten, indem wir an eines der Objekte eine Nachricht gesendet haben. Würden wir die Zeile 71 genau so wie sie hier steht noch einmal wiederholen, würde das den Zeiger des Hauptprogrammes auf das Objekt *Grosz* zeigen lassen. Wir hätten dann die gesamte Liste mit ihren drei Elementen durchwandert. Abbildung 7.2 zeigt den Speicherinhalt nach Abarbeiten der Zeile 70. Beachte, daß nur ein Fragment jedes Objektes dargestellt ist, um die Zeichnung einfach zu halten.

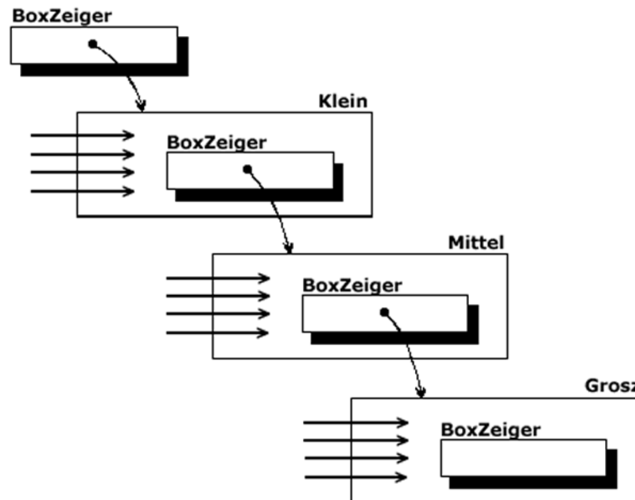

Abbildung 7.2: Speicherinhalt nach Zeile 70

7.8 Noch ein neues Schlüsselwort: this

In C++ ist ein neues Schlüsselwort verfügbar: **this**. Das Wort **this** ist in jedem Objekt definiert als Zeiger auf das Objekt selbst, in dem dieser enthalten ist. Er ist einfach definiert als:

KlassenName *this;

und wird so initialisiert, daß es auf das Objekt zeigt, für das die Elementfunktion aufgerufen wird. Dieser Zeiger ist natürlich am nützlichsten im Umgang mit anderen Zeigern und da besonders in einer verbundenen Liste wenn Du eine Referenz auf das Element, das Du der Liste gerade hinzufügst, brauchst. Für diesen Zweck ist das Schlüsselwort **this** verfügbar und kann in jedem Objekt verwendet werden. Eigentlich wäre die „richtige“ Art, eine Variable in einer Liste zu verwenden, mittels *this-¿VariablenName*, der Compiler nimmt aber implizit an, daß wir den Zeiger verwenden und so können wir das ganze vereinfachen, indem wir den Zeiger weglassen. Die Verwendung des Schlüsselwortes **this** ist in keinem Beispielprogramm illustriert, wird aber in einem der größeren Programme dieser Einführung vorkommen.

Du solltest Dir dieses Programm ansehen, bis Du das sichere Gefühl hast, es verstanden zu haben und es dann kompilieren und ausführen, als Vorbereitung auf unser nächstes Beispiel.

7.9 Eine verbundene Liste von Objekten

Beispielprogramm: OBJVERB.CPP

Unser nächstes Programm in diesem Kapitel heißt OBJVERB.CPP und ist ein komplettes Beispiel einer verbundenen Liste in objektorientierter Schreibweise.

Dieses Beispielprogramm ist dem letzten sehr ähnlich. In der Tat ist es mit diesem

identisch, bis wir zur Funktion *main()* kommen. Du erinnerst Dich, wir haben im letzten Programm nur über die beiden Methoden *ZeigeAufNaechste()* und *HoleNaechste()* Zugriff auf die internen Zeiger der Objekte gehabt. Diese Methoden finden sich in den Zeilen 42 bis 52 des Programmes, dem wir uns jetzt widmen wollen. Wir werden sie verwenden, um unsere Liste zu erstellen und sie dann durchzugehen, um die Liste am Bildschirm auszugeben. Schließlich werden wir die komplette Liste löschen, um den Speicher freizugeben.

In den Zeilen 57 bis 59 definieren wir drei Zeiger. Der Zeiger mit dem Namen *Start* wird immer auf das erste Element der Liste zeigen, *Temp* hingegen wird durch die Liste wandern, wenn wir sie erstellen. Den Zeiger mit dem Namen *BoxZeiger* werden wir zum Erzeugen der Objekte verwenden. Die Schleife in den Zeilen 62 bis 75 erzeugt unsere Liste. Zeile 64 erzeugt dynamisch ein neues Objekt der Klasse *Box* und in Zeile 65 füttern wir dieses Objekt zur Illustration mit sinnlosen Daten. Ist das Objekt, das wir gerade erzeugt haben, das erste in der Liste, lassen wir den Zeiger *Start* darauf zeigen. Gibt es allerdings schon Elemente in der Liste, lassen wir das letzte Element (repräsentiert durch den Zeiger *Temp*) auf das neue Zeigen. In jedem Fall weisen wir dem Zeiger *Temp* das letzte Element in der Liste zu, damit wir noch weitere anfügen können, sollte das gewünscht werden.

In Zeile 78 lassen wir den Zeiger mit dem Namen *Temp* auf das erste Element zeigen. Wir verwenden diesen Zeiger, um die gesamte Liste zu durchschreiten, wenn er sich selbst durch die Zuweisung in Zeile 82 vorarbeitet. Wenn *Temp* den Wert NULL hat, den er vom letzten Element in der Liste erhält, haben wir die komplette Liste abgearbeitet.

Schlussendlich löschen wir die gesamte Liste. Wir beginnen beim ersten Element und löschen bei jedem Schleifendurchlauf in den Zeilen 87 bis 92 ein Element.

Ein Studium des Programmes wird zeigen, daß es tatsächlich eine verbundene Liste mit zehn Elementen erzeugt, wobei jedes Element ein Objekt der Klasse *Box* ist. Die Länge der Liste ist dadurch eingeschränkt, wie viele Elemente wir am Bildschirm ausgeben wollen, sie könnte aber eine Größe von vielen tausend Elementen erreichen, vorausgesetzt, wir haben genug Speicher, um alle diese zu speichern.

Auch hier überprüfen wir die dynamische Speicherverwaltung nicht, wie es in einem korrekt geschriebenen Programm eigentlich geschehen sollte. Kompiliere das Beispielprogramm und führe es aus.

7.10 Das Schachteln von Objekten

Beispielprogramm: SCHACHT.CPP

Das Programm SCHACHT.CPP zeigt uns das Schachteln von Klassen, was zu einem Schachteln von Objekten wird. Verschachtelte Objekte können zum Beispiel ganz einfach durch Deinen Computer versinnbildlicht werden. Der Computer selbst besteht aus vielen Teilen, die zusammen, aber doch jeder für sich arbeiten, wie etwa eine Maus, ein Laufwerk oder ein Transformator. Der Computer ist aus diesen so verschiedenen Teilen zusammengesetzt und es ist nur sinnvoll, die Maus getrennt vom Transformator zu betrachten, einfach weil sie so verschieden sind. Eine Klasse für Computer könnte also

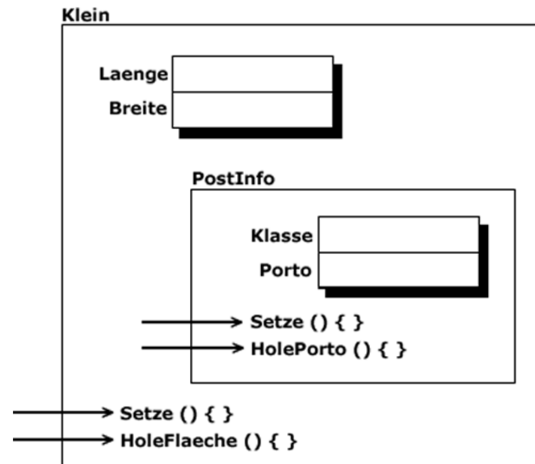

Abbildung 7.3: Geschachtelte Objekte

aus sehr verschiedenen Objekten bestehen, indem die einzelnen Objekte in der Klasse enthalten sind.

Wenn wir aber, zum Beispiel, die Charakteristika von Laufwerken betrachten wollen, werden wir uns zunächst ansehen, was ein Laufwerk generell ausmacht, dann die Eigenheiten einer Festplatte und jene eines CD-ROM Laufwerkes. Das führt zu einer gewissen Vererbung, da viele Daten über die beiden Laufwerke generalisiert und dem Typ „Laufwerk" zugewiesen werden können. Wir werden die Vererbung in den nächsten drei Kapiteln noch eingehend studieren, nun aber wollen wir uns geschachtelten Klassen widmen.

Das Beispielprogramm enthält eine Klasse mit dem Namen VariBox, die wiederum ein Objekt einer anderen Klasse mit dem Namen *PostInfo* enthält, das wir in Zeile 17 einfügen. Das stellen wir in Abbildung 7.3 graphisch dar. Dieses Objekt ist nur innerhalb der Implementation von *Box* verfügbar, denn dort wurde es ja definiert. Die Funktion *main()* definiert Objekte der Klasse *Box*, aber keine der Klasse *PostInfo*, wir können uns also innerhalb dieser Funktion auf diese Klasse nicht beziehen. In unserem Fall ist die Klasse *PostInfo* dazu da, von der Klasse *Box* intern verwendet zu werden, wofür wir in Zeile 22 ein Beispiel geben: Wir senden eine Nachricht an die Methode *Etikett.Setze()*, um die Variablen zu initialisieren. Zusätzliche Methoden können je nach Bedarf verwendet werden, wir haben nur Beispiele für die Verwendung gegeben.

Wichtig ist, daß wir im Hauptprogramm kein Objekt der Klasse *PostInfo* deklarieren, sondern solche bei der Deklaration von Objekten der Klasse *Box* implizit deklarieren. Natürlich könnten wir auch im Hauptprogramm je nach Bedarf Objekte der Klasse *PostInfo* deklarieren, wir tun das aber in diesem Beispielprogramm nicht. Um die Sache zu komplettieren, sollte die Klasse *Box* eine oder mehrere Methoden haben, mit denen die Informationen, die im Objekt der Klasse *PostInfo* gespeichert sind, verwendet werden können. Schau Dir dieses Programm an, bis Du das neue Konstrukt verstanden hast, kompilieren Sie es und führe es aus. (Kino diesmal?)

Wenn die Klasse und die in sie geschachtelte Klasse Parameter für ihre jeweiligen Konstruktoren benötigen, kann man eine Initialisierungsliste verwenden. Das werden wir später in dieser Einführung diskutieren.

7.11 Operatorüberladung

Beispielprogramm: OPUEBER.CPP

Das Programm OPUEBER.CPP enthält Beispiele für das Überladen von Operatoren. Das ermöglicht Dir, eine Klasse von Objekten zu definieren und die Verwendung von normalen Operatoren für diese Klasse neu zu definieren. Das Ergebnis ist, daß die Objekte der Klasse genauso natürlich gehandhabt werden können wie vordefinierte Datentypen. Sie erscheinen als Teil der Sprache und nicht mehr als etwas von Dir Hinzugefügtes.

In diesem Fall überladen wir mit den Deklarationen in den Zeilen 11 bis 13 und den Definitionen in den Zeilen 17 bis 41 die Operatoren + und *****. Wir deklarieren die Methoden als Freundfunktionen (**friend**). Täten wir das nicht, wären die Funktionen Teil eines Objektes und die Nachricht würde an dieses Objekt gesendet werden. Durch die Deklaration als Freundfunktion trennen wir diese vom Objekt und können die Methode quasi „ohne“ Objekt aufrufen. Das erlaubt es uns, statt Objekt1.operator+(Objekt2) schlicht und einfach Objekt1 + Objkekt2 zu schreiben. Außerdem wäre ohne das **friend** Konstrukt ein Überladen mit einer Variable des Typs **int** als erstem Parameter nicht möglich, da wir an eine solche Variable keine Nachricht senden können, wie etwa int.operator+(Objekt1). Zwei der drei überladenen Operatoren verwenden eine ganze Zahl als ersten Parameter, die Deklaration als Freundfunktion ist also notwendig.

Es gibt kein Limit für die Anzahl an Überladungen pro Parameter. Du kannst jede beliebige Zahl an Überladungen eines Parameters durchführen, solange die Parameter für jede Überladung verschieden sind.

Der Methodenkopf in Zeile 17 illustriert die erste Überladung, die den Operator + betrifft. Wir geben den Rückgabetyp, gefolgt vom Schlüsselwort **operator** und dem Operator, den wir überladen wollen, an. Die beiden formalen Parameter und deren Typen führen wir in der Klammer an. Die Implementation der Funktion erfolgt in den Zeilen 19 bis 22. Dir ist sicherlich aufgefallen, daß die Implementationen von Freundfunktionen nicht Teil der Klasse sind, deshalb auch der Klassenname nicht angegeben ist. Die Implementation selbst ist biederes Handwerk und sollte Dir auf den ersten Blick einleuchten. Zu Illustrationszwecken vollbringen wir natürlich wieder mathematische Meisterwerke, Du kannst aber auch ganz einfach etwas Nützliches machen.

Die größte Außergewöhnlichkeit passiert in Zeile 57, wo wir die Methode mittels der Operatornotation aufrufen und nicht im normalen Format einer Nachricht, die wir an ein Objekt senden. Da die beiden Variablen *Klein* und *Mittel* Objekte der Klasse *Box* sind, wird das System nach einem Weg suchen, den Operator + mit zwei Objekten der Klasse *Box* zu verwenden und bei der eben diskutierten Methode *operator+()* fündig werden. Die Operationen, die wir in der Implementation der Methode durchführen müssen wie gesagt nicht so sinnlos sein, wie das, was wir hier demonstrieren.

In Zeile 59 weisen wir das System an, eine Variable vom Typ **int** zu einem Objekt der

Klasse *Box* zu addieren, das System findet also die zweite Überladung des Operators + in Zeile 26 und führt diese aus. In Zeile 61 schicken wir das System auf die Suche nach einer Methode, die eine Konstante vom Typ **int** und ein Objekt der Klasse *Box* mit dem Operator * verbindet und es landet in Zeile 35. Beachte bitte, daß es nicht erlaubt wäre, den Operator * anders herum zu verwenden, also zum Beispiel Grosz * 4, da wir keine Methode definiert haben, die diese Typen in dieser Reihenfolge akzeptiert. Würden wir eine solche Methode definieren könnten wir auch die umgekehrte Schreibweise verwenden, wobei dies zu einem komplett anderen Ergebnis führen kann, da es sich ja um zwei verschiedene Methoden handelt und wir in der Implementation ganz nach unserem Gutdünken panschen können.

Wenn wir Operatorüberladung verwenden, verwenden wir gleichzeitig auch das Überladen von Funktionsnamen, da einige der Funktionsname gleich sind. Wenn wir Operatorüberladung in dieser Form verwenden, sehen unsere Programme so aus, als wäre die Klasse ein natürlicher Teil der Sprache selbst. Deshalb ist C++ eine erweiterbare Sprache und kann an das jeweilige Problem angepaßt werden.

7.12 Nachteile des Überladens von Operatoren

Jedes neue Thema, das wir uns ansehen, hat seine Nachteile, vor denen gewarnt werden muß. Das Überladen von Operatoren hat deren anscheinend die meisten, da es so leicht falsch verwendet werden kann und viele Probleme mit sich bringt. Das Überladen von Operatoren ist nur für Klassen verfügbar, Du kannst die Operatoren für die vordefinierten einfachen Typen nicht neu definieren. Das wäre aber ohnedies nicht anzuraten, da Dein Code dann nur sehr schwer lesbar wäre.

Wenn Du den Inkrement- (++) oder den Dekrementoperator (–) überlädst, kann das System nicht feststellen, ob die Aktion vor der Verwendung der Variable im Programm durchgeführt werden soll oder nachher. Welche Methode verwendet wird, hängt allein von der Implementation ab, Du solltest diese Operatoren also in einer Art und Weise verwenden, wo es keine Rolle spielt, wie sie implementiert sind.

Kompiliere dieses Programm und führe es aus, bevor Du zum nächsten Beispielprogramm weitergehst.

7.13 Funktionsüberladung in einer Klasse

Beispielprogramm: FUNUEBER.CPP

Das Programm FUNUEBER.CPP gibt uns ein Beispiel für das Überladen von Funktionsnamen in einer Klasse. In dieser Klasse überladen wir den Konstruktor und eine der Methoden, um zu demonstrieren, was gemacht werden kann.

Diese Datei zeigt einige der Verwendungsmöglichkeiten von überladenen Funktionsnamen und ein paar Regeln für ihre Anwendung. Du wirst Dich erinnern, daß die Auswahl der Funktion allein nach der Anzahl und dem Typ der formalen Parameter erfolgt. Der Typ des Rückgabewertes hat darauf keinen Einfluß.

In diesem Fall haben wir drei Konstruktoren. Welcher der Konstruktoren aufgerufen wird, entscheidet sich anhand der Anzahl und der Typen der Parameter in der Definition. In Zeile 78 des Hauptprogrammes deklarieren wir die drei Objekte, jedes mit einer anderen Anzahl an Parametern und wenn wir uns das Ergebnis ansehen, können wir leicht erkennen, dass der jeweils richtige Konstruktor aufgerufen wurde.

Bei den anderen überladenen Methoden wird die passende Methode genauso festgestellt. Eine der Nachrichten beinhaltet eine Variable vom Typ **int**, eine andere eine Variable vom Typ **float**, das System findet aber die passende Methode. Du kannst eine Methode so oft überladen, wie Du willst, solange die Parameter eindeutig sind.

Vielleicht denkst Du nun, daß das ganze nicht viel Sinn macht und bist geneigt, es als Spielerei abzutun, das Überladen ist aber sehr wichtig. Die gesamte Einführung hindurch haben wir einen überladenen Operator verwendet, ohne daß Du irgendetwas Außergewöhnliches daran gefunden hättest. Es handelt sich um den Operator **¡¡**, der Teil der Klasse **cout** ist. Dieser Operator ist überladen, da die Form der Ausgabe von der Eingabevariable abhängt. Viele Programmiersprachen haben überladene Ausgabefunktionen, damit Du alle möglichen Daten mit ein und demselben Funktionsname ausgeben kannst.

Kompiliere das Programm und führe es aus.

7.14 Getrennte Kompilation

Die getrennte Kompilation von einzelnen Dateien folgt in C++ genau denselben Regeln wie in ANSI-C. Getrennt kompilierte Dateien können zusammen-„gelinkt“ werden. Da aber Klassen dazu verwendet werden, um Objekte zu definieren, ist die Natur der getrennten Kompilation in C++ von jener in ANSI-C verschieden. Die Klassen werden nämlich nicht als externe Variablen angesehen, sondern als interne Klassen. Das läßt das Programm anders als ein reines ANSI-C Programm aussehen.

7.15 Einige Methoden kommen von selbst

Beispielprogramm: AUTOMETH.CPP

Selbst wenn Du keine Konstruktoren oder überladene Operatoren schreibst, definiert der Compiler einige automatisch. Wirf einen Blick auf die Datei AUTOMETH.CPP, die einige dieser Methoden illustriert und zeigt, warum Du manchmal die automatisch bereitgestellten nicht verwenden kannst, sondern selbst Methoden schreiben mußt, die deren Arbeit übernehmen.

Bevor wir uns dem Programm selbst widmen, wollen wir einige der Regeln angeben, die Autoren von Compilern berücksichtigen müssen. Wir werden sie zuerst überblicksmäßig anführen, um dann ein wenig darauf einzugehen.

1. Wenn für eine Klasse kein Konstruktor existiert, erzeugt der Compiler automatisch einen Konstruktor und einen Kopierkonstruktor. Beide werden wir definieren.
2. Ist ein Konstruktor vorhanden, erzeugt der Compiler keinen.

3. Wenn für eine Klasse kein Kopierkonstruktor existiert, erzeugt der Compiler einen, nicht jedoch, wenn der Autor der Klasse einen Kopierkonstruktor geschrieben hat.

4. Ist ein Zuweisungsoperator vorhanden, erzeugt der Compiler nicht automatisch einen, andernfalls schon.

Für jede Klasse, die wir einem C++ Programm deklarieren und verwenden, muß es möglich sein, ein Objekt zu „konstruieren", da der Compiler definitionsgemäß einen Konstruktor aufrufen muß, wenn wir ein Objekt definieren. Wenn wir selbst keinen Konstruktor schreiben, erzeugt der Compiler selbst einen, den er beim Konstruieren von Objekten aufrufen kann. Das ist der Standardkonstruktor, den wir ohne es zu wissen, schon in vielen unserer Beispielprogramme verwendet haben. Der Standardkonstruktor initialisiert keine Elementvariablen, setzt aber alle internen Referenzen und ruft den Konstruktor (oder die Konstruktoren) der Basisklasse auf, falls solche existieren. Wir haben die Vererbung noch nicht studiert, werden uns diesem Kapitel aber im nächsten Abschnitt unserer Einführung widmen, dann wirst Du auch über Basisklassen Bescheid wissen. In Zeile 12 deklarieren wir einen Konstruktor, der aufgerufen wird, wenn wir ein Objekt ohne Parameter definieren. In diesem Fall ist der Konstruktor notwendig, da wir eine interne Zeichenkette haben, für die dynamisch Speicher beschafft und die mit einer leeren Zeichenkette initialisiert werden muß. Denk kurz darüber nach, und Du wirst sehen, daß dieser unser Konstruktor viel besser ist als der Standardkonstruktor, der einen initialisierten Zeiger zurücklassen würde.

Dieser Konstruktor wird in Zeile 79 des Beispielprogrammes verwendet.

7.16 Der Kopierkonstruktor

Wenn Du für eine Klasse selbst keinen Kopierkonstruktor definieren, wird dieser vo Compiler automatisch erzeugt. Er wird dazu verwendet, um bei der Erzeugung eines neuen Objektes den Inhalt eines anderen Objektes in das neue zu kopieren. Wenn der Compiler den Kopierkonstruktor für Dich definiert, kopiert dieser einfach den Inhalt Byte für Byte, was möglicherweise nicht genau das ist, was Du willst. Für einfache Klassen ohne Zeiger reicht der Standard-Kopierkonstruktor aus, aber in diesem Beispielprogramm haben wir einen Zeiger auf ein Element der Klasse und eine Kopie Byte für Byte würde auch diesen Zeiger einfach kopieren. Das würde dazu führen, daß beide Objekte auf dieselbe Adresse im Speicher und damit dieselbe Variable zeigen. Für unsere Klasse deklarieren wir den Kopierkonstruktor in der Zeile 15, die Implementation erfolgt in den Zeilen 35 bis 40. Wenn Du Dir die Implementation etwas genauer ansiehst, wirst Du feststellen, daß das neue Objekt in der Tat eine Kopie des alten ist, aber mit einer eigenen Zeichenkette. Da beide, der Konstruktor und der Kopierkonstruktor, dynamisch Speicherplatz beschaffen, müssen wir sicherstellen, daß dieser Speicher auch wieder freigegeben wird, wenn die Objekt zu existieren aufhören. Deshalb ist es notwendig, eine Destruktor bereitzustellen, dessen Implementation in den Zeilen 51 bis 54 dieses Beispielprogrammes zu finden ist. Den Kopierkonstruktor verwenden wir in Zeile 85.

7.17 Der Zuweisungsoperator

Es ist zwar nicht gleich ersichtlich, aber auch ein Zuweisungsoperator ist für dieses Beispielprogramm notwendig, da der Standardzuweisungsoperator einfach eine Kopie des Speicherinhaltes herstellt. Das würde in demselben Problem resultieren wie beim Kopierkonstruktor. Wir deklarieren den Zuweisungsoperator in Zeile 18 und definieren ihn in den Zeilen 42 bis 49. Wir löschen zuerst die alte Zeichenkette des existierenden Objektes, dann stellen wir den Speicher für den neuen Text bereit und kopieren ihn vom Quellobjekt in das neue Objekt. Den Zuweisungsoperator verwenden wir in der Zeile 92.

Es ist ziemlich offensichtlich, dass die obigen drei Methoden zusätzlich zu einem Destruktor bereitgestellt werden sollten, wenn eine Klasse dynamische Speicherverwaltung beinhaltet. Wenn Du einen der vier vergißt, kann das Programm ein eigenartiges Verhalten an den Tag legen. Kompiliere dieses Programm und führe es aus.

7.18 Ein praktisches Beispiel

Beispielprogramm: PHRASE.H

Die Verwendung des Schlüsselwortes **inline** kann einige Verwirrung stiften, wenn Du nicht weißt, wie der Compiler die Definition des Codes bei der Implementation verwendet. Betrachte die header-Datei PHRASE.H, die einige **inline**-Methoden enthält. Wir wollen damit einen sauberen Weg aufzeigen, wie **inline**-Methoden deklariert werden können, mit denen der Compiler auch etwas anfängt.

Jede Implementation, die diese Klasse verwendet, muß Zugriff auf die Implementation der **inline**-Methoden haben, um den Code für die Methoden einsetzen zu können. Ein Weg, dies zu erreichen ist, eine eigenen Datei (Erweiterung INL) mit allen **inline**-Methoden zu erstellen und diese Datei dann am Ende der header-Datei einzufügen, wie wir es hier in Zeile 17 tun. Dadurch steht dem Compiler jederzeit der gesamte Code zur Verfügung.

Beispielprogramm: PHRASE.INL

Die Datei PHRASE.INL enthält alle **inline**-Methoden der Klasse.

Beispielprogramm: PHRASE.CPP

Der einzige Grund, warum es diese Datei überhaupt gibt, ist, um die statische Variable *GanzePhrase* zu definieren. Da es sich um eine Definition handelt und deshalt Speicher belegt wird, können wir sie nicht in eine header-Datei schreiben. Täten wir das doch, hätte es den Anschein, als funktionierte alles prächtig, da die header-Datei nur einmal verwendet wird. Das Verwenden einer schlechten Programmiertechnik wie dieser würde aber früher oder später zu Problemen führen. Aus Demonstrationsgründen haben wir alle Methoden **inline** deklariert, deshalb gibt es in dieser Datei keine Definitionen von Methoden.

Beispielprogramm: VERPHRAS.CPP

Die Datei VERPHRAS.CPP verwendet die Klasse *Phrase*, die wir in den letzten beiden Beispielprogrammen definiert haben. Es ist klar zu sehen, daß diese Klasse sich in nichts von all den anderen, die wir schon betrachtet haben, unterscheidet. Sie zeigt nur, wie

inline-Code einfach und effizient gehandhabt werden kann.

7.19 Ein weiteres praktisches Beispiel

Auch hier wollen wir uns eine praktische Klasse ansehen, die in einem Programm verwendet werden kann, aber doch einfach genug ist, damit Du sie komplett verstehen kannst.

Beispielprogramm: ZEIT.H

Im letzten Kapitel haben wir uns die Klasse *Datum* angesehen, in diesem wollen wir uns der Klasse *Zeit* widmen. Du solltest mit dem Studium der Datei ZEIT.H beginnen, die der header-Datei der *Datum* Klasse sehr ähnlich sieht. Der einzige große Unterschied sind die überladenen Konstruktoren und Methoden. Es handelt sich hier um ein sehr praktisches Beispiel, das recht gut illustriert, daß viele Überladungen des Konstruktors möglich sind.

Beispielprogramm: ZEIT.CPP

Die Implementation der Klasse *Zeit* erfolgt in der Datei ZEIT.CPP. Auch dieser Code ist sehr einfach und Du solltest keinerlei Schwierigkeit haben, ihn vollkommen zu verstehen. Drei der vier überladenen Funktionen rufen die vierte auf, damit der Code dieser Methode nicht vier Mal wiederholt werden muss. Das ist relativ gute Programmierpraxis und zeigt, dass innerhalb der Implementation andere Elementfunktionen aufgerufen werden können.

Wie wir schon einmal erwähnt haben, verwendet der Code Betriebssystemaufrufe und ist damit nicht ohne weiteres portierbar. Du mußt den Code dann adaptieren oder einfach Standardwerte zuweisen.

Beispielprogramm: VERZEIT.CPP

Das Beispielprogramm VERZEIT.CPP verwendet die Klasse *Zeit* in einer sehr einfachen Art und Weise. Du solltest es sofort verstehen. Es ist sicherlich nützlich, wenn Du die beiden praktischeren Beispiele am Ende des letzten und dieses Kapitels verstanden hast. Wie schon erwähnt werden wir nämlich die Klassen *Zeit* und *Datum* als Basis der einfachen und der vielfachen Vererbung benutzen, wenn wir diese in den nächsten drei Kapiteln untersuchen.

7.20 Was sollte der nächste Schritt sein?

Jetzt weißt Du schon genug über C++, um sinnvolle Programme zu schreiben und es hilft Dir sicherlich sehr, wenn Du diese Einführung ein wenig beiseite legst und beginnst, das anzuwenden, was Du gelernt hast. Da C++ ja eine Erweiterung von ANSI-C ist, können die Schritte beim Erlernen etwas kleiner sein, als wenn man eine komplett neue Sprache erlernt. Du hast mittlerweile genug gelernt, um Dich mit dem Beispielprogramm des Kapitels 13, dem Abenteuerspiel, auseinanderzusetzen. Damit solltest Du jetzt beginnen.

Das größte Problem ist es, objektorientiert zu denken. Das ist gar nicht so einfach, wenn man einige Zeit in prozeduralen Sprachen programmiert hat. Aber auch das kann man (nur) durch Übung lernen, Du solltest also versuchen, in Klassen und Objekten

zu denken. Dein erstes Projekt sollte nur einige Objekte umfassen, den Rest kannst Du ruhig prozedural schreiben. Mit der Zeit und der Erfahrung wirst Du immer mehr Code mit Klassen und Objekten schreiben, vollenden wirst Du aber jedes Projekt mit prozeduralen Programmiertechniken.

Wenn Du einige Zeit mit den Techniken, die wir bis hierher in dieser Einführung besprochen haben, programmiert hast, kannst Du zum nächsten Kapitel weitergehen. Dieses Kapitel wird sich mit Vererbung und virtuellen Funktionen beschäftigen.

7.21 Programmieraufgaben

1. Mach aus *Klein* und *Mittel* in OBJDYNAM.CPP Zeiger. Beschaffe Speicherplatz für sie, bevor Du sie verwendest.

2. Ändere in OBJVERB.CPP die Schleife in Zeile 62 so, daß sie 1000 Elemente speichert. Es ist wahrscheinlich klug, die Ausgabe in Zeile 81 auszukommentieren. Möglicherweise erzeugst Du mit so großen Zahlen einen Speicherüberlauf, was Du an falschen Antworten erkennst, wenn Du eine Nachricht an *HoleFlaeche()* sendest. Das hängt von Deinem Compiler ab.

3. Schreibe ein Programm, das sowohl die *Zeit* als auch die *Datum* Klasse sinnvoll verwendet. Diese beiden Klassen kannst Du in allen zukünftigen C++ Programmen verwenden, um Zeit und Datum der Ausführung festzuhalten.

8 Vererbung

Ein Grund für die Verwendung von Vererbung ist die Möglichkeit, alten Code wiederzuverwenden, wobei Du diesen aber leicht verändern kannst, wenn der Code eines früheren Projektes die Anforderungen des neuen nicht ganz genau erfüllt. Es macht keinen Sinn, bei jedem neuen Programm mit null zu beginnen, da es sicherlich Code gibt, der in vielen Programmen vorkommt und Du solltest so viel davon verwenden wie möglich. Allerdings werden beim Adaptieren alter Klassen auch leicht und schnell Fehler gemacht. Solche Fehler passieren nicht so schnell, wenn Du die alte Klasse läßt, wie sie ist, und ihr einfach hinzufügst, was Du brauchst. Ein weiterer Grund für die Verwendung von Vererbung ergibt sich, wenn Du in einem Programm viele Klassen benötigst, die sehr ähnlich, aber eben doch nicht gleich sind.

In diesem Kapitel werden wir uns auf die Technik der Vererbung konzentrieren und wie wir diese in ein Programm einbauen. Bessere Beispiele, warum wir Vererbung überhaupt verwenden würden werden wir in späteren Kapiteln geben, wenn wir uns einige praktische Beispiele objektorientierten Programmierens ansehen. Das Konzept der Vererbung ist in einigen modernen Programmiersprachen vorhanden, jede geht natürlich ein wenig anders damit um. C++ erlaubt es Ihnen, alle oder einen Teil der Elemente und Methoden einer Klasse zu übernehmen, einige zu verändern und neue Elemente und Methoden hinzuzufügen. Es steht Dir also alle Flexibilität zur Verfügung und auch hier wurde die Methode gewählt, die in einem möglichst effizienten Code resultiert.

8.1 Eine einfache Klasse als Hors d'oeuvre

Beispielprogramm: VEHIKEL.H

Die einfache Klasse, die Du in VEHIKEL.H findest, soll uns als Ausgangspunkt für das Studium der Vererbungslehre dienen. An dieser header-Datei ist nichts Außergewöhnli-

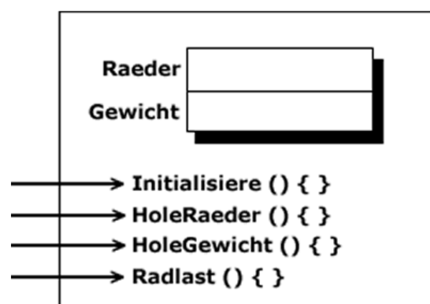

Abbildung 8.1: Die Klasse *Vehikel*

ches, sie ist sehr einfach gehalten. Sie besteht aus vier Methoden, die wir zum Manipulieren der Daten unseres Vehikels verwenden können. Was die einzelnen Funktionen tun ist jetzt nicht wichtig. Wir werden uns auf diese Klasse später als Basisklasse oder Elternklasse beziehen, einstweilen aber soll sie nichts sein als eine ganz normale Klasse und wir werden sie auch so verwenden, um zu zeigen, daß sie sich nicht von all den Klassen unterscheidet, die wir schon kennen. Auf das neue Schlüsselwort **protected** werden wir gleich zurückkommen. Abbildung 8.1 ist eine graphische Darstellung der Klasse *Vehikel*.

Beachte die Zeilen 4, 5 und 19 bis zum Ende des Kapitels nicht. Wir werden Sie dann erklären. Diese Datei kann nicht kompiliert oder ausgeführt werden, da es sich ja nur um eine header-Datei handelt.

8.2 Die Implementierung useres Vehikels

Beispielprogramm: VEHIKEL.CPP

Die Datei VEHIKEL.CPP enthält die Implementation der Klasse *Vehikel*. Die Methode mit dem Namen *Initialisiere()* weist den Variablen *Raeder* und *Gewicht* die Werte der Parameter zu. Wir haben Methoden, die die Anzahl der *Raeder* und das *Gewicht* zurückgeben und schließlich eine, die eine einfache Rechnung ausführt und das Gewicht zurückgibt, das auf jedem Rad lastet. Später werden wir einige Beispiele für Methoden geben, die interessantere Dinge vollbringen, jetzt sind wir aber mehr an der Schnittstelle interessiert, wir halten also die Implementation relativ simpel.

Wie wir oben festgehalten haben, handelt es sich um eine sehr einfache Klasse, die wir im nächsten Programm verwenden werden. Später in diesem Kapitel werden wir sie als Basisklasse verwenden. Du solltest die Klasse nun kompilieren, ausführen kannst Du die Klasse alleine freilich nicht.

8.3 Wir verwenden die Vehikel Klasse

Beispielprogramm: TRANSPRT.CPP

Das Beispielprogramm TRANSPRT.CPP verwendet die Klasse *Vehikel*. Damit sollte klar sein, daß die Klasse *Vehikel* tatsächlich nicht mehr ist als eine normale Klasse in C++. Wir werden ein bißchen mehr aus ihr machen, wenn wir sie in den nächsten paar Beispielprogrammen als Basisklasse verwenden, um die Vererbung zu demonstrieren. Vererbung verwendet eine existierende Klasse und erweitert sie um Funktionalität, um eine andere, möglicherweise komplexere Aufgabe zu erfüllen.

Das Verständnis dieses Programmes sollte Dir mittlerweile wirklich keine Schwierigkeiten mehr bereiten. Wir deklarieren vier Objekte der Klasse *Vehikel*, initialisieren sie und geben einige der Werte am Bildschirm aus, um zu zeigen, daß die Klasse *Vehikel* wie eine einfache Klasse verwendet werden kann, da es sich ja um eine einfache Klasse handelt. Wir nennen diese Klasse (noch) einfache Klasse, im Gegensatz zu einer Basisklasse oder einer Kindklasse, wie wir es gleich tun werden.

Wenn Du dieses Programm verstanden hast, kompiliere es und führe es aus. Linke dazu die Objektdatei der Klasse *Vehikel* mit der des Programmes.

8.4 Unsere erste abgeleitete Klasse

Beispielprogramm: AUTO.H

Die Datei AUTO.H enthält unser erstes Beispiel einer abgeleiteten- oder Kindklasse. Die Klasse *Vehikel* wird durch „:public Vehikel" in Zeile 7 ererbt. Die abgeleitete Klasse *Auto* besteht aus allen Elementen der Basisklasse *Vehikel* und allen ihren eigenen Elementen. Obwohl wir an der Klasse mit dem Namen *Vehikel* nichts verändert haben, wurde sie zur Basisklasse allein durch die Art und Weise, wie wir sie hier verwenden. Obzwar sie hier als Basisklasse eingesetzt wird, gibt es keinen Grund, sie nicht weiterhin auch als ganz normale Klasse zu verwenden wie etwa im vorigen Beispielprogramm. Sie kann in ein und demselben Programm als Basisklasse und als einfache Klasse auftreten. Über die Frage, ob es sich um eine Basisklasse oder eine einfache Klasse handelt, entscheidet allein die Verwendung.

Wir müssen uns wieder ein wenig der Terminologie widmen. Im objektorientierten Programmieren wird eine Klasse, die von einer anderen Elemente erbt, generell abgeleitete Klasse oder Kindklasse genannt, in C++ spricht man meist von einer abgeleiteten Klasse. Da die beiden Ausdrücke recht bildhaft sind und sie meist austauschbar verwendet werden, werden auch wir in dieser Einführung beide gebrauchen. So ist der „richtige" Ausdruck für die Klasse, von der geerbt wird, in C++ Basisklasse, aber auch Elternklasse oder Superklasse können verwendet werden.

Eine Basisklasse ist eine recht generelle Klasse, die ein breites Spektrum an Objekten umfaßt, wogegen eine abgeleitete Klasse etwas spezieller und eingeengter, dafür aber auch nützlicher ist. Nehmen wir zum Beispiel an, wir hätten eine Basisklasse mit dem Namen Programmiersprache und eine abgeleitete Klasse mit dem Namen C++. Mit der Basisklasse könnten wir Pascal, Ada, C++ oder jede andere Programmiersprache definieren, sie könnte uns aber keine Auskunft darüber geben, wie Klassen in C++ verwendet werden, da sie von jeder Programmiersprache nur ein allgemeines Bild vermittelt. Die abgeleitete Klasse C++ hingegen könnte die Verwendung von Klassen definieren, nicht jedoch eine der anderen Programmiersprachen, da sie dafür zu eingeengt ist. Eine Basisklasse ist genereller, eine abgeleitete Klasse spezieller.

Im Fall unseres Beispielprogrammes kann die Basisklasse *Vehikel* verwendet werden, um so unterschiedliche Objekte wie Lastwägen, Autos, Fahrräder oder jedes andere Vehikel zu deklarieren. Mit der Klasse *Auto* hingegen können wir nur Objekte deklarieren, die auch Autos sind, da wir die Informationen, die mit dieser Klasse verwendet werden können, eingeschränkt haben. Deshalb ist die Klasse *Auto* spezieller und eingeschränkter als die Klasse *Vehikel.* Die *Vehikel* Klasse ist umfassender.

Wenn wir eine noch spezieller Klasse wollen, können wir *Auto* als Basisklasse verwenden um eine Klasse mit dem Namen *Sportauto* zu deklarieren und Informationen wie etwa *Hoechstgeschwindigkeit* oder *Manta?* inkludieren, Informationen also, die für ein normales Auto nicht wirklich von Belang sind. Die Klasse *Auto* würde dann zur selben Zeit als abgeleitete Klasse und als Basisklasse fungieren, womit klar sein sollte, daß diese Ausdrücke sich nur darauf beziehen, wie eine Klasse verwendet wird.

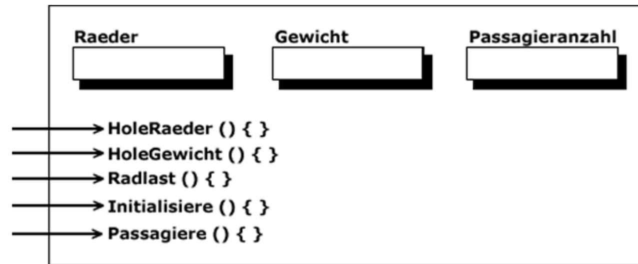

Abbildung 8.2: Ein Objekt der Klasse *Auto*

8.5 Wie deklarieren wir eine abgeleitete Klasse?

Genug des allgemeinen Geschwafels. Wir definieren eine abgeleitete Klasse indem wir erstens die header-Datei der Basisklasse importieren (Zeile 5) und zweitens den Namen der Basisklasse nach dem Namen der abgeleiteten Klasse, von diesem durch einen Doppelpunkt getrennt, angeben (Zeile 7). Beachte das Schlüsselwort **public** gleich nach dem Doppelpunkt vorerst nicht. Es definiert öffentliche Vererbung, der wir unser nächstes Kapitel widmen wollen. Alle Objekte, die wir als Objekte der Klasse *Auto* deklarieren, bestehen also aus den beiden Variablen der Klasse *Vehikel*, da sie diese erben, und der Variable, die wir in der Klasse *Auto* selbst deklarieren, *Passagieranzahl*.

Ein Objekt dieser Klasse hat drei der vier Methoden der Klasse *Vehikel* und die beiden neuen, die wir hier deklarieren. Die Methode *Initialisiere()* der Klasse *Vehikel* ist nicht verfügbar, da sie durch die lokale Version der Methode *Initialisiere()*, die Teil der Klasse *Auto* ist, verdeckt wird. Wird also ein Methodenname der Basisklasse in der abgeleiteten Klasse wiederholt, wird die lokale Version verwendet, um ein Anpassen der Methoden zu erlauben. Abbildung 8.2 zeigt ein Objekt der Klasse *Auto*.

Die Implementation der Basisklasse muß nur in kompilierter Form vorhanden sein. Der Code der Implementation kann aus ökonomischen Gründen vorenthalten werden, um Entwicklerinnen von Software zu helfen. Die header-Datei der Basisklasse muß allerdings verfügbar sein, da die Definition der Klasse benötigt wird, um sie zu verwenden.

8.6 Die Implementierung der Klasse

Beispielprogramm: AUTO.CPP

Die Datei AUTO.CPP ist die Implementation der Klasse *Auto*. Es wird Dir sicherlich gleich auffallen, daß man nirgends erkennen kann, daß es sich um eine abgeleitete Klasse handelt. Das kann nur aus der header-Datei der Klasse ersehen werden. Da wir nicht einmal feststellen können, ob es sich um eine abgeleitete oder eine einfache Klasse handelt, erfolgt die Implementation in genau derselben Weise wie bei jeder anderen Klasse auch.

Die Implementationen der beiden Methoden sind in genau derselben Art und Weise geschrieben wie Methoden für jede andere Klasse. Wenn Du das Gefühl hast, verstanden zu haben, worum es geht, solltest Du diese Datei für spätere Verwendung kompilieren.

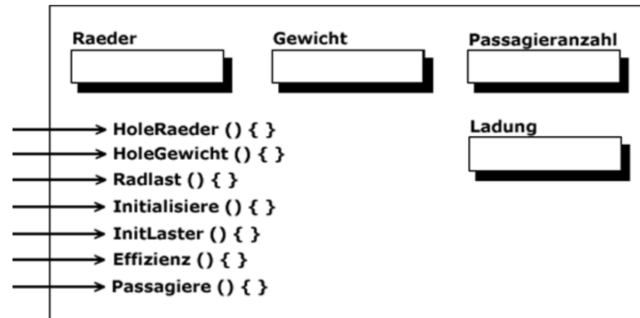

Abbildung 8.3: Die Klasse *Laster*

8.7 Eine weitere abgeleitete Klasse

Beispielprogramm: LASTER.H

Die Datei LASTER.H gibt ein weiteres Beispiel für eine Klasse, die die Klasse *Vehikel* als Basisklasse verwendet und sie erweitert. Natürlich erweitert sie die Basisklasse anders als die Klasse *Auto*, da wir uns jetzt auf die Dinge konzentrieren, die für einem Lastwagen signifikant sind. Die Klasse fügt zwei Variablen und drei Methoden hinzu. Beachte auch hier das Schlüsselwort **public** nicht. Abbildung 8.3 repräsentiert die Klasse *Laster*.

Es ist wichtig, zu erkennen, daß die beiden Klassen *Auto* und *Laster* absolut nichts miteinander zu tun haben, sie sind nur zufällig beide von derselben Basisklasse abgeleitet.

Sowohl die Klasse *Auto* als auch die Klasse *Laster* haben eine Methode mit dem Namen *Passagiere()*. Das ist aber kein Problem. Wenn Klassen in irgendeiner Beziehung zueinander stehen (wie diese mit derselben Basisklasse), erwartet man, daß sie auch ähnliche Dinge tun. In diesem Fall ist es logisch, auch denselben Methodennamen in beiden abgeleiteten Klassen zu verwenden.

8.8 Die Implementierung des Lasters

Beispielprogramm: LASTER.CPP

Die Datei LASTER.CPP enthält die Implementation der Klasse *Laster*. Es geschieht hier nichts Außergewöhnliches.

Du solltest kein Problem haben, diese Implementation zu verstehen. Kompiliere also diese Klasse in Vorbereitung unseres Beispielprogrammes, das dann alle drei Klassen, die wir in diesem Kapitel definiert haben, verwenden wird.

8.9 Wir verwenden alle drei Klassen

Beispielprogramm: ALLEVEH.CPP

Die Datei ALLEVEH.CPP gibt ein Beispiel für ein Programm, das alle drei in diesem Kapitel diskutierten Klassen verwendet. Es deklariert Objekte der Elternklasse *Vehikel*

und der beiden Kindklassen. Das soll zeigen, daß alle drei Klassen in einem Programm verwendet werden können.

Wir importieren die header-Dateien aller Klassen in den Zeilen 3 bis 5, damit wir die Klassen auch verwenden können. Beachte, daß die Implementationen der Klassen nicht sichtbar sind und dies auch nicht notwendig ist. Das ermöglicht es, den Code zu verwenden, ohne direkten Zugriff auf den Quellcode zu haben. Die Informationen in den header-Dateien müssen allerdings schon verfügbar sein.

In diesem Beispiel deklarieren wir nur ein Objekt für jede Klasse, wir könnten aber nach Herzenslust Objekte deklarieren und verwenden, so viele wir brauchen. Der Code für dieses Programm ist sehr klar. Wir haben die Klassen zuvor entwickelt, fehlerbereinigt und gespeichert und dabei die Schnittstelle möglichst einfach gehalten. Ansonsten findet sich in diesem Programm nichts Neues, das Verständnis sollte also keine allzu großen Schwierigkeiten bereiten.

Das Kompilieren dieses Programmes ist mit ein wenig Arbeit verbunden, aber nicht kompliziert. Die drei Klassen und das Hauptprogramm (mit der Funktion *main()*) können in beliebiger Reihenfolge kompiliert werden. Es müssen nur alle vier kompiliert sein, bevor wir sie linken können. Schließlich kannst Du das resultierende Programm ausführen. Tu das. Eine effiziente Nutzung von C++ wird es ständig erfordern, viele einzelne Dateien zu kompilieren und dann zu linken. Das liegt in der Natur von C++, sollte aber keine allzu schwere Last auf Deinen (atlantischen) Schultern sein, wenn Du über ein praktisches „make“ oder einen Compiler mit Projektverwaltung verfügst.

8.10 Wozu #ifndef VEHIKEL_H?

Ich habe versprochen, auf die eigenartigen Präprozessoranweisungen in den Zeilen 4, 5 und 19 von VEHIKEL.H zurückzukommen. Nun ist die Zeit reif. Wenn wir die abgeleitete Klasse *Auto* definieren, müssen wir die gesamte Definition der Schnittstelle der Klasse *Vehikel* zur Verfügung stellen, da *Auto* eine von *Vehikel* abgeleitete Klasse ist und alles über die Basisklasse wissen muß. Wir tun dies, indem wir die header-Datei der Klasse *Vehikel* in die Klasse *Auto* importieren. Aus dem gleichen Grund müssen wir sie auch in die Klasse *Laster* importieren.

Schließlich müssen wir auch das Programm ALLEVEH.CPP über die Details der drei Klassen informieren und deshalb die header-Dateien importieren, was wir in den Zeilen 3 bis 5 auch tun. Das führt aber zu einem Problem. Wenn der Präprozessor zur Klasse *Auto* kommt, importiert es die Klasse *Vehikel*, da sie in der header-Datei der Klasse *Auto* aufgeführt ist. Da aber die Klasse *Vehikel* schon in Zeile 3 von ALLEVEH.CPP importiert wurde, wird sie zweimal importiert und wir versuchen damit, die Klasse *Vehikel* noch einmal zu deklarieren. Natürlich ist die Deklaration dieselbe, was das System aber nicht weiter kümmert: es verbietet schlicht und einfach eine zweite Deklaration einer Klasse. Wir gestatten das zweifache Importieren der Datei und verhindern gleichzeitig das zweifache Importieren der Klasse mit dem Wort VEHIKEL_H. Wenn das Wort schon definiert wurde, wird die Deklaration der Klasse übergangen, wurde es aber noch nicht definiert, wird die Deklaration importiert und gleichzeitig VEHIKEL_H definiert. Das

Endergebnis dieses Kunstgriffs ist, daß die Klasse nur einmal importiert (und damit deklariert) wird, obwohl wir die Datei öfters importieren. Du solltest kein Problem haben, die Logik zu verstehen, wenn Du Dich kurz damit auseinandergesetzt hast.

Obwohl ANSI-C mehrfache Deklarationen erlaubt, solange sie identisch sind, verbiete C++ dies. Der Hauptgrund dafür ist, daß es für den Compiler sehr schwierig wäre, festzustellen, ob er schon einen Konstruktoraufruf für ein nochmals definiertes Element gemacht hat. Ein wiederholter Konstruktoraufruf für ein einzelnes Objekt kann ziemlichen Unsinn anrichten, also wurde das von vornherein ausgeschlossen, indem per Definitionem wiederholte Definitionen verboten wurden. Das stellt in keinem praktischen Programm ein Problem dar.

Wir haben den Namen VEHIKEL_H gewählt, weil es sich dabei um den Namen der Datei handelt, wobei der Punkt durch eine Unterstrich ersetzt wurde. Wenn Du das konsequent mit allen Deinen Klassen so hältst, kann es nie zu Namenskonflikten kommen, da jede Datei einen eigenen Namen haben muß, solange sich alle in einem Verzeichnis befinden. Es ist sicherlich gut, sich an diesen Kunstgriff zu gewöhnen. Alle weiteren header-Dateien in dieser Einführung werden ihn verwenden, um wiederholte Definitionen von Klassen zu verhindern und um Dir ein Beispiel zu sein. Du solltest das wirklich in allen Klassendefinitionen verwenden, wie trivial sie auch sein mögen.

8.11 Unsere erste praktische Vererbung

Beispielprogramm: DATUMNEU.H

Wir schließen an Kapitel 6 an, erben die Klasse *Datum* in der Datei DATUMNEU.H und erweitern sie um eine Variable und eine Methode. Eigentlich ist dies kein besonders guter Weg, *TagDesJahres* zur Klasse *Datum* hinzuzufügen, da sie ein Teil der Struktur ist, die der Systemaufruf in der Klasse *Datum* zurückgibt. Wir sind aber mehr daran interessiert, Vererbung in einem brauchbaren Beispiel zu demonstrieren als eine perfekte Klasse zu entwerfen, wir werde also damit leben können.

Beispielprogramm: DATUMNEU.CPP

Die Datei DATUMNEU.CPP enthält die Implementation der neuen Methoden und sollte Dich vor keine großen Rätsel stellen. Diese Klassenimplementation verwendet den Array *Tage[]* aus der Implementation der Klasse *Datum*, der dort als globale Variable definiert wurde. Die Logik der Methode *HoleTagDesJahres()* ist sehr einfach. Sie kümmert sich nicht einmal um Schaltjahre. Wir sind eben nicht wirklich daran interessiert, eine gute Datumsklasse zu schreiben, wohl aber am Erlernen der Vererbung.

Beispielprogramm: VERDATN.CPP

Schließlich verwendet das Programm mit dem Namen VERDATN.CPP die neue Klasse in einer einfachen Weise, um zu illustrieren, daß die abgeleitete Klasse genauso einfach zu verwenden ist wie die Basisklasse und das Programm eigentlich keine Möglichkeit hat, festzustellen, daß es eine abgeleitete Klasse verwendet.

Du solltest auch dieses Programm kompilieren und linken, um darin weitere Erfahrung zu sammeln. Du mußt die Objektdateien der originären Klasse *Datum*, der abgeleiteten Klasse *DatumNeu* und der Hauptprogrammes linken.

8.12 Programmieraufgaben

1. Erzeuge in ALLEVEH.CPP ein weiteres Objekt der Klasse *Vehikel* mit dem Namen *Fahrrad* und verfahre damit in etwa so wie mit dem *Hochrad*. Du mußt dann nur das Hauptprogramm neu kompilieren und alle vier Dateien linken. Die drei Klassen mußt Du nicht neu kompilieren.

2. Erweitere die Klasse *Laster* um eine Methode, die das Gewicht des Lasters plus dem der Ladung zurückgibt und schreibe in ALLEVEH.CPP Code, der diesen Wert liest und am Bildschirm ausgibt. Du mußt also die Dateien LASTER.H, LASTER.CPP und natürlich ALLEVEH.CPP ändern.

3. Erweitere die Klasse *Name*, die Du in Kapitel 6 geschrieben hast, um die Variable *Geschlecht* und um Methoden, den Wert dieser Variable zu setzen und auszulesen. Erlaubt sind nur 'M' oder 'F' (?). Du solltest diese Erweiterungen durchführen, indem Du eine von *Name* abgeleitete Klasse erstellst.

9 Mehr Vererbungslehre

Im letzten Kapitel haben wir uns ein Modell mit Fortbewegungsmöglichkeiten zusammengezimmert, um das Konzept der Vererbung zu illustrieren. Wir werden dieses Modell in diesem Kapitel fortführen, um einige Feinheiten der Vererbung und mögliche Anwendung zu demonstrieren. Wenn es schon längere Zeit her ist, dass Du das Kapitel 8 durchgelesen hast, wäre es eine gute Idee, dieses Kapitel noch einmal kurz zur Hand zu nehmen, als Vorbereitung auf ein eingehenderes Studium der dort besprochenen Konzepte.

9.1 Neuorganisierte Dateistruktur

Beispielprogramm: VERERB1.CPP

Wenn Du Dir das Programm VERERB1.CPP etwas genauer ansiehst, wirst Du bemerken, daß es sich um das Programm ALLEVEH.CPP handelt, das wir in Kapitel 8 entwickelt haben. Der Code wurde nur neu arrangiert. Der größte Unterschied ist, daß einige der einfacheren Funktionen zu inline-Funktionen umfunktioniert wurden. Solche kurzen Funktionen solltest Du immer **inline** deklarieren, da dies nicht nur weniger Schreibarbeit für Dich bedeutet, sondern auch die Abarbeitungsgeschwindigkeit erhöht.

Die andere Änderung ist die der Anordnung von Klassen und deren Methoden, wobei wir zuerst alle Klassen definieren, gefolgt vom Hauptprogramm. Damit haben wir alle Schnittstellendefinitionen auf einer Seite, was das Studium des Code etwas erleichtern sollte. Die Implementationen der Methoden kommen ganz zum Schluß, wo sie auch bei der Hand sind, aber sie stören nicht die Klassendefinitionenen, die wir uns in diesem Kapitel etwas genauer ansehen wollen. Das sollte Dir zeigen, daß die Anordnung von Klassen und Methoden in C++ sehr flexibel erfolgen kann. Natürlich hast Du auch bemerkt, daß wir hier gegen die Natur von C++ und ihr Konzept der getrennten Kompilation verstoßen. Am besten sollten alle Dateien so aufgeteilt werden wie wir es in Kapitel 8 getan haben.

Wie wir schon erwähnt haben, kennen beide abgeleiteten Klassen, *Auto* und *Laster* eine Variable mit dem Namen *Passagieranzahl*, was erlaubt ist. Die Klasse *Auto* hat eine Methode, *Initialisiere()*, die denselben Namen trägt wie eine Methode der Superklasse mit dem Namen *Vehikel*. Die Neuordnung der Dateien ändert nichts an der Legalität dieses Konstrukts.

Wenn Du Dich überzeugt hast, daß diese Datei identisch mit dem Programm ALLEVEH.CPP aus Kapitel 8 ist, kompiliere es und führe es aus. Da wir alles in eine Datei gezwängt haben, benötigst Du dafür auch kein „make“ mehr. Das ist auch einer der Gründe, warum wir für dieses Kapitel diese Form gewählt haben: das kompilieren geht schneller und einfacher.

9.2 Der scope Operator

Da die Methode *Initialisiere()* in der abgeleiteten Klasse *Auto* deklariert wird, versteckt sie die Methode mit demselben Namen, die ein Element der Basisklasse ist. Es kann aber vorkommen, daß Du die Methode der Basisklasse in der abgeleiteten Klasse verwenden willst. Das erreichst Du, indem Du den scope Operator im Hauptprogramm folgendermaßen verwendest:

Sedan.Vehikel::Initialisiere(4, 1600.0);

Wie Du vermuten wirst, müssen Anzahl und Typ der Parameter mit denen der Methode der Basisklasse übereinstimmen, da diese auf diese Nachricht antwortet.

9.3 Was sind geschützte Daten?

Wären alle Daten der Basisklasse in allen von dieser abgeleiteten Klassen verfügbar, wäre es für die Programmiererin eine leichte Übung, die Basisklasse zu ererben und auf alle Daten freien Zugriff zu haben. Das würde den Zugriffsschutz komplett ausschalten. Aus diesem Grund sind die Daten einer Klasse in einer von dieser abgeleiteten Klasse nicht automatisch verfügbar. Es kommt aber vor, daß Du alle Variablen in die abgeleitete Klasse übernehmen und so verwenden wills, als wären sie dort deklariert. Aus diesem Grund haben die Autoren von C++ das Schüsselwort **protected** erfunden.

In unserem Beispielprogramm finden wir dieses Schlüsselwort in Zeile 6, womit alle Daten der Klasse *Vehikel* in allen abgeleiteten Klassen zur Verfügung stehen, nicht jedoch außerhalb der Basisklasse oder einer der abgeleiteten Klassen. Wie wir schon früher festgestellt haben, sind alle Daten am Beginn einer Klasse automatisch **private**, wenn wir nichts angeben.

Die Variablen *Raeder* und *Gewicht* sind in der Methode mit dem Namen *Initialisiere()* in den Zeilen 83 bis 88 verfügbar, als wären sie als Teil der Klasse *Auto* selbst deklariert worden. Wir können sie verwenden, da wir sie in der Basisklasse als **protected** deklariert haben. Natürlich wären sie hier auch verfügbar, wenn wir sie als **public** deklariert hätten, dann wären sie aber auch außerhalb der beiden Klassen verfügbar und wir würden allen Zugriffsschutz verlieren. Selbstredend sind die beiden Variablen in der Basisklasse in den Zeilen 77 und 78 auch verfügbar. Jetzt können wir die Regeln für die drei Arten des Zugriffes auf Variablen und Methoden aufstellen.

- **private** - Die Variablen und Methoden sind außerhalb der Klasse selbst nicht verfügbar, auch nicht in von dieser Klasse abgeleiteten Klassen.
- **protected** - Die Variablen und Methoden sind außerhalb der Klasse selbst und von ihr abgeleiteten Klassen nicht verfügbar.
- **public** - Alle Variablen und Methoden sind überall verfügbar.

Diese drei Arten des Zugriffes können auch in Strukturen verwendet werden. Der einzige Unterschied besteht darin, daß in einer Struktur alles automatisch **public** ist, bis wir etwas anderes festlegen.

9.4 Was sind private Daten?

Beispielprogramm: VERERB2.CPP

In dem Beispielprogramm VERERB2.CPP sind die Daten der Basisklasse automatisch als **private** deklariert, da wir Zeile 6 auskommentiert haben. In diesem Programm sind diese Daten also in abgeleiteten Klassen nicht verfügbar. Die einzige Möglichkeit, diese Daten zu verwenden, stellen dann die Methoden der Basisklasse dar, was jetzt auch für abgeleitete Klassen gilt.

Es mutet ein wenig eigenartig an, Methoden aufrufen zu müssen, um auf Daten zugreifen zu können, die eigentlich Teil der abgeleiteten Klasse sind, aber C++ ist nun einmal so definiert. Das heißt, daß Du bei jeder Klasse ein wenig darüber nachdenken solltest, wie sie verwendet wird. Wenn Du glaubst, daß jemand eine Klasse von der Deinen ableiten wird, solltest Du die Daten als **protected** deklarieren, damit sie in der neuen Klasse einfach verwendet werden können. Die Zeilen 86 und 87 unseres Programmes sind jetzt nicht mehr erlaubt, da die Daten nicht sichtbar sind, also übernimmt Zeile 88 die Arbeit der beiden anderen, indem sie eine **public** Methode der Basisklasse aufruft. Auch Zeile 104 wurde aufgrund der versteckten Daten geändert. Kompiliere das Programm und führe es aus.

9.5 Versteckte Methoden

Beispielprogramm: VERERB3.CPP

Auch das Beispielprogramm VERERB3.CPP ist eine Wiederholung des ersten Programmes in diesem Kapitel mit einigen wenigen geringfügigen Änderungen.

Die abgeleiteten Klassen mit den Namen *Auto* und *Laster* haben das Schlüsselwort **public** nicht mehr vor dem Namen der Basisklasse in der ersten Zeile der jeweiligen Klassendeklaration. Wenn das Schlüsselwort **public** vor dem Namen der Basisklasse steht, erlaubt es, alle Methoden der Basisklasse in der abgeleiteten Klasse mit denselben Sicherheitseinschränkungen zu verwenden wie in der Basisklasse. Deshalb konnten wir auch im letzten Programm die Methoden, die als Teil der Basisklasse definiert waren, im Hauptprogramm aufrufen, obwohl wir wir mit einem Objekt einer der abgeleiteten Klassen gearbeitet haben.

In diesem Programm erben wir alle Elemente der Basisklasse als **private**, da vor dem Namen der Basisklasse das Schlüsselwort **private** steht. Deshalb sind sie außerhalb der abgeleiteten Klasse nicht verfügbar. Alle Elemente, die geerbt werden, fallen also unter zwei Zugriffsbeschränkungen, die Definitionen in der Basisklasse und die Einschränkungen bei der Ableitung. Es gilt jeweils die restriktivere der beiden. Das legt fest, wie die Elemente außerhalb der abgeleiteten Klasse verfügbar sind.

Alle Elemente werden in die abgeleitete Klasse mit denselben Zugriffsbeschränkungen übernommen, denen sie auch in der Basisklasse unterlegen sind, was ihre Sichtbarkein innerhalb der abgeleiteten Klasse selbst betrifft. In der Basisklasse verwenden wir jetzt wieder **protected** anstelle von **private**, deshalb sind die Elementvariablen innerhalb der abgeleiteten Klasse verfügbar.

In diesem Programm sind nur jene Methoden für Objekte der Klasse *Auto* verfügbar, die wir als Teil der Klasse selbst definiert haben. Deshalb können wir mit Objekten der Klasse *Auto* nur die Methoden mit den Namen *Initialisiere()* und *Passagiere()* verwenden.

Wenn wir ein Objekt der Klasse *Auto* deklarieren, enthält es der Definition von C++ gemäß drei Variablen, und zwar die eine von seiner Klasse definierte mit dem Namen *Passagieranzahl* und jene zwei, die Teil der Basisklasse sind, *Raeder* und *Gewicht*. Aufgrund des Schlüsselwortes **protected** in der Basisklasse können wir in den Methoden der abgeleiteten Klasse alle direkt verwenden. Die Variablen sind Teil eines Objektes der Klasse *Auto*, wenn wir es deklarieren und werden als solche gespeichert.

Einige der Bildschirmausgaben des Hauptprogrammes haben wir auskommentiert, da sie nicht mehr gültig oder sinnvoll sind. Die Zeilen 57 bis 59 mußten wir auskommentieren, da die Methoden *HoleGewicht()* und *Radlast()* nicht als Elemente der Klasse *Auto* verfügbar sind. Wir verwenden **private** bei der Vererbung. *Initialisiere()* ist immer noch verfügbar, es handelt sich aber um die Methode, die wir als Teil der Klasse *Auto* selbst definieren, nicht jene aus der Klasse *Vehikel*.

9.6 Wir verwenden die Laster-Klasse

Wenn wir uns nun ansehen, wie die Klasse *Laster* im Hauptprogramm verwendet wird, sehen wir, daß die Zeilen 63 und 65 aus obigem Grund auskommentiert wurden. Die Zeilen 66 und 67 allerdings haben wir aus einem anderen Grund auskommentiert. Zwar ist die Methode mit dem Namen *Effizienz()* verfügbar und kann als Element der Klasse *Laster* aufgerufen werden, wir können sie aber doch nicht verwenden, da keine Möglichkeit besteht, die Variablen *Raeder* und *Gewicht* des Objektes der Klasse *Laster* zu initialisieren. Wir können das *Gewicht* des *Laster* auslesen, wie wir es in Zeile 102 tun, aber da wir die Variable nicht initialisieren können, ist das Ergebnis bedeutungslos und wir haben die Zeilen 66 und 67 auskommentiert.

Private Vererbung ist der Verwendung eines eingebetteten Objektes sehr ähnlich und wird praktisch nicht verwendet. Bis Du sehr viel Erfahrung mit C++ und objektorientiertem Programmieren überhaupt gesammelt hast, solltest Du ausschließlich **public** bei der Vererbung verwenden. Es gibt wahrscheinlich keinen guten Grund, **private** oder **protected** überhaupt jemals anzuwenden. Vermutlich existieren sie nur, damit die Sprache auch komplett ist.

Kompiliere das Programm und führe es aus. (Amen.) Du solltest einige der auskommentierten Zeilen aus-kommentieren (d.h. die Kommentare entfernen), um zu sehen, mit welchen Fehlern sich der Compiler meldet.

9.7 Alle Daten initialisieren

Beispielprogramm: VERERB4.CPP

Beim Durchsehen des Beispielprogrammes VERERB4.CPP wirst Du feststellen, daß wir das Initialisierungsproblem des letzten Kapitels beiseite geschafft haben. Wir haben auch jeder Klasse Standardkonstruktoren hinzugefügt, wir können also studieren, wie

diese im Zusammenhang mit Vererbung verwendet werden. Außerdem sind wir bei der Vererbung zu **public** zurückgekehrt.

Beim Erzeugen eines Objektes der Basisklasse *Vehikel* haben wir kein Problem, da Vererbung hier keine Rolle spielt. Der Konstruktor der Basisklasse funktioniert genauso wie es alle Konstruktoren bisher getan haben. Bei der Erzeugung von unserem *Hochrad* in Zeile 47 verwenden wir den Standardkonstruktor und das Objekt wird mit den Werten des Konstruktors initialisiert. Die Zeile 49 haben wir auskommentiert, da wir keinen Initialisierungscode für das Objekt mehr brauchen.

Etwas anders stellt sich die Situation dar, wenn wir ein Objekt einer der abgeleiteten Klassen deklarieren, wie in Zeile 57 den *Sedan*. Wir müssen nicht nur einen Konstruktor für die abgeleitete Klasse aufrufen, sondern haben uns auch darum zu kümmern, wie wir den Konstruktor der Basisklasse aufrufen können. Das ist aber eigentlich kein Problem, da der Compiler automatisch den Standardkonstruktor der Basisklasse aufruft, solange wir nicht explizit einen (anderen) Konstruktor für die Basisklasse aufrufen. Im nächsten Beispielprogramm wollen wir gerade das tun, jetzt aber lassen wir es beim Standardkonstruktor für die Basisklasse, der automatisch aufgerufen wird, bewenden.

9.8 Reihenfolge der Konstruktion

Das nächste Problem, das sich uns stellt, ist das der Reihenfolge der Konstruktion. Die Antwort darauf merkt man sich relativ leicht: „C++ Klassen ehren ihre Eltern, indem sie deren Konstruktor vor ihren eigenen aufrufen“. Der Konstruktor der Basisklasse wird vor dem der abgeleiteten Klasse aufgerufen. Das macht Sinn, da es sicherstellt, daß die Basisklasse fertig konstruiert ist, wenn der Konstruktor für die abgeleitete Klasse abgearbeitet wird. Das erlaubt Dir, einige der Daten der Basisklasse während der Konstruktion der abgeleiteten Klasse schon zu verwenden. In unserem Fall wird der Teil von *Sedan*, der ein *Vehikel* ist, zuerst konstruiert, dann werden die lokalen Elemente des Objektes *Sedan* konstruiert, schlußendlich sind alle Elementvariablen initialisiert. Deshalb können wir die Methode *Initialisiere()* in Zeile 59 auskommentieren. Wir benötigen sie nicht mehr.

Auch das Objekt *Sattelschlepper* wird in Zeile 66 in derselben Weise konstruiert. Der Konstruktor für die Basisklasse wird abgearbeitet, dann der Konstruktor für die abgeleitete Klasse. Jetzt ist das Objekt komplett definiert und wir können es mit den Standardwerten in allen Elementvariablen verwenden. Die Zeilen 68 und 69 brauchen wir also nicht mehr.

Der Rest des Programmes sollte, die Reihenfolge bei der Abarbeitung der Destruktoren ausgenommen, keine Probleme aufwerfen.

9.9 Wie werden die Destruktoren abgearbeitet?

Wenn die Objekt ihre Gültigkeitsbereiche verlassen, müssen auch ihre Destruktoren abgearbeitet werden und da wir keine definiert haben, werden die Standarddestruktoren verwendet. Auch hier ist das Löschen des Objektes der Basisklasse, *Hochrad*, kein Problem. Der Destruktor wird abgearbeitet und das Objekt ist nicht mehr. Das Objekt

Sedan allerdings muß schon zwei Destruktoren abarbeiten, um alle seine Teile, den einen der Basisklasse und den anderen der abgeleiteten Klasse, zu zerstören. Es sollte Dich nicht zu sehr wundernehmen, daß die Destruktoren für dieses Objekt in umgekehrter Reihenfolge zu den Konstruktoren abgearbeitet werden. Der Destruktor der abgeleiteten Klasse wird zuerst ausgeführt, dann der Destruktor für die Basisklasse und das Objekt damit zerstört.

Bedenke: wann immer ein Objekt definiert oder erzeugt wird, muß für jeden seiner Teile ein Konstruktor ausgeführt werden. Genauso muß für jeden Teil ein Destruktor ausgeführt werden, wenn das Objekt seinen Gültigkeitsbereich verläßt.

Nach Deinem eingehenden Studium solltest Du dieses Programm kompilieren und ausführen.

9.10 Vererbung, wenn wir Konstruktoren verwenden

Beispielprogramm: VERERB5.CPP

VERERB5.CPP ist eine weitere Variation des Themas, jetzt mit Konstruktoren, die mehr sind als die bloßen Standardkonstruktoren. Jede der Klassen hat ihren eigenen Konstruktor. Der zusätzliche Konstruktor der Klasse *Vehikel* in den Zeilen 12 bis 14 ist nichts Besonderes, so etwas wie all die anderen Konstruktoren, die wir in dieser Einführung schon kennengelernt haben. Wir verwenden ihn in Zeile 59 des Hauptprogrammes, wo wir das Objekt *Hochrad* mit zwei Werten definieren.

Der Konstruktor für die Klasse *Auto* ist ein bißchen anders, da wir drei Variablen übernehmen. Einer der Werte, *Leute*, wird in der abgeleiteten Klasse selbst verwendet, um die Elementvariable mit dem Namen *Passagieranzahl* zu initialisieren. Die anderen beiden Werte allerdings müssen wir irgendwie an die Basisklasse übergeben, um die Werte von *Raeder* und *Gewicht* zu initialisieren. Das erreichen wir mit einer Elementinitialisierung, wie wir in diesem Konstruktor zeigen. Der Doppelpunkt am Ende der Zeile 28 bedeutet, daß eine Liste von Elementinitialisierungen folgt. Alles, was zwischen dem Doppelpunkt und der offenen geschwungenen Klammer des Konstruktors folgt, sind Elementinitialisierungen. Die erste passiert in Zeile 29 und sieht aus wie ein Konstruktoraufruf an die Klasse *Vehikel*, der zwei Parameter verlangt. Genau das ist sie auch und ruft den Konstruktor der Klasse *Vehikel* auf, um diesen Teil des Objektes *Seden* (als Beispiel) zu initialisieren. So können wir kontrollieren, welcher Konstruktor der Basisklasse aufgerufen wird, wenn wir ein Objekt der abgeleiteten Klasse erzeugen.

Die nächste Elementinitialisierung in Zeile 30 funktioniert in etwa wie ein Konstruktor für eine einzelne Variable. Indem wir den Namen der Variable angeben und in Klammern einen Wert des korrekten Typs wird der Variable dieser Wert zugewiesen obwohl die Variable keine Klasse, sondern ein einfacher vordefinierter Typ ist. Diese Technik kann verwendet werden, um alle Elemente der abgeleiteten Klasse oder Teile davon zu initialisieren. Sind alle Elementinitialisierungen durchgeführt, wird der Code in den geschwungenen Klammern ausgeführt. In diesem Fall befindet sich dort kein Code. Wäre dort jedoch einer, wäre er genauso geschrieben, wie dies für alle Konstruktoren gilt.

9.11 Wie sieht es mit der Reihenfolge der Ausführung aus?

Es mag eigenartig erscheinen, aber die Elemente der Elementinitialisierungsliste (...) werden nicht in der Reihenfolge abgearbeitet, in der sie aufgeführt sind. Zuerst werden die Konstruktoren für die beerbte Klasse ausgeführt, in der Reihenfolge ihrer Deklaration im Klassenkopf. Bei der Verwendung von Vielfachvererbung können mehrere Klassen in der Kopfzeile erscheinen, in diesem Programm verwenden wir aber nur eine. Dann werden die Elementvariablen initialisiert, aber nicht in der Reihenfolge, in der sie in der Liste erscheinen, sondern in der Reihenfolge ihrer Deklaration in der Klasse. Schließlich wird der Code des Konstruktors selbst ausgeführt, sofern ein solcher Code existiert.

Es gibt einen guten Grund für diese befremdend anmutende Ordnung. Die Destruktoren müssen in umgekehrter Reihenfolge der Konstruktoren aufgerufen werden, wenn aber zwei Konstruktoren mit unterschiedlicher Reihenfolge der Konstruktion definiert sind, welcher soll dann die Reihenfolge der Destruktion bestimmen? Die korrekte Antwort ist: keiner. Das System verwendet die Ordnung der Deklaration für die Konstruktion und dreht sie für die Destruktion um.

Die Klasse *Laster* initialisiert zum einen die Basisklasse und zum anderen zwei Elementvariablen, *Passagieranzahl* und *Ladung*. Der Konstruktor selbst enthält keinen Code, so wie schon bei der Klasse *Auto*.

Wir verwenden die beiden Konstruktoren für die Klassen *Auto* und *Laster* in den Zeilen 69 und 78 für ein Objekt der Klasse *Auto* und eines der Klasse *Laster*.

Der Rest des Programmes sollte kein Problem mehr darstellen. Kompiliere dieses Programm und führe es aus, bevor Du weitergehst.

9.12 Zeiger auf ein Objekt und ein Array von Objekten

Beispielprogramm: VERERB6.CPP

Die Datei VERERB6.CPP zeigt ein Beispiel für die Verwendung eines Arrays von Objekten und eines Zeigers auf ein Objekt. In diesem Programm sind die Objekte Instanzen einer abgeleiteten Klasse und der Sinn und Zweck dieses Programmes ist, zu zeigen, daß nichts Besonderes an einer abgeleiteten Klasse ist. Eine Klasse verhält sich immer gleich, sei sie nun Basisklasse oder abgeleitete Klasse.

Diese Programm ist identisch mit dem ersten in diesem Kapitel bis wir zum Hauptprogramm kommen. Dort finden wir einen Array von drei Objekten der Klasse *Auto* in Zeile 53. Es ist offensichtlich, daß jede Operation, die für ein einzelnes Objekt erlaubt ist, auch für ein Objekt eines Arrays gestattet ist. Wir müssen dem System nur immer mitteilen, an welchem Objekt des Arrays wir interessiert sind, indem wir den Index des Objektes in eckigen Klammern anführen, wie wir es in den Zeilen 58 bis 64 tun. Was hier geschieht, sollte Dir mittlerweile geläufig sein, wir gehen also weiter zum nächsten interessanten Konstrukt.

In Zeile 68 deklarieren wir kein Objekt der Klasse *Laster*, sondern einen Zeiger auf ein Objekt der Klasse *Laster*. Damit wir den Zeiger verwenden können, müssen wir ihm etwas geben, auf das er zeigen kann, was wir in Zeile 70 tun, indem wir dynamisch ein

Objekt erzeugen. Sobald der Zeiger auf etwas zeigt, können wir das Objekt genauso wie jedes andere auch verwenden, wir müssen nur die Zeigernotation verwenden, um auf die Methoden des Objektes zuzugreifen. Das zeigen wir in den Zeilen 76 bis 80, und auch im Beispielprogramm des Kapitel 13 dieser Einführung.

Schließlich löschen wir das Objekt in Zeile 81. Du solltest Dich mit diesem Programm befassen, bis Du das neue Material verstanden hast und das Programm dann kompilieren und ausführen.

9.13 Die neue Zeit-Klasse

Wir haben eine Serie von nicht-trivialen Klassen in Kapitel 6 mit einer *Datum* Klasse begonnen, es kamen eine *Zeit* Klasse und schließlich eine *DatumNeu* Klasse hinzu. Jetzt ist die Reihe an Dir, diese Serie zu erweitern. Deine Aufgabe ist es, eine Klasse *ZeitNeu* zu entwickeln, die von der Klasse *Zeit* abgeleitet ist und eine neue Elementvariable mit dem Namen *SekundenHeute* hinzufügt sowie eine Methode, die die Sekunden, die seit Mitternacht vergangen sind, berechnet.

9.14 Programmieraufgaben

1. Entferne die Kommentare aus den Zeilen 57 bis 59 von VERERB3.CPP und sieh Dir an, welche Fehler der Compiler ausgibt.

2. Erweitere jeden der Konstruktoren in VERERB4.CPP um cout Anweisungen, damit Du die Reihenfolge der Abarbeitung verfolgen kannst.

10 Mehrfachvererbung und Ausblick

Mehrfachvererbung ist die Möglichkeit, Daten und Methoden aus mehr als nur einer Basisklasse in eine abgeleitete Klasse zu übernehmen. Mehrfachvererbung und einige andere neuere Erweiterungen der Sprache werden wir in diesem Kapitel besprechen, dazu einige der möglichen zukünftigen Entwicklungen.

Nachdem Du mit dieser Einführung fertig bist, solltest Du über genug Erfahrung mit C++ verfügen, um neue Konstrukte selbst zu studieren, wenn die Compiler-Autorinnen sie implementieren.

10.1 Mehrfachvererbung

Eine bedeutende Erweiterung von C++ war die Möglichkeit, beim Erstellen einer neuen Klasse Methoden und Variablen von zwei oder mehr Elternklassen zu übernehmen. Das ist Mehrfachvererbung, die von vielen als wesentlich für eine objektorientierte Programmiersprache angesehen wird. Es war aber nicht leicht, ein gutes Beispiel für Mehrfachvererbung zu finden, das Ergebnis kann auch nicht gut genannt werden, es tut nichts sinnvolles. Aber es illustriert die Verwendung von Mehrfachvererbung in C++, und das soll einstweilen unser primäres Anliegen sein.

Das größte Problem der Mehrfachvererbung stellt das Übernehmen von Variablen oder Methoden mit demselben Namen aus verschiedenen Elternkalassen dar. Welche der gleichnamigen Variablen oder Methoden soll verwendet werden? Das wollen wir in den nächsten Beispielprogrammen zeigen.

10.2 Simple Mehrfachvererbung

Beispielprogramm MEHRVER1.CPP

Schon der ersten Blick auf MEHRVER1.CPP wird Dir enthüllen, daß wir in den Zeilen 4 und 22 zwei einfache Klassen definieren, *FahrenderLastwagen* und *Fahrerin*.

Um das Programm möglichst einfach zu halten, haben wir alle Methoden **inline** definiert. Somit findet sich der Code für die Methoden schnell. Alle Variablen in beiden Klassen haben wir als **protected** deklariert, sie sind also in allen abgeleiteten Klassen verfügbar. Den Code selbst haben wir möglichst einfach gehalten, um uns auf das Studium der Schnittstelle konzentrieren zu können anstatt uns mit komplexen und -izierten Methoden herumschlagen zu müssen. [Lieber schon schlagen wir uns mit komplexierten Sätzen (herum)!] Wie wir schon festgestellt haben, wird sich das Kapitel 12 dann mit nicht-trivialen Methoden befassen.

In Zeile 32 beginnen wir die Definition einer weiteren Klasse mit dem Namen *GefahrenerLaster*, die alle Daten und alle Methoden der beiden zuvor definierten Klassen übernimmt. In den letzten beiden Kapiteln haben wir uns damit befaßt, wie wir eine Klasse beerben. Das ableiten einer Klasse aus zwei oder mehreren Klassen geschieht im Grunde genauso, nur daß wir eine Liste von Elternklassen, die durch Beistriche getrennt sind, verwenden, wie etwa in Zeile 32. Du hast sicherlich bemerkt, daß wir vor den Klassennamen das Schlüsselwort **public** verwenden, um die Methoden der Elternklasse in der Subklasse frei verwenden zu können. In diesem Fall haben wir keine neuen Variablen definiert, führen mit der Subklasse aber zwei neue Methoden ein, in den Zeilen 35 bis 42.

Wir definieren ein Objekt mit dem Namen *MercedesBMW*, was bedeuten soll, daß jemand mit dem Namen Mercedes einen BMW Lastwagen fährt. Das Objekt *MercedesBMW* besteht aus vier Variablen, drei von der Klasse *FahrenderLastwagen* und die eine der Klasse *Fahrerin.* Alle vier Variablen können wir in jeder der Methoden der Klasse *GefahrenerLaster* in derselben Weise manipulieren wie in einer Klasse, die nur von einer Basisklasse abgeleitet ist. Einige Beispiele findest Du in den Zeilen 50 bis 59 des Hauptprogrammes und es sollte Dir ein Leichtes sein, weitere Bildschirmausgaben hinzuzufügen.

Alle Regeln, die wir bei der einfachen Vererbung für **private** und **protected** Variablen und **public** und **private** Methodenübernahme aufgestellt haben, gelten auch für die Mehrfachvererbung.

10.3 Doppelte Methodennamen

Wie Dir nicht entgangen sein wird, haben beide Elternklassen eine Methode mit dem Namen *Initialisiere()* und wir übernehmen beide in die Subklasse ohne Probleme. Sobald wir aber versuchen sollten, an eine der beiden eine Nachricht zu senden, haben wir ein Problem, da das System nicht weiß, welche der beiden Methoden wir meinen. Diese Schwierigkeit werden wir uns im nächsten Beispielprogramm ansehen und auch eine Lösung haben wir natürlich schon im Talon.

Wir haben im Hauptprogramm keine Objekte der Elternklassen definiert. Da es sich bei den beiden Elternklassen aber auch nur um ganz normale Klassen handelt, sollte es offensichtlich sein, daß sie auch als solche verwendet werden können. Du kannst das ruhig machen, um Deine Kenntnisse auf diesem Gebiet zu überprüfen.

Kompiliere dieses Programm und führe es aus, wenn Du es komplett verstanden hast.

10.4 Mehr über doppelte Methodennamen

Beispielprogramm: MEHRVER2.CPP

Das zweite Beispielprogramm dieses Kapitels, MEHRVER2.CPP, illustriert die Verwendung von Klassen, die Methoden mit demselben Namen aus verschiedenen Elternklassen übernehmen.

Wir haben jeder der drei Klassen eine Methode mit dem Namen *KostenProGanzem-Tag()* hinzugefügt. Damit wollen wir zeigen, wie derselbe Methodenname in allen Klassen verwendet werden kann. Die Klassendefinitionen stellen kein Problem dar, die Methoden werden einfach wie gehabt definiert. Ein Problem stellt sich uns jedoch, wenn wir eine der Methoden verwenden wollen, haben sie doch alle denselben Namen, dieselbe Anzahl an und dieselben Typen von Parametern und auch denselben Rückgabetyp. Dadurch kann keine Überladungsregeln entscheiden, welche der Methoden verwendet werden soll.

Wie wir eindeutig festlegen, welche Methode wir meinen, zeigen die Methodenaufrufe in den Zeilen 63, 67 und 71 des Hauptprogrammes. Wir geben den Klassennamen vor dem Methodennamen an, durch einen zweifachen Doppelpunkt von diesem getrennt, also so wie in der Implementation der Methode. Das wird auch Qualifikation des Methodennamen genannt. In Zeile 71 wäre diese Vorgangsweise nicht notwendig, da es sich um die Methode der abgeleiteten Klasse handelt, die Vorrang vor den Elternklassen hat. Du könntest durchaus auch alle Methodennamen qualifizieren, wenn die Namen aber eindeutig sind, übernimmt der Compiler diese Arbeit für Dich, was den Code einfacher zu schreiben und lesen macht.

Kompiliere dieses Programm, führe es aus und sieh Dir das Ergebnis an.

10.5 Wir vergehen uns an einigen Prinzipien des OOP

Ein Grundpfeiler objektorientierten Programmierens ist, daß man bei der Vererbung einer Basisklasse in eine abgeleitete Klasse implizit annehmen kann, die abgeleitete Klasse „ist eine“ Art der Basisklasse, mit Betonung auf „ist eine“. Es wäre nicht besonders sinnvoll (vielleicht aber originell und interessant), die Charakteristika eines Eisbären in eine Klasse für Hochhäuser zu übernehmen, da wir nicht sagen können „Ein Hochhaus ist ein Eisbär.“ Da aber „Ein Hochhaus ist ein Gebäude“ sehr wohl seine Berechtigung hat, wäre es sinnvoll, die Charakteristika eines Gebäudes in einer Hochhausklasse zu übernehmen und diesen die für Hochhäuser kennzeichnenden Eigenschaften hinzuzufügen.

Unser Beispiel paßt also nicht wirklich in dieses Schema („Ein gefahrener Laster ist eine Fahrerin“?), allerdings ist es doch relativ einfach, das Konzept zu erfassen. Auch hier hat unser Interesse wieder mehr der Technik gegolten als gutem objektorientiertem Programmieren (??). Du kannst alle möglichen „richtigen“ und „falschen“ Arten der Vererbung später studieren. Zunächst wollen wir uns einmal das Handwerk aneignen.

10.6 Doppelte Variablennamen

Beispielprogramm: MEHRVER3.CPP

Beide Basisklassen in MEHRVER3.CPP haben eine Variable mit dem gleichen Namen.

Gemäß den Regeln der Vererbung hat ein Objekt der Klasse *GefahrenerLaster* zwei Variablen mit dem Namen *Gewicht*. Das wäre ein Problem, würde der Definition von C++ nicht einen Weg enthalten, auf jede einzeln zuzugreifen. Wie Du vielleicht erraten hast, verwenden wir die Qualifikation, um die Variablen zu benutzen. Es sollte auch noch gesagt werden (obwohl es auf der Hand liegt), daß kein Grund besteht, nicht auch

der abgeleitete Klasse eine Variable mit wieder demselben Namen zu verpassen. Um diese zu verwenden, wäre dann keine Qualifizierung notwendig, aber mit dem Namen der abgeleiteten Klasse natürlich möglich.

Hast Du also einmal die einfache Vererbung verstanden, ist Vielfachvererbung nichts weiter als eine Ausdehnung der gleichen Gesetze und Regeln. Wenn Du Variablen oder Methoden mit demselben Namen übernimmst, mußt Du dem Compiler mittels der Qualifizierung die richtige mitteilen.

Die Konstruktoren der beiden ererbten Klassen werden vor dem Konstruktor der abgeleiteten Klasse ausgeführt. Die Konstruktoren der Basisklassen werden in der Reihenfolge ihrer Deklaration in der header-Datei aufgerufen.

10.7 Praktische Mehrfachvererbung

Beispielprogramm: ZEITDAT.H

Die Datei ZEITDAT.H gibt ein praktisches Beispiel für Mehrfachvererbung. Wir kehren zu den uns schon wohlbekannten Klassen *DatumNeu* und *Tageszeit* zurück.

Du kannst von dieser kleinen header-Datei einiges lernen, da es unser erstes Beispiel für Elementinitialisierung bei Mehrfachvererbung ist. Für diese Klasse existieren zwei Konstruktoren. Der erste ist sehr simpel und tut selbst überhaupt nichts, wie Du aus Zeile 13 ersehen kannst. Es werden also die Standardkonstruktoren für die Klassen *DatumNeu* und *Tageszeit* ausgeführt. In beiden Fällen wird der Konstruktor verwendet, der keine Parameter verlangt. Ein solcher existiert naturgemäß für beide Klassen.

Der zweite Konstruktor ist da schon interessanter, da er nicht einfach die Standardkonstruktoren verwendet, sondern einige seiner Parameter an die Konstruktoren der ererbten Klassen weitergibt. Nach dem Doppelpunkt in Zeile 14 stehen zwei Elementinitialisierer, mit denen wir Elemente der Klasse initialisieren. Da die beiden Elternklassen vererbt wurden, sind auch sie Elemente dieser Klasse und können initialisiert werden, wie wir es zeigen. Beide Elementinitialisierungen sind Aufrufe eines Konstruktors der Basisklasse und es ist offensichtlich, daß ein Konstruktor für die jeweilige Basisklasse mit einer korrespondierenden Anzahl und denselben Typen von Parametern existieren muß. In Zeile 15 rufen wir eigentlich den Standardkonstruktor der Klasse *DatumNeu* auf, da wir keine Parameter angeben. Wir könnten auch einfach das System den Standardkonstruktor aufrufen lassen, wenn wir dies aber selbst tun, sehen wir gleich, was passiert.

Nach den Elementinitialisierungen wird der normale Code des Konstruktors der abgeleiteten Klasse ausgeführt. Dieser findet sich in unserem Beispiel in Zeile 17.

10.8 Reihenfolge der Elementinitialisierung

Die Reihenfolge der Initialisierung der Elemente scheint ein wenig eigenartig zu sein, folgt aber einigen bestimmten Regeln. Du wirst Dich erinnern, wie wir im letzten Kapitel festgestellt haben, daß die Reihenfolge nicht der in der Liste folgt, sondern einer anderen strikten Reihenfolge, über die Du volle Kontrolle hast. Alle geerbten Klassen werden

zunächst in der Reihenfolge, in der sie im Klassenkopf aufgeführt sind, initialisiert. Wären die Zeilen 15 und 16 vertauscht, würde die Klasse *DatumNeu* immer noch als erste initialisiert werden, da sie in Zeile 8 als erste erscheint. Wir haben erwähnt, daß C++-Klassen ihre Ahnen ehren und ihre Eltern vor sich selbst initialisieren. Das sollte eine ganz nette Eselsbrücke sein.

Dann werden alle lokalen Klassenelemente, so es solche gibt, initialisiert. Dabei folgt C++ wieder der Reihenfolge, in der sie in der Klasse deklariert sind, und nicht der der Initialisierungsliste.

Schließlich, nachdem alle Elementinitialisierungen in der richtigen Reihenfolge ausgeführt worden sind, wird der Code des Konstruktors selbst ganz normal abgearbeitet.

10.9 Wir verwenden die neue Klasse

Beispielprogramm: VERWZD.CPP

Das Beispielprogramm mit dem Namen VERWZD.CPP verwendet die Klasse *DatumZeit*, die wir gerade erstellt haben. Auch hier ist das Hauptprogramm wieder möglichst simpel gehalten. Für das Objekt mit dem Namen *Jetzt* verwenden wir den Standardkonstruktor, für die Objekte *Geburtstag* und *Spezial* den Konstruktor mit den Elementinitialisierungen.

Du solltest beim Verständnis des übrigen Codes keinerlei Schwierigkeiten haben. Du wirst feststellen, daß die Klasse, haben wir sie einmal fertiggestellt, sehr einfach zu verwenden ist.

10.10 Klassenschablonen

Bei der Entwicklung eines Programmes kommt es oft vor, daß man eine Operation auf mehr als einen Datentyp anwenden will. Zum Beispiel willst Du eine Liste von **int** Variablen, eine von **float** Variablen und eine mit Zeichenketten sortieren. Es ist nicht besonders sinnvoll, für alle drei Listen eigene Routinen zu schreiben, wenn die Sortierlogik immer dieselbe ist. Mit Klassenschablonen bist Du aus dem Schneider, da Du eine Funktion schreiben kannst, die imstande ist, alle drei Listen zu sortieren.

In der Ada Programmiersprache ist dieses Konzept schon länger verwirklicht. Die Softwareindustrie hat fertige, fehlerbereinigte Routinen entwickelt, die mit vielen verschiedenen Datentypen arbeiten. Im Zuge der allgemeinen Verfügbarkeit dieser Technik in C++ wird dies auch für diese Programmiersprache bald der Fall sein. Es wird dann Routinen zum sortieren, für Stapel, Schlangen usw. geben. Eine solche Bibliothek ist im Rahmen des ANSI-C++ Standards schon verfügbar: die STL - Standard Template Library. Das Studium dieser Bibliothek übersteigt den Rahmen dieser Einführung, aber es ist sicherlich sinnvoll, sich diese und ihre Verwendung einmal genauer anzusehen.

Die meisten Autoren von Compilern haben die Möglichkeit, Schablonen zu verwenden, in ihren neueren Compilern implementiert. Die nächsten drei Beispielprogramme werden die Verwendung von Schablonen gemäß ANSI-C++ Standard demonstrieren. Da noch nicht alle Compiler Schablonen unterstützen, kann es sein, daß sie sich (noch) nicht

kompilieren lassen. Schlußendlich werden aber alle Compiler diesem Standard folgen müssen, um konkurrenzfähig zu bleiben. Du solltest die Beispiele auf jeden Fall studieren.

10.11 Die erste Schablone

Beispielprogramm: SCHABL1.CPP

Das Programm SCHABL1.CPP ist unser erstes Beispiel für die Verwendung einer Schablone. Dieses Programm ist so einfach, daß wir eigentlich nicht viel darüber sagen müssen, aber es illustriert die Verwendung des Typenparameters.

Die Schablone steht in den Zeilen 4 bis 8. Die erste Zeile zeigt uns (und dem Compiler), daß ein Typ ersetzt werden soll, nämlich der Typ JEDER_TYP. Dieser Typ kann von jedem anderen ersetzt werden, der mit der Vergleichsoperation in Zeile 7 verwendet werden kann. Wenn Du eine Klasse definiert und den Operator ¿ überladen hast, kannst Du die Schablone mit Objekten dieser Klasse verwenden. Du mußt also nicht extra eine Funktion *Maximum()* für jeden Typen oder jede Klasse in Deinem Programm schreiben.

Diese Funktion wird automatisch für jeden Typen, mit dem sie im Programm aufgerufen wird, verfügbar. Der Code selbst ist glaube ich nicht der schwierigste.

Es wird Dir vielleicht aufgefallen sein, daß Du einen ganz ähnlichen Effekt auch mit einem Makro erzielen kannst. Bei einem Makro wird aber keine Typenüberprüfung durchgeführt. Daher und weil in C++ die **inline** Deklaration verfügbar ist, werden Makros von C++ Programmiererinnen (zum Glück!) so gut wie nie verwendet.

10.12 Eine Klassenschablone

Beispielprogramm: SCHABL2.CPP

Das Beispielprogramm in Datei SCHABL2.CPP ist etwas komplizierter, da es eine Schablone für eine ganze Klasse anstatt für eine einzelne Funktion beinhaltet. Der Code der Schablone steht in den Zeilen 6 bis 16 und ein genaueres Hinsehen zeigt, daß es sich dabei um eine komplette Klassendefinition handelt. Du hast recht, wenn Du meinst, der Stapel ist schwach bis sehr schwach, da nichts den Zugriff auf einen leeren Stapel verhindert und nichts einen vollen anzeigt. Unsere Absicht war es aber wieder, die Verwendung von Typenparametern (anhand einer möglichst simplen Klasse) zu illustrieren.

In Zeile 25 erzeugen wir ein Objekt mit dem Namen *intStapel*, einen Stapel für Variablen vom Typ **int** und ein zweites Objekt mit dem Namen *floatStapel* zum Speichern von Variablen vom Typ **float**. In beiden Fällen geben wir den Typ, mit dem das Objekt arbeiten soll, in „¡¡“ Klammern an. Das System erzeugt dann das Objekt indem es zuerst alle Vorkommen von *JEDER_TYP* durch den angegebenen Typ ersetzt und dann ein Objekt diesen Typs erzeugt. Es kann jeder Typ verwendet werden, dem etwas zugewiesen werden kann, da wir dies mit dem Typenparameter in den Zeilen 13 und 14 tun.

Obwohl alle Zeichenketten verschieden lang sind, können wir unseren Stapel sogar zum Speicher von Zeichenketten verwenden, wenn wir nur Zeiger auf die Zeichenketten und

nicht die ganze Zeichenkette selbst speichern. Das zeigen wir mit dem Objekt *charStapel*, das wir in Zeile 27 definieren und später dann auch verwenden.

Wenn Du Dich ein wenig mit diesem Programm beschäftigst, sollte Dir die Funktionsweise bald vertraut und klar sein. Du solltest es dann kompilieren und ausführen, wenn Dein Compiler mit Schablonen umgehen kann.

10.13 Wiederverwertung der Stapelklasse

Beispielprogramm: SCHABL3.CPP

Das Programm mit dem Namen SCHABL3.CPP verwendet die gleiche Klasse als Schablone, die wir im letzten Programm definiert haben, aber als Typen der Stapelelemente verwendet sie die Klasse *Datum*. Genauer gesagt verwendet sie einen Zeiger auf ein Objekt der Klasse *Datum*.

Du kannst auch die Klasse selbst im Stapel speichern und nicht nur einen Zeiger auf sie. Das wäre aber extrem ineffizient, da bei jeder Interaktion mit dem Stapel die gesamte Klasse in jenen hineinkopiert oder wieder herauskopiert würde. Die Verwendung eines Zeigers wird also bevorzugt und deshalb haben auch wir das getan.

Die drei letzten Beispielprogramme kannst Du kompilieren (und ausführen), wenn Ihr Compiler Schablonen unterstützt. Typenparameter sind Teil der C++ Spezifikationen und die meisten neueren Compiler unterstützen sie.

10.14 Was soll mein nächster Schritt sein?

Wieder haben wir einen Eckpfeiler des Programmierens in C++ aufgestellt. Mit dem Wissen um die Vererbungslehre hast Du nahezu alles, was Du benötigst, um die objektorientierten Konzepte von C++ effektiv einzusetzen. Du solltest das Studium also wieder einmal Studium sein lassen und zu programmieren beginnen. Was wir noch nicht behandelt haben, sind die virtuellen Funktionen, denen wir die nächsten beiden Kapitel widmen werden.

11 Virtuelle Funktionen

In diesem Kapitel betreten wir wieder völliges Neuland mit einer neuen Sprache. Wenn objektorientiertes Programmieren noch (relativ) neu für Dich ist, solltest Du dieses Kapitel sehr sorgfältig durchgehen. Wir haben versucht, die Details dieses neuen und ein wenig einschüchternden Themas klar darzulegen. Solltest Du aber schon Erfahrung im objektorientierten Programmieren besitzen und nur C++ als Programmiersprache erlernen wollen, kannst Du die ersten vier Beispielprogramme getrost überspringen und gleich zum Programm mit dem Namen VIRTL5.CPP gehen.

Ein Ausdruck, der dringend einer Definition bedarf, ist Polymorphismus. Auf deutsch heißt das soviel wie „Vielgestaltigkeit“, im Kontext des OOP aber eher „Ähnlichkeit“. Objekte sind polymorph, wenn sie sich zwar ähnlich, aber eben doch verschieden sind. Was das genau heißt, werden wir im Laufe dieses Kapitels feststellen.

Wir haben in dieser Einführung schon das Überladen von Operatoren und Funktionen studiert. Das ist in beiden Fällen eine einfache Form von Polymorphismus, da sich eine Einheit auf zwei oder mehrere Dinge bezieht. Die Verwendung von virtuellen Funktionen kann bei gewissen Programmieraufgaben eine große Hilfe sein, wie Du in den nächsten beiden Kapiteln sehen wirst.

11.1 Ein einfaches Programm mit Vererbung

Beispielprogramm: VIRTL1.CPP

Das Programm VIRTL1.CPP soll Ausgangspunkt unserer virtuellen Tour sein. Da dieses Programm nichts mit virtuellen Funktionen zu tun hat, mag der Name ein wenig in die Irre führen. Wir haben es so genannt, weil es Teil einer Serie von Programmen ist, die die Verwendung von virtuellen Funktionen illustrieren soll. Das letzte Programm dieses Kapitels wird dann die „richtige“ Verwendung von virtuellen Funktionen zeigen.

Das erste Programm ist sehr einfach und wie Du sicher bemerkt hast, ist es einem Programm, das wir früher in dieser Einführung besprochen haben, ähnlich. Allein, es ist radikal vereinfacht, damit wir uns auf die virtuellen Funktionen konzentrieren können. Die meisten der Funktionen des letzten Kapitels haben wir aus diesem Grund komplett weggelassen, und eine neue Methode haben wir der Elternklasse in Zeile 9 hinzugefügt: *Nachricht()*. Das Studium dieser Funktion in der Basisklasse und den abgeleiteten Klassen wird uns das gesamte Kapitel hindurch begleiten. Aus diesem Grund gibt es eine Methode mit dem Namen *Nachricht()* auch in der Klasse *Auto* und in der neuen Klasse mit dem Namen *Boot*, die wir in den Zeilen 30 bis 36 definieren.

Dir ist sicherlich aufgefallen, daß die Klasse *Laster* keine Methode mit dem Namen *Nachricht()* hat. Das ist natürlich mit Absicht geschehen (haha), um die Verwendung

einer virtuellen Methode (oder, wenn Dir das besser gefällt, einer virtuellen Funktion) illustrieren zu können. Da die Methoden der Basisklasse mit dem Schlüsselwort **public** übernommen werden, ist die Methode mit dem Namen *Nachricht()* der Basisklasse in der Klasse Laster verfügbar.

Die Methode *Nachricht()* wurde auch absichtlich so einfach gehalten. Wir sind eben an virtuellen Funktionen interessiert und nicht an Funktionen mit langem, kompliziertem Code.

Das Hauptprogramm ist so einfach wie die Klassen. Wir deklarieren ein Objekt jeder der vier Klassen in den Zeilen 41 bis 44 und rufen die Methode mit dem Namen *Nachricht()* für jedes Objekt einmal auf. Die Ausgabe auf dem Bildschirm zeigt uns, daß die Methode für jedes Objekt aufgerufen wird mit Ausnahme des *Sattelschlepper*s, dessen Klasse keine Methode diesen Namens hat. Wie wir im letzten Kapitel gesehen haben, wird die Methode mit dem Namen *Nachricht()* der Elternklasse aufgerufen. Das beweist auch die Ausgabe auf dem Bildschirm, wo für das Objekt mit dem Namen *Sattelschlepper* „Nachricht vom Vehikel“ zu lesen ist.

Die Daten der Objekte sind in diesem Kapitel von keinem Belang, also sind alle standardmäßig **private** und werden in die abgeleiteten Klassen nicht übernommen. Einiges haben wir dennoch übrig gelassen, damit die Klassen zumindest wie Klassen aussehen.

Wenn Du dieses Programm verstanden hast, kompiliere es und führe es aus.

11.2 Das Schlüsselwort virtual

Beispielprogramm: VIRTL2.CPP

Beim Durchsehen des nächsten Programmes, VIRTL2.CPP wird Dir eine kleine Änderung in Zeile 9 auffallen. Wir haben der Deklaration der Methode *Nachricht()* in der Elternklasse das Schlüsselwort **virtual** hinzugefügt.

Du bist wahrscheinlich ein wenig enttäuscht, wenn Du feststellst, daß sich dieses Programm nicht anders verhält als das letzte. Das ist deshalb der Fall, weil wir direkt mit Objekten arbeiten und virtuelle Methoden nichts mit Objekten zu tun haben, sondern nur mit Zeigern auf Objekt, wie wir gleich sehen werden. In Zeile 50 haben wir noch einen zusätzlichen Kommentar hinzugefügt, der illustriert, daß es nicht möglich ist, die Objekte einander zuzuweisen. Sie sind ja Objekte von verschiedenen Klassen. Wir werden gleich sehen, daß einige Zeigerzuweisungen zwischen Objekten verschiedener Klassen erlaubt sind.

Nachdem Du Dir sicher bist, daß virtuelle Funktionen oder Methoden nichts mit den Objekten direkt zu tun haben, kompiliere das Programm und führe es aus.

11.3 Wir verwenden Zeiger auf Objekte

Beispielprogramm: VIRTL3.CPP

Sieh Dir das Programm VIRTL3.CPP an, und Du wirst darin eine Wiederholung des vorigen Beispielprogrammes erkennen, allerdings mit einem anderen Hauptprogramm.

In diesem Programm haben wir das Schlüsselwort **virtual** wieder aus der Methodendeklaration der Elternklasse in Zeile 9 entfernt. Das Hauptprogramm definiert in den Zeilen 41 bis 44 Zeiger auf die Objekte statt der Objekte selbst. Da wir nur Zeiger auf Objekte definiert haben, müssen wir in den Zeilen 46 bis 53 zunächst die Objekte mit dem Operator **new** erzeugen, bevor wir sie verwenden können. Beim Ausführen des Programmes stellen wir fest, daß sich durch die Verwendung von Zeigern an dem, was das Programm tut, sich nichts geändert hat. Es operiert genauso wie das erste Programm in diesem Kapitel. Das sollte Dich aber nicht überraschen, da ein Zeiger auf eine Methode mit einem Objekt genauso arbeiten kann wie wir mit dem Objekt selbst arbeiten können.

Kompiliere dieses Beispielprogramm und führe es aus, bevor Du zum nächsten weitergehst. Dir wird wahrscheinlich wieder (unangenehm) aufgefallen sein, daß wir die Beschaffung der Objekte nicht kontrolliert haben, ja sie nicht einmal gelöscht haben. Wie immer spielt das aber in einem so kleinen Programm keine Rolle, da der Speicher mit der Rückkehr zum Betriebssystem automatisch freigegeben wird.

11.4 Ein Zeiger und eine virtuelle Funktion

Beispielprogramm: VIRTL4.CPP

Das Programm mit dem Namen VIRTL4.CPP ist mit dem letzten identisch, nur haben wir in Zeile 9 das Schlüsselwort **virtual** wieder hinzugefügt.

Ich hoffe, Du bist jetzt nicht ganz am Boden zerstört, wenn Du herausfindest, daß dieses Programm – nun mit dem Schlüsselwort **virtual** – immer noch dasselbe Ergebnis liefert wie das erste. Wir verwenden einfach Zeiger auf die Objekte, wobei in jedem Fall der Typ des Zeigers mit dem des Objektes, auf das er zeigt, übereinstimmt. Die ersten Veränderungen werden wir im nächsten Beispielprogramm bemerken, harre also aus, wir sind fast am Ziel.

Auch dieses Programm solltest Du kompilieren und ausführen.

Die letzten vier Programme haben gezeigt, was virtuell Funktionen nicht tun. Die nächsten beiden sollen zeigen, was sie tun.

11.5 Ein einfacher Zeiger auf die Elternklasse

Beispielprogramm: VIRTL5.CPP

Im Beispielprogramm VIRTL5.CPP verwenden wir schon fast eine virtuelle Methode. Habe noch ein ganz wenig Geduld, wir sind beinahe so weit.

Dieses Beispielprogramm ist eine weiter Variante unseres Programmes ohne das Schlüsselwort **virtual**, aber mit einem komplett veränderten Hauptprogramm. Hier definieren wir nur einen einzigen Zeiger auf eine Klasse, und diese Klasse ist die Basisklasse unserer Klassenhierarchie. Wir verwenden den Zeiger mit jeder der vier Klassen und achten auf die Ausgabe der Methode *Nachricht()*.

Ein kurzer Exkurs soll uns erklären, warum ein Zeiger, den wir als auf eine spezifische Klasse zeigend deklariert haben, plötzlich auf eine andere zeigen kann. Wenn wir von einem Vehikel sprechen (zugegeben: wir tun das nicht allzu oft), kann es sich dabei um

ein Auto, einen Lastwagen, ein Motorrad oder alles mögliche, das der Fortbewegung dient, handeln. Wir beziehen uns auf eine sehr umfassende Form eines Objektes. Wenn wir aber von einem Auto sprechen, meinen wir eben keinen Lastwagen, kein Motorrad oder irgendetwas anderes, diese Möglichkeiten sind ausgeschlossen. Der umfassendere Begriff des Vehikels kann sich also auf viele Arten von Fortbewegungsmitteln beziehen, der spezifischere Terminus des Autos bezieht sich nur auf eine Art von Vehikel, nämlich ein Auto.

Denselben Gedankengang können wir nun auf C++ übertragen. Wenn wir demnach eine Zeiger auf ein *Vehikel* haben, kann dieser Zeiger auf alle die spezielleren Objekte zeigen, und das ist in der Tat erlaubt in C++. Ebenso können wir aber einen Zeiger auf ein *Auto* nicht benutzen, um auf eine der anderen Klassen zu zeigen, auch nicht auf die Klasse *Vehikel*, da der Zeiger der Klasse *Auto* zu spezifisch und eingeschränkt ist.

11.6 Die C++ Zeiger Regel

Die Regel ist folgende: Ein Zeiger, der per Deklaration auf die Basisklasse zeigt, kann auf ein Objekt einer von der Basisklasse abgeleiteten Klasse zeigen, aber ein Zeiger auf eine abgeleitete Klasse kann nicht auf ein Objekt der Basisklasse oder einer anderen von der Basisklasse abgeleiteten Klasse zeigen. In unserem Programm dürfen wir also einen Zeiger auf *Vehikel*, die Basisklasse, deklarieren und diesen Zeiger auf Objekte der Basisklasse oder einer der abgeleiteten Klasse zeigen lassen.

Genau das tun wir im Hauptprogramm. Wir definieren einen einzelnen Zeiger, der auf die Klasse *Vehikel* zeigt und verwenden ihn, um auf Objekte aller Klassen zu zeigen, in derselben Reihenfolge wie in den letzten Beispielprogrammen. In jedem Fall erzeugen wir das Objekt, senden eine Nachricht an die Methode mit dem Namen *Nachricht()* und löschen das Objekt, bevor wir zur nächsten Klasse weitergehen. Wie Du sicher bemerkt hast, senden wir die Nachricht immer an dieselbe Methode, nämlich die Methode mit dem Namen *Nachricht()*, die Teil der Basisklasse *Vehikel* ist. Das ist der Fall, weil der Zeiger mit einer Klasse verbunden ist. Obwohl der Zeiger eigentlich auf Objekte von vier verschiedenen Klassen zeigt, verhält sich das Programm als würde er immer auf ein Objekt der Basisklasse zeigen, da der Zeiger von diesem Typ ist.

Das nächste Programm wird schließlich und endlich etwas tun, was Du in keinem C Programm oder C++ Programm in dieser Einführung gesehen hast. Wenn Du also dieses Programm kompiliert und ausgeführt hast, werden wir uns unserer ersten virtuellen Funktion widmen.

11.7 Eine tatsächlich virtuelle Funktion

Beispielprogramm: VIRTL6.CPP

Schließlich sind wir doch zu einem Programm mit einer virtuellen Funktion, die sich auch wie eine virtuelle Funktion benimmt und den Polymorphismus illustriert, vorgedrungen. Es ist das Programm mit dem Namen VIRTL6.CPP.

Das Programm ist mit dem letzten identisch, nur haben wir in Zeile 9 das Schlüsselwort **virtual** wieder eingefügt, um die Methode *Nachricht()* zu einer virtuellen Methode zu machen. Es genügt, das Schlüsselwort **virtual** in der Basisklasse zu verwenden, die entsprechende Methode in den abgeleiteten Klassen wird vom System automatisch als **virtual** deklariert. In diesem Programm verwenden wir wieder den einzelnen Zeiger auf die Basisklasse, um Objekte zu erzeugen, zu verwenden, und zu löschen. Dies tun wir mit demselben Code wie im letzten Programm. Allein durch das Schlüsselwort **virtual** in Zeile 9 ist dieses Programm aber vom vorigen komplett verschieden.

Da die Methode *Nachricht()* in der Basisklasse als virtuelle Methode deklariert worden ist, wird jedesmal, wenn wir diese Methode mit einem Zeiger auf die Basisklasse aufrufen, eigentlich die Methode einer der abgeleiteten Klassen ausgeführt. Das ist aber nur der Fall, wenn in der abgeleiteten Klasse eine Methode mit diesem Namen existiert und der Zeiger auf ein Objekt dieser Klasse zeigt. Beim Ausführen des Programmes erhalten wir das gleich Ergebnis wie in den Programmen, in denen wir die Methoden der abgeleiteten Klassen direkt aufgerufen haben. Jetzt aber verwenden wir eine Zeiger auf den Typ der Basisklasse für die Methodenaufrufe.

Obwohl also der Code in den Zeilen 44, 48, 52 und 56 derselbe ist, entscheidet das System anhand des Typs, auf den der Zeiger zeigt, welche Methode ausgeführt wird. Diese Entscheidung fällt nicht beim Kompilieren, sondern erst beim Ausführen des Programmes. Das kann in manchen Programmiersituationenen sehr nützlich sein. Eigentlich passieren hier nur drei verschiedene Aufrufe, da die Klasse *Laster* keine Methode mit dem Namen *Nachricht()* hat und das System einfach die Methode der Basisklasse verwendet. Aus diesem Grund muß eine virtuelle Funktion eine Implementation in der Basisklasse haben, die verwendet werden kann, wenn für eine oder mehrere abgeleitete Klassen keine verfügbar ist. Beachte, daß die Nachricht eigentlich an einen Zeiger auf das Objekt gesendet wird, aber das ist eine gewisse Haarspalterei, die uns jetzt nicht allzu wichtig sein sollte.

Es ist vielleicht nicht offensichtlich, Du hast aber wahrscheinlich bemerkt, daß die Struktur der virtuellen Funktion in der Basisklasse und in jeder der abgeleiteten Klassen identisch ist. Der Rückgabetyp und die Anzahl sowie die Typen der Parameter müssen für alle Funktionen identisch sein, da wir ja mit der gleichen Anweisung eine jede der Funktionen aufrufen können.

11.8 Ist das wirklich wichtig?

Auf den ersten Blick mag dieses Programm nichts Weltbewegendes tun, virtuelle Funktionen sind aber ein sehr nützliches Konstrukt, wie wir auch im nächsten Kapitel zeigen wollen, wenn wir ein Personalliste für eine kleine Firma entwickeln.

Bei der Verwendung des Schlüsselwortes **virtual** bindet das System die Methodenaufrufe erst zur Laufzeit, wird es nicht verwendet, geschieht dies beim Kompilieren. Im ersteren Fall weiß der Compiler nicht, welche Methode schlußendlich auf die Nachricht reagieren wird, da der Typ des Zeigers beim Kompilieren nicht bekannt ist. Im zweiten Fall entscheidet der Compiler bei der Kompilation welche Methode auf die Nachricht,

die er an den Zeiger schickt, antworten wird.

Kompiliere dieses Programm und führe es aus, bevor Du zum nächsten Kapitel weitergehst, wo wir ein praktisches Beispiel für diese Technik geben wollen.

11.9 Programmieraufgaben

1. Ändere VIRTL3.CPP so, daß die Objekte vor dem Programmende gelöscht werden.
2. Erweitere VIRTL6.CPP um eine Methode *Nachricht()* für die Klasse *Laster* und beachte, dass nun diese Methode anstatt der der Elternklasse verwendet wird.

12 Mehr virtuelle Funktionen

Dieses Kapitel ist eine Fortsetzung des im letzten Kapitel behandelten Themas, soll aber eine komplettere Erklärung, was virtuelle Funktionen sind und wie sie in einem Programm verwendet werden können, liefern. Wir werden eine einfache Datenbank mit einer virtuellen Funktion vorstellen, um die Verwendung zu illustrieren. Dann werden wir eine etwas komplexere Verwendung von virtuellen Funktionen zeigen, die deren Leistungsnachweis und Daseinsberechtigung erbringen soll.

12.1 Wie beginne ich ein OOP-Projekt?

Unser objektorientiertes Abenteuer beginnen wir damit, daß wir ein Objekt finden, oder, wie in diesem Fall, eine Klasse von Objekten und sogar einige untergeordnete Objekte, und diese vollständig definieren. Beim Hauptprogramm angelangt, haben wir dann einfaches Spiel mit dem restlichen notwendigen Code, den wir mit den bekannten prozeduralen Techniken schreiben. So fängt also alles an: Wir suchen uns Objekte, die sich vom übrigen Code sinnvoll trennen lassen, programmieren diese und schreiben dann das Hauptprogramm. Wir sollten noch erwähnen, daß Du gerade am Anfang nicht versuchen solltest, aus allem (und jedem???) mit aller Gewalt ein Objekt zu machen. Wähle Dir ein paar Objekte aus; wenn Du dann Erfahrung mit objektorientierten Programmiertechniken gesammelt hast, wirst Du in Deinen späteren Projekten mehr Objekte verwenden. Die meisten Programmiererinnen verwenden in ihrem ersten Projekt zu viele Objekte und schreiben eigenartigen, unleserlichen Code.

12.2 Die Person Header-Datei

Beispielprogramm: PERSON.H

Die Datei PERSON.H definiert die Klasse *Person*. Es findet sich hier nichts Neues, Du solltest also mit dem Verständnis keinerlei Problem haben. Das einzig Erwähnenswerte an dieser Klasse ist, daß wir die Variablen als **protected** deklarieren, womit sie in den von dieser abgeleiteten Klassen verfügbar sind. Beachte auch, daß die einzige Methode dieser Klasse in Zeile 11 virtuell ist.

12.3 Die Implementierung der Klasse „Person"

Beispielprogramm: PERSON.CPP

In der Datei PERSON.CPP findet sich die Implementation der Klasse *Person* und die ist ein wenig eigenartig. Es ist vorgesehen, daß die virtuelle Methode mit dem Namen

Zeige() in dieser Datei nie verwendet wird, der C++ Compiler verlangt sie aber, damit er auf sie zurückgreifen kann, wenn eine Subklasse keine Methode mit diesem Namen bereitstellt. Wir werden uns hüten, diese Funktion im Hauptprogramm aufzurufen. Merke Dir aber, daß C++ eine Implementation für alle virtuellen Funktionen verlangt, auch wenn diese auch nie verwendet werden. In unserem Fall geben wir offensichtlich eine Fehlermeldung aus.

Kompiliere dieses Programm, bevor Du zur nächsten Klassendefinition weitergehst.

12.4 Die Aufseherin Header-Datei

Beispielprogramm: AUFSHR.H

In der Datei AUFSHR.H findest Du die Definition der drei abgeleiteten Klassen: *Aufseherin*, *Programmiererin* und *Sekretaer*. Aus zwei Gründen sind sie alle in einer Datei. Erstens wollen wir bewiesen haben, daß das funktioniert und zweitens können wir so einige Klassen kombinieren und Du mußt nicht so viel kompilieren. Es macht auch Sinn, diese Klassen zusammenzufassen, da sie alle von einer gemeinsamen Elternklasse abgeleitet sind.

Alle drei Klassen haben eine Methode mit dem Namen *Zeige()* und alle diese Methoden haben wiederum den Rückgabetyp **void** und dieselbe Anzahl an Parametern wie die gleichnamige Methode der Elternklasse. Diese Ähnlichkeiten sind notwendig, da alle diese Methoden mit demselben Aufruf aufgerufen werden können. Auch die andere Methode der drei abgeleiteten Klassen trägt überall denselben Namen, die Parameter sind aber in Anzahl und Typen verschieden. Deshalb können wir diese Methode nicht als virtuelle Methode verwenden.

Der Rest dieser Datei ist einfach.

12.5 Die Implementierung der drei abgeleiteten Klassen

Beispielprogramm: AUFSHR.CPP

In der Datei AUFSHR.CPP implementieren wir die drei Klassen. Wenn Du Dir den Code kurz ansiehst, wirst Du erkennen, daß die Methoden mit dem Namen *InitDaten()* einfach alle Elemente mit den als Parametern gegebenen Werten initialisieren.

Die Methode mit dem Namen *Zeige()* gibt die Daten für jede Klasse auf eine andere Weise aus, da die Daten so verschieden sind. Obwohl also die Schnittstelle zu allen diesen Funktionen dieselbe ist, ist der Code der Implementation recht verschieden. Natürlich hätten wir auch allen möglichen anderen Code schreiben können, die Ausgabe ist nun aber einmal so schön sichtbar und deshalb für Illustrationszwecke am besten geeignet.

Du solltest diese Datei jetzt kompilieren als Vorbereitung auf das nächste Beispielprogramm, das alle vier Klassen, die wir in den letzten vier Dateien definiert haben, verwenden wird.

12.6 Das erste Programm

Beispielprogramm: ANGEST.CPP

In der Datei ANGEST.CPP verwenden wir zum ersten Mal die Klassen, die wir in diesem Kapitel entwickelt haben. Wie Du leicht erkennen kannst, handelt es sich um ein einfaches Programm.

Wir beginnen mit einem Array von zehn Zeigern, die alle auf die Basisklasse zeigen. Du erinnerst Dich sicher, daß es für virtuelle Funktionen sehr wichtig ist, daß ein Zeiger auf die Basisklasse zeigt. Die Zeiger, die wir in diesem Array dann speichern, werden allerdings alle auf Objekte der abgeleiteten Klassen zeigen. Wenn wir mit den Zeigern Methoden aufrufen, wird das System die richtige beim Ausführen auswählen und nicht schon beim Kompilieren, wie dies fast alle unsere bisherigen Beispielprogramme getan haben.

In den Zeilen 16 bis 39 erzeugen wir sechs Objekte [tsts] und initialisieren sie mit den Methoden *InitData()*. Dann weisen wir den Elementen des Arrays von Zeigern auf *Person* die Zeiger zu. In den Zeilen 41 bis 44 rufen wir schließlich die Methoden mit dem Namen *Zeige()* auf, um die gespeicherten Daten auf dem Bildschirm auszugeben. Obwohl wir also in Zeile 43 nur einen Funktionsaufruf verwenden, senden wir doch an alle drei Methoden *Zeige()* in den abgeleiteten Klassen Nachrichten.

Kompiliere dieses Programm und führe es aus, bevor Du in diesem Kapitel weitergehst. Das Linken erfordert auch bei diesem Beispiel, daß Du die die Teile zuvor einzeln kompiliert hast.

12.7 Die Klasse der verbundenen Liste

Beispielprogramm: ELEMLIST.H

In der Datei ELEMLIST.H findest Du die Definitionen von zwei weiteren Klassen, die wir zum Erzeugen einer verbundenen List von Angestellten verwenden werden.

Die zwei Klassen sind in einer Datei zusammengefasst, weil sie sehr eng zusammenarbeiten und die eine ohne die andere so gut wie nutzlos ist. Die Elemente der verbundenen Liste enthalten keine Daten, sondern lediglich einen Zeiger auf die Klasse *Person*, die wir für das letzte Programm entwickelt haben. Die Liste besteht also aus Elementen der Klasse *Person*, ohne diese Klasse zu modifizieren.

Zwei interessante Aspekte dieser Datei müssen wir noch erwähnen. Der erste ist die partielle Deklaration in Zeile 8, die es uns erlaubt, die Klasse mit dem Namen *AngestelltenListe* zu verwenden, bevor wir sie überhaupt deklarieren. Die komplette Deklaration steht in den Zeilen 23 bis 31. Das zweite interessante Konstrukt ist die Freundklasse in Zeile 18, wo wir der gesamten Klasse mit dem Namen *AngestelltenListe* freien Zugriff auf die Variablen der Klasse *AngestelltenElement* gestatten. Das ist notwendig, weil die Methode mit dem Namen *AngestellteHinzu()* auf die Zeiger in *AngestelltenElement* zugreifen können muß. Wir hätten eine zusätzliche Methode der Klasse *AngestelltenElement* definieren können und mit dieser auf die Zeiger zugreifen können, aber diese beiden Klassen arbeiten so gut und eng zusammen, daß es kein Problem darstellt, wenn wir in

unserer Mauer ein Loch lassen. Die Privatsphäre wird ja vor allen anderen Funktionen und Klassen des Programmes gewahrt.

Die einzige Methode der Klasse *AngestelltenElement* haben wir **inline** implementiert. Zwei Methoden der Klasse *AngestelltenListe* sind noch undefiniert, wir brauchen also eine Implementation für diese Datei.

12.8 Die Implementation der verbundenen Liste

Beispielprogramm: ELEMLIST.CPP

Die Datei mit dem Namen ELEMLIST.CPP ist die Implementation der verbundenen Liste und sollte kein Problem sein, wenn Dir klar ist, wie eine einfach verbundene Liste funktioniert. Alle neuen Elemente werden am Ende der aktuellen Liste angefügt, um die Liste einfach zu gestalten. Ein alphabetischer Sortiermechanismus, um die Angestellten nach dem Namen zu sortieren, könnte natürlich hinzugefügt werden. Wenn der benötigte Speicher nicht beschafft werden kann, stoppt das Programm einfach. Das ist für ein „ordentliches“ Programm natürlich nicht akzeptabel. Die Fehlerbehandlung ist ein wichtiges Thema, mit dem Du Dich früher oder später auseinandersetzen müssen wirst.

Die Methode zum Anzeigen der Liste durchwandert diese einfach und ruft in Zeile 30 für jedes Element einmal die Methode mit dem Namen *Zeige()* auf.

Ist Dir aufgefallen, daß nirgendwo in dieser Klasse auch nur die Existenz der drei abgeleiteten Klassen erwähnt ist? Nur die Basisklasse wird genannt. In der Link-Liste existieren die drei Subklassen also nicht. Trotzdem sendet diese Klasse - wie wir sehen werden - Nachrichten an die drei Subklassen. Genau so funktioniert der dynamische Aufruf von Methoden. Nachdem wir uns ein Programm angesehen haben, das die verbundene Liste verwendet, werden wir noch mehr darüber zu sagen haben.

12.9 Wir verwenden die verbundene Liste

Beispielprogramm: ANGEST2.CPP

Das Programm ANGEST2.CPP ist unser bestes Beispiel für dynamischen Methodenaufruf in dieser Einführung, aber doch ein sehr einfaches Programm.

Dieses Programm ist dem Programm ANGEST.CPP sehr ähnlich, die wenigen Änderungen machen aber eine insgesamt bessere Illustration aus. In Zeile 7 definieren wir ein Objekt der Klasse *AngestelltenListe* und beginnen unsere verbunden Liste. In diesem Programm benötigen wir nur dieses eine Objekt. Für alle Elemente der Liste beschaffen wir den Speicherbereich, füllen ihn an und senden das Element an die Liste, um es ihr anzufügen. Der Code ist dem des vorigen Programmes bis Zeile 40 in der Tat sehr ähnlich.

In Zeile 43 senden wir eine Nachricht an die Methode *ZeigeListe()*, die die gesamte Personalliste ausgibt. Die Klasse für die verbundene List, wie wir sie in den Dateien ELEMLIST.H und ELEMLIST.CPP definiert haben, hat keinerlei Kenntnis von den

Subklassen, übergibt aber die Zeiger auf diese Klassen an die richtigen Methoden und das Programm verhält sich so, wie wir es erwarten.

12.10 Wozu ist das alles gut [Zawosbrauchides]?

Stell Dir vor, wir kommen auf die Idee, unser Programm – jetzt fix und fertig und fehlerbereinigt und funktionstüchtig und überhaupt – um eine weitere Klasse erweitern. Wir könnten zum Beispiel eine Klasse *Beraterin* hinzufügen, weil wir eine Beraterin in unserer Firma brauchen.

Wir müßten natürlich zuerst die Klasse mit ihren Methoden schreiben. Die verbundene Liste braucht aber nichts davon zu erfahren, daß wir ihr eine weitere Klasse untergejubelt haben und wir müssen das Programm also überhaupt nicht verändern, damit es auch die Klasse *Beraterin* miteinbezieht. In diesem spezifischen Fall ist die verbundene Liste sehr klein und einfach zu verstehen, stell Dir aber vor, der Code wäre umfangreich und komplex wie bei einer großen Datenbank. Es wäre sehr schwierig, jede Referenz auf die Subklassen zu aktualisieren und die neue Subklasse überall hinzuzufügen. Dieses umständliche Verfahren stellte natürlich auch ein Paradies für den *Fehlerteufel* dar. Wir müssen unser Beispielprogramm nicht einmal neu kompilieren, um seinen Einsatzbereich zu erweitern.

Es sollte Dir klar sein, daß es möglich wäre, während der Abarbeitung des Programmes neue Typen zu definieren, sie dynamisch zu erzeugen und auch gleich zu verwenden. Wir müßten dazu den Code auf verschiedene Module aufteilen, die dann parallel ablaufen. Das wäre nicht einfach, aber durchaus möglich.

Wenn Du aufgepaßt hast, ist Dir wahrscheinlich aufgefallen, daß wir weder die Elemente der Liste noch die Liste selbst „zerstören“. Wir müßten also die Klasse *AngestelltenListe* um eine Methode *LoeschePerson()* und die Klasse *Person* um einen Destruktor erweitern.

12.11 Ein Anwendungsgerüst

Beispielprogramm: ANWGER1.CPP

Das Beispielprogramm ANWGER1.CPP illustriert die Methode, die beim Erstellen eines Gerüstes für eine Anwendung zur Anwendung kommt. Wenn Du viel programmierst, wird Dir so etwas sicherlich bei Programmen für ein Betriebssystem mit GUI (Graphical User Interface - Graphische Benutzeroberfläche; etwa: Mac OS X, X Windows, MS Windows,...) nützlich sein. Es kann also nicht schaden, damit vertraut zu sein.

Die Klasse *CForm* ist die Basisklasse für unser triviales, aber wichtiges Beispiel und besteht aus vier Methoden, aber keinen Datenelementen. Die Methode mit dem Namen *ZeigeForm()* ruft die anderen drei auf, um unsere kleine Form (Platon: Idee) am Bildschirm auszugeben. Es ist also an diesem Programm nichts Besonderes, außer, daß es Das Gerüst für alle momentan verfügbaren Anwendungsgerüste ist (ein Meta-Gerüst also?). Beachte, daß drei der Methoden in den Zeilen 9 bis 11 als **virtual** deklariert werden.

Das Interessante passiert, wenn wir in Zeile 27 die Klasse in unsere neue Klasse mit dem Namen *CMeineForm* importieren und neue Methoden für zwei der Basisklassenmethoden schreiben. Wir haben so viel Funktionalität aus der Basisklasse übernommen, wie uns angenehm war und neue Methoden geschrieben, wo das in der Basisklasse vorhandene nicht den gewünschten Zweck erfüllt. Wenn wir in Zeile 42 schließlich ein Objekt der neuen Klasse verwenden, mischen sich also Teile der Basisklasse mit neu geschriebenen Methoden.

In einem so einfachen Beispiel ist das nicht sonderlich attraktiv, wenn wir aber bedenken, wie diese Technik in einem wirklichen Anwendungsgerüst verwendet wird, erscheint es sehr nützlich. Die Autorin des Anwendungsgerüstes schreibt ein komplettes Programm, das alle Notwendigkeiten und das Management der Fenster übernimmt und teilt dieses Programm auf virtuelle Funktionen auf, so wie wir es hier getan haben. Wir suchen uns dann wieder die Teile aus diesem Kuchen, die wir brauchen können und schreiben die Teile neu, die wir ändern wollen. Ein großes Stück Programmierarbeit ist schon für uns erledigt worden.

Das Anwendungsgerüst kann noch viele weitere schon programmierte Funktionen enthalten, wie zum Beispiel solche zur Textverarbeitung oder zum Darstellen von Dialogen. Du wirst es sehr interessant und nützlich finden, Anwendungsgerüste nach allen ihren Funktionen zu durchforsten.

12.12 Eine rein virtuelle Funktion

Beispielprogramm: ANWGER2.CPP

Das Beispielprogramm ANWGER2.CPP zeigt eine rein virtuelle Funktion. Eine reine virtuelle Funktion wird deklariert, indem wir ihr den Wert 0 zuweisen, wie in Zeile 10. Eine Klasse, die eine oder mehrere rein virtuelle Funktionen enthält kann nicht zum Erzeugen von Objekten verwendet werden. Das stellt sicher, daß für jeden Aufruf eine Funktion verfügbar ist und keiner von der Basisklasse beantwortet werden muß, wozu sie in Ermangelung einer Funktionsimplementation auch gar nicht in der Lage ist.

Du kannst kein Objekt einer Klasse erzeugen, die eine oder mehrere rein virtuelle Funktionen beinhaltet, da eine Nachricht an eine rein virtuelle Funktion nicht behandelt werden könnte. Der Compiler stellt sicher, daß die beiden Regeln eingehalten werden. Wenn eine Klasse eine abstrakte Klasse ererbt, ohne die rein virtuelle Methode/n zu überschreiben, wird sie selbst zur abstrakten Klasse, mit der keine Objekte erzeugt werden können.

Da unser Beispiel eine abstrakte Basisklasse verwendet, ist es nicht mehr möglich, ein Objekt der Basisklasse zu verwenden, wie wir dies im vorigen Programm getan haben. Aus diesem Grund ist ein Teil des Programmes auskommentiert.

Abstrakte Klassen finden in vielen Bibliotheken und Anwendungsgerüsten Anwendung. Kompiliere das Programm und führe es aus. Verändere dann den Code ein wenig, um zu sehen, welche Fehler der Compiler ausgibt, wenn Du die Regeln, die wir für abstrakte Klassen aufgestellt haben, verletzt.

12.13 Programmieraufgaben

1. Erweitere die Dateien AUFSHR.H und AUFSHR.CPP um eine Klasse mit dem Namen *Beraterin*, dann die Datei ANGEST2.CPP um Code, der die neue Klasse verwendet. Du mußt die verbundene Liste nicht neu kompilieren, um die neue Klasse zu verwenden.

13 Abflug – Das Spiel

So, jetzt haben wir eine Menge über C++ gelernt und wissen ziemlich genau, wie wir eine einzelne Klasse zu schreiben und zu verwenden haben. Wie aber erstellen wir ein Programm, in dem verschiedene Klassen nebeneinander und zusammen arbeiten, um eine Aufgabe zu erfüllen? Ein Abenteuerspiel scheint sich für ein größeres Beispielprogramm recht gut zu eignen. Wir haben es mit viel Ein- und Ausgabe zu tun und das Programm muß ziemlich flexibel sein, da so viele Dinge das Spiel als Hindernisse, Rätsel, Überraschungen beeinflussen können.

Ort dieses einzigartigen Spieles ist ein Flughafen. Wichtiger als der Flughafen ist der Code, den wir brauchen, um heil durch den Flughafen zu kommen. Du solltest das Spiel zunächst einmal spielen, bis Du weißt, was der Code tut, um dann im Code selbst nachzuschauen, wie er das tut. Schließlich bekommst Du noch eine Aufgabe, den Code zu erweitern. Dann siehst Du, ob Du wirklich alles verstanden hast.

13.1 Spielispieli

Erst das Spiel, dann das Vergnügen. Zuerst also solltest Du ein bißchen Abflug spielen, bevor Du Dich auf den Quellcode stürzt.

Wenn Du schon einmal ein „Text-Adventure“ gespielt hast (wirst du einigermaßen enttäuscht sein), solltest Du einfach ein paar Kommandos ausprobieren, um Deinen Weg durch den Flughafen zum richtigen Flieger zu finden. Ist Dir diese Art von Spiel völlig fremd, sollen einige Hinweise weiterhelfen.

Ziel des Spieles ist es, rechtzeitig das richtige Flugzeug zu besteigen und in den wohlverdienten Urlaub zu fliegen. Selbstredend stellen sich Dir einige Hindernisse und Probleme in den Weg. Die Lösung mußt Du selbst finden. Um das ganze ein bißchen spannender zu machen, hast Du nur ungefähr 25 Minuten Zeit. Jedes Kommando dauert eine Minute, Du mußt Dich also sputen! Nur das Flugzeug rechtzeitig zu erreichen ist aber auch noch nicht genug, es müssen einige zusätzliche Bedingungen erfüllt sein. Diese werden im Laufe des Spieles klar werden (spätestens am Schluß).

Du hast weniger als fünf Versuche gebraucht, um das Spiel zu schaffen (ohne den Code zu kennen)? Respekt.

13.2 Der Spielablauf

Die Spielweise ist extrem einfach. Du rennst einfach am Flughafen herum, immer auf der Suche nach Dingen, die Du tun könntest. Du bewegst dich, indem Du dem Programm mitteilst, in welche der vier Himmelsrichtungen (Norden, Süden, Osten oder Westen)

Du gehen willst. Alle diese Richtungen können durch den ersten Buchstaben abgekürzt werden und entweder in Groß- oder in Kleinbuchstaben eingegeben werden. Es kann auch vorkommen, daß Du in eine bestimmte Richtung nicht gehen kannst. Starte das Programm jetzt, versuche es einmal mit den vier Richtungen und schau, was passiert. Wenn das noch unklar ist, dann gib das Wort „Hilfe“ ein.

Du kannst auch Dinge nehmen oder Dich in den einzelnen Räumen umsehen. Sag dem System jetzt, daß Du Dich in diesem Raum umsehen willst, indem Du Schau eingibst. Dahinter verbirgt sich die Lösung zu so manchem Rätsel. Ein weiteres wichtiges Kommando ist Rucksack, das eine Liste der Dinge ausgibt, die Du bei Dir trägst. Was haben wir im Rucksack?

Alle anderen Kommandos bestehen aus zwei Wörtern, einem Verb und einem Hauptwort. Die Reihenfolge ist dabei unerheblich, da das Programm den Unterschied erkennt. Wenn Du ein Kommando eingibst, das das Programm nicht kennt, wird es Dir sagen, daß es Dich nicht versteht. Lege jetzt etwas hin, das Du besitzt, oder Nimm etwas, das sich im selben Raum befindet wie Du selbst.

Ein paar Freunde haben Abflug mit genau den Angaben gespielt, die ich auch Dir jetzt gegeben habe. Einer hat nur 40 Minuten gebraucht, die meisten aber eine Stunde, bis sie es geschafft haben, in Urlaub zu fliegen. Der gesamte Code für das Spiel ist in CPPSRC.ZIP enthalten, das Du für die übrigen Kapitel dieser Einführung schon heruntergeladen haben solltest. Das Spiel ist absichtlich relativ einfach und kurz, damit es auch einfach und schnell verstanden werden kann. Es gibt keinen Grund, warum das Programm nicht durch zusätzliche Räume, Gegenstände und Fallen viel größer gemacht werden könnte. Vielleicht machst Du ja gerade das, um weitere Erfahrungen im Programmieren in C++ zu sammeln.

13.3 Einige spezielle Konstanten

Beispielprogramm: ABFLUG.H

Die Datei ABFLUG.H beinhaltet die Definitionen von TRUE und FALSE sowie den Aufzählungstypen, die unseren Wortschatz für das Spiel definieren. Wir beginnen die List mit 1, damit wir den Wert 0 später verwenden können, um anzuzeigen, daß ein Wort nicht im Wörterbuch aufscheint.

Das **#ifndef** in Zeile 4 ist notwendig, da diese header-Datei in vielen anderen Dateien importiert wird. Wir müssen also eine multiple Definition vermeiden. Eine Klasse muß nur einmal definiert werden und wenn dies durch das Importieren dieser Datei geschehen ist, wird *ABFLUG_H* definiert und damit eine nochmalige Definition verhindert. Genauer haben wir das am Ende des Kapitel 8 beschrieben.

13.4 Die erste Klasse – Uhr

Beispielprogramm: UHR.H

In der Datei UHR.H definieren wir die gleichnamige Klasse, *Uhr*. Dies ist die Klasse für die Spieluhr und wir werden nur eine Instanz dieser Klasse verwenden, und zwar

Tageszeit, definiert in Zeile 23 von ABFLUG.CPP.

Die Klasse ist sehr einfach und besteht lediglich aus zwei Variablen, *Stunde* und *Minute*, und vier Methoden. Die erste Methode ist der Konstruktor, mit dem wir die Uhr mit 8:51 initialisieren, wie aus der Implementation der Klasse in UHR.CPP ersichtlich ist. Die nächsten beiden Methoden dienen dazu, die Werte der Variablen zu bekommen. Die letzte Methode ist da schon interessanter. Sie aktualisiert die Tageszeit und gibt die Eingabeaufforderung aus. Dies ist wahrscheinlich nicht der beste Platz, um die Eingabeaufforderung auszugeben, da es sich in dieser Klasse nur um die Tageszeit dreht. Wir haben sie aber dafür gewählt, weil die Zeitangabe Teil der Eingabeaufforderung ist. Es ist Dir sicher aufgefallen, daß wir zwar die Uhr mit 8:51 initialisieren, die erste Ausgabe aber auf 8:52 lautet. Um es uns später leichter zu machen, wenn wir bei jedem Spielzug überprüfen müssen, ob das Flugzeug rechtzeitig erreicht wurde, zählen wir die Zeit schon am Beginn eines jeden Spielzuges hinauf. Deshalb ist die Zeit bei der Eingabe des Kommandos und dessen Abwicklung dieselbe und deshalb wird auch schon vor der ersten Ausgabe hinaufgezählt. Die Klasse *Uhr* ist die einfachste unseres Spieles und Du solltest mit dem Verständnis keine Probleme haben.

13.5 Kommandos

Beispielprogramm: WOERTER.CPP

Die Routinen, mit denen wir die Eingabe bearbeiten, werden in der Klasse *Woerter* definiert. Der Code für diese Klasse findet sich in WOERTER.CPP. Der Code ist relativ einfach zu verstehen, wir beschränken uns also auf einige Kommentare.

Die Methode *HoleAnweisung()* liest mit der Funktion *LiesEineZeile()* zwei Wörter und speichert sie in den Klassenelementen *Verb* und *Substantiv*. Die Methode speichert null für eines oder beide Wörter, wenn sie kein gültiges Verb oder Substantiv findet.

Zwei Methoden machen das Verb oder das Substantiv der letzten Eingabe verfügbar. Damit können wir in jedem Code, in dem das Objekt, das wir mit dieser Klasse erstellen, sichtbar ist, herausfinden, was die Spielerin will.

Die vier Methoden, die mit *Ist* beginnen, stellen fest, ob sich um ein Verb, ein Substantiv, eine Richtung oder eine Handlung handelt. Diese Methoden werden wir des öfteren aufrufen.

Schließlich und endlich setzt die einfachste der Methoden, *StoppeSpiel()* das *Verb* auf Aus, damit das Gespiel ein Ende hat. Dies bewerkstelligt der Code im Hauptprogramm ABFLUG.CPP.

Die Implementation dieser Klasse findest Du in der Datei WOERTER.CPP. Der Code ist sehr einfach und reichlich kommentiert, Du darfst ihn Dir also alleine ansehen.

13.6 Die zweite Klasse - Gegenstaende

Beispielprogramm: GGSTDE.CPP

In den Dateien GGSTDE.H und GGSTDE.CPP findest Du die komplette Definition und das Handling aller Gegenstände, die Du beim Spielen herumgeschleppt hast. Es gibt

genau vier mobile Gegenstände, die entweder in einem Raum oder im Rucksack sind: die *Schluessel*, das *Konfekt* (perfekt! und ich frage mich, warum ich nicht mindestens einen Müesliriegel daraus gemacht habe... nur des Namenswegen.), das *Ticket* und das *Geld*. Mit Geld oder Schlüssel kommt man nicht durch die Sicherheitskontrolle und Ticket und Konfekt benötigen wir, damit wir ins richtige Flugzeug und auch bis ans Ziel kommen.

Die vier Gegenstände werden in der Klasse mit dem Namen *Gegenstaende* in der Form TRUE oder FALSE gespeichert. Dies reicht völlig aus; ein TRUE bedeutet, daß der Gegenstand hier ist, ein FALSE das Gegenteil. Die Werte TRUE und FALSE definieren wir in ABFLUG.H. Schließlich haben wir noch sechs Methoden, um mit den Gegenständen zu arbeiten.

Die erste Methode setzt alle Gegenstände auf FALSE. Die nächsten beiden werden dazu verwendet, den angegeben Gegenstand hinzulegen oder zu nehmen. Die vierte Methode sagt uns, ob sich ein Gegenstand hier befindet und die letzten beiden sagen uns, welche Gegenstände bei der Hand sind.

Auch diese header-Datei schützen wir vor mehrfachem Importieren durch das **#ifndef** Konstrukt.

In Zeile 24 von ABFLUG.CPP verwenden wir diese Klasse, um ein Objekt für die Spielerin mit dem Namen *PersoenlicheGegenstaende* zu definieren, das die List der Gegenstände speichert, die die Spielerin spazierenträgt. Auch in der Klasse *Ort* verwenden wir diese Klasse als eingebettetes Objekt, um die Gegenstände zu speichern, die an jedem der 19 Orte liegen.

Die Implementation dieser Klasse ist wiederum so einfach, daß Du keine Schwierigkeiten beim Verständnis haben solltest.

13.7 Die Klasse der Flüge und Gates – Plan

Beispielprogramm: PLAN.H

Die Dateien PLAN.H und PLAN.CPP sind unser erstes Beispiel einer etwas größeren Klasse. Sie kümmert sich um die Flüge und die Gates. In dieser Klasse findest Du eine Vielzahl von Variablen und Anzahl von 8 Methoden, die mit der Vielzahl arbeiten. Anstelle einer detaillierten Beschreibung jeder einzelnen Variable und Methode wollen wir uns auf einen Überblick über die Klasse beschränken.

Nur ein Objekt dieser Klasse, *FlugInfo*, wird in Zeile 22 des Hauptprogrammes ABFLUG.CPP deklariert. Der Konstruktor initialisiert die möglichen Flüge. Die Methode mit dem Namen *AendereGates()* (ein Schelm, wer da and'res denkt!) ändert alle Gates, wenn die Spielerin an ihrem richtigen Gate ankommt, ohne den Monitor in der Wartezone gelesen zu haben. Wenn letzteres aber passiert ist, wird die Variable *FluegeStehen* auf TRUE gesetzt. Die Destination wird von der Methode *AendereFluege()* bei jedem Spielzug geändert bis die Spielerin ihr Ticket liest und damit die Methode *ZeigeDestination()* aufruft.

Diese Klasse enthält die Methoden für die Daten der Monitoranzeige und die Daten, die angezeigt werden, wenn die Spielerin an einem der Gates schaut. Schließlich enthält diese Klasse noch die Methode mit dem Namen *UeberpruefeFlug()*, die die List der

Voraussetzungen durchsucht, um festzustellen, ob die Spielerin alle erfüllt hat.

Einige der Ortsobjekte mußten in dieser Klasse verfügbar sein, deshalb scheinen sie in den Zeilen 12 bis 21 der Implementation der Klasse als **extern** auf. Erwähnenswert wäre dann noch das Problem, dessen Du dich erleichtern mußt, bevor Du abfliegen kannst. In Zeile 28 definieren und initialisieren wir die globale Variable. Auf TRUE setzen wir sie dann in Zeile 77, wenn der momentane Aufenthaltsort die Toilette ist. Schließlich überprüfen wir den Wert der Variable in Zeile 230 dieser Datei und machen davon den weiteren Spielverlauf (respektive Nicht-Verlauf) abhängig. Das Hauptprogramm weiß von der Existenz dieser Variable nichts und auch nicht davon, daß sie das Spielgeschehen beeinflußt.

Du hast zwar eine relativ große und komplexe Klasse vor Dir, sie ist aber durchgehend kommentiert und wir werden uns deshalb nicht länger mit ihr aufhalten.

13.8 Die meistbenutzte Klasse – Ort

Beispielprogramm: ORT.H

Die Datei mit dem Namen ORT.H ist die header-Datei der Klasse mit dem Namen *Ort*. Diese Klasse kontrolliert alle Bewegungen von einem Ort zu einem anderen.

Diese Klasse ist insofern ein wenig unüblich, als die meisten Daten als Zeiger gespeichert sind. Die ersten vier sind Orte, zu denen wir kommen, wenn wir uns von unserem momentanen Aufenthaltsort in eine der vier Richtungen bewegen. Es handelt sich dabei um Zeiger auf diese vier Orte. Dann kommen Zeiger auf zwei verschiedene Zeichenketten, die zu diesem Raum gehören. Schließlich, in Zeile 22, definieren wir das Objekt *GegenstaendeListe*, ein Objekt der Klasse *Gegenstaende*, die wir zuvor definiert haben. Es handelt sich also um eine eingebettete Klasse. Das ist keine Elternklasse, von der wir etwas erben können. Wir erzeugen ein Objekt der Klasse *Gegenstaende*, das wir im Raum verwenden.

In dieser Klasse verwenden wir keinen Konstruktor, da wir die Orte einzeln initialisieren. Die Methode mit dem Namen *Init()* hat 6 Parameter, allesamt Zeiger, mit denen sie die ersten sechs Variablen des Objektes initialisiert. Die letzte Variable, ein Objekt der Klasse *Gegenstaende*, wird mit dem Konstruktor dieser Klasse initialisiert. In den Zeilen 40 bis 171 der Implementation der Klasse *Karte* findest Du den Code für die Initialisierung aller 19 Objekte der Klasse *Ort*. Da das Auto wegfährt, sobald Du es verläßt, kannst Du auch nicht dorthin zurückgehen.

Die nächste Methode mit Namen *Geh()* gibt einen Zeiger auf den neuen Ort zurück, wenn die Bewegung erfolgreich war [??], sonst NULL. Du hast sicher bemerkt, daß das Verlassen der Snackbar und das Passieren der Sicherheitskontrolle etwas Besonderes darstellen. Der Code dafür findet sich hier, weil es sich um Teile des Geh!-Kommandos handelt.

Die restlichen Methoden sind einfach und wir gehen nicht weiter auf sie ein.

13.9 Die Nachrichten

Beispielprogramm: NACHR.TXT

In der Datei NACHR.TXT findet sich eine komplette List der Nachrichten, die am Bildschirm erscheinen, wenn einer der Orte betreten wird. Auch die Nachrichten für das Kommando Schau findest Du hier. Die Nachrichten haben wir in einer eigenen Datei gesammelt, um den Umfang der Implementation der Klasse *Plan* zu reduzieren. Die Kompilationszeit können wir so nicht reduzieren, da sich NACHR.TXT ja nicht separat kompilieren läßt, sondern einfach in die Datei PLAN.CPP inkludiert wird. Einige der Nachrichten bestehen nur aus (leeren) Anführungszeichen, wir führen sie aber trotzdem an, um Argumente für die Initialisierung zu haben.

In den Zeilen 5 bis speichern wir noch weitere 3 Nachrichten in dieser Datei. Was sie bedeuten und wozu sie gut sind, sollte klar sein.

13.10 Das Hauptprogramm

Beispielprogramm: ABFLUG.CPP

Schlussendlich kommen wir also zum Hauptprogramm. Schau Dir die Datei ABFLUG.CPP an, wir werden einige interessante Dinge entdecken.

Wir beginnen mit *main()*, und nach einem Aufruf der Methode *Flughafen.Initialisiere()* beginnen wir eine **do. . . while** Schleife, die dann endet, wenn die Spielerin „Aus“ eingibt oder das Programm selbst dieses Wort generiert.

Die Schleife besteht aus 5 Methodenaufrufen. Zuerst rufen wir in Zeile 30 die Funktion *EingabeWoerter.HoleAnweisung()* auf, um die Eingabe zu holen. Dann senden wir zwei Nachrichten an das Objekt *FlugInfo*, um die Flüge und Gates zu ändern, wenn dies notwendig ist, um anschließend *Flughafen.TuHandlung()* aufzurufen, worauf wir in Kürze eingehen werden. Schließlich senden wir eine Nachricht an das Objekt mit dem Namen *FlugInfo*, um zu überprüfen, ob die Spielerin eines der Gates erreicht hat. In den meisten Methoden, die wir hier aufrufen, überprüfen wir, ob überhaupt Handlungsbedarf gegeben ist und führen demgemäß entweder die Handlung aus oder kehren einfach zum Hauptprogramm zurück.

13.11 Die Arbeitsmethode

Beispielprogramm: PLAN.CPP

Die einzige Methode, die wir noch nicht implementiert haben, ist die interessanteste. Die Funktion *TuHandlung()* beginnt in Zeile 183 der Datei PLAN.CPP. Sie schaut sich *Verb* und *Substantiv*, so es sie gibt, an und läßt die adäquate Aktion ablaufen. Wenn Du die **else if**-Anweisungen der Reihe nach durchgehst, wird es sogleich klar sein, welche Eingabe welche Aktion bewirkt. Für viele der Aktionen müssen noch gewisse Voraussetzungen erfüllt sein, bevor sie durchgeführt werden. So ist es etwa nicht möglich, Konfekt zu kaufen, wenn Du kein Geld hast, am Ort kein Konfekt ist oder der Ort nicht die Snackbar ist.

Am Ende dieser Methode, in Zeile 277, findet sich die Standardaktion, wenn sonst nichts passiert ist. Wir nehmen an, daß die Spielerin Gebrauch ihrer Kreativkraft gemacht hat, und die Snackbar nehmen wollte oder trocken wie treffend „Aus -einz!“ eingegeben hat.

13.12 Schlussbemerkungen zu Abflug

Nun, da Du das Spiel gespielt und den Code dazu studiert hast, solltest Du verstanden haben, wie ein solches Programm geschrieben werden kann. Natürlich gibt es unzählige Variationen dieses Themas.

Du hast jetzt vielleicht das Gefühl, daß diese Methode der Implementation fix und fertig vom Himmel gefallen ist oder doch zumindest durch spirituelle Eingebung offenbart wurde. Das ist überraschenderweise nicht so. Es gibt einige (unrühmliche) Vorgängerversionen dieses Programmes, die wieder verworfen wurden. Beim Update von Version 2.0 des Tutorials auf 2.2, wurde das Programm restrukturiert. In Version 2.0 waren ungefähr 50% des Code in Klassen, jetzt aber sind es circa 98%.

Für objektorientiertes Programmieren ist dasselbe vorausschauende Denken vonnöten wie auch für Nicht-Objekt-orientiertes-Programmieren, der objektorientierte Compiler hilf Dir aber beim Schreiben des Codes und bei der Fehlerbereinigung, da er viele der Fehler findet, die in unseren Programmen zu verstecken wir wahre Meister sind.

13.13 Dein Projekt

Diese Aufgabe soll Dir ein wenig Erfahrung im Umgang mit einem relativ großen Projekt vermitteln.

Erweitere das Programm um einen Koffer, der bei der Ankunft am Flughafen im Auto liegt. Der Koffer muß eingechecked werden, bevor die Sicherheitskontrolle passiert werden kann. Davon sind einige Klassen betroffen. So mußt du etwa das folgende tun:

1. Erweitere die Wortliste um „Koffer“ und „check“. Das muß natürlich an der richtigen Stelle passieren.
2. Füge der Klasse *Gegenstaende* den Koffer hinzu.
3. Initialisiere die Gegenstände im Auto (mit dem Koffer).
4. Überprüfe beim Durchschreiten der Sicherheitskontrolle, daß die Spielerin auch den Koffer nicht bei sich trägt. Die Strafe bleibt Dir überlassen (aber muß es immer ein Massaker sein?)
5. Überprüfe im Flieger, ob die Spielerin auch den Koffer eingechecked hat. Sollte sie das nicht getan haben, sei kreativ!

Wenn Du das Spiel erfolgreich um den Koffer erweitert hast, kannst Du Dich einmal zurücklehnen und befriedigt feststellen, daß es (alles) doch einen Sinn hat (nur: welchen?

Zwar ist alles egal, das Wunderbare aber ist, daß es (uns) egal ist (sein muß), daß dem so ist); Du hast verstanden, wie das Programm arbeitet und wie Objekte miteinander Aufgaben erfüllen.

Dann solltest Du Dir ein Projekt überlegen, das objektorientierte Programmiertechniken verwendet und sogleich damit beginnen. Der beste Weg, etwas zu lernen, ist immer noch, es zu tun.

Viel Spaß!

Abbildungsverzeichnis

4.1 Speicherinhalt nach Zeile 13 19

6.1 Speicherzustand nach Zeile 17 34
6.2 Eine Mauer schützt die Daten 35
6.3 Speicherinhalt nach Zeile 30 36
6.4 Speicherinhalt nach Zeile 34 39
6.5 Die beiden Objekte von KLASMAST.CPP 40

7.1 Variablen in OBJARRAY.CPP 53
7.2 Speicherinhalt nach Zeile 70 57
7.3 Geschachtelte Objekte 59

8.1 Die Klasse *Vehikel* 67
8.2 Ein Objekt der Klasse *Auto* 70
8.3 Die Klasse *Laster* 71