



C++ ist im Wesentlichen eine Programmiersprache wie andere auch, im Gegensatz zu diesen aber weit verbreitet: Praktisch alle größeren, kommerziellen Anwendungen werden in dieser Sprache implementiert. Sie baut auf einem standardisierten und gut bekannten Fundament auf, und zwar auf der Sprache C.



Einführung in C++

Crashkurs

THOMAS WÖLFER

C ist eine sehr einfache Sprache, die nicht viele Sprachfeatures bietet: Sie setzt sich im Wesent-

QUICK INDEX

- **Klassen definieren die Form der Objekte**
Datenstrukturen und die zugehörigen Funktionen definieren eine Klasse
- **Klasse als Typ**
Fast wie Variablen: Objekte vom Typ einer Klasse
- **Konstruktor und Destruktor**
Methoden, die Objekte ins (Speicher-) Leben rufen und wieder entfernen
- **Klassenbibliotheken**
Nutzen Sie die vorgefertigten Klassen aus Bibliotheken
- **Vererben und Basisklassen**
Aus vorhandenen Klassen neue und detailliertere Klassen ableiten
- **Vererben kann gefährlich sein**
Vorsicht beim Mischen der Eigenschaften mehrerer Basisklassen
- **Vererbung bringt Gewinn**
Einmal definierte Funktionalitäten können in abgeleiteten Klassen aufgerufen werden
- **Überladene Operatoren**
Definieren Sie +, -, =, ... neu

lichen aus Makros, Zeigern, Strukturen und Funktionen zusammen. Egal was in C implementiert wird, die Implementierung wird immer wieder auf genau diesen Makros, Zeigern, Strukturen und Funktionen aufbauen.

C++ ist anders. Es bietet eine Vielzahl von Sprachfeatures. Da C++ auf C aufbaut, sind die Makros, Zeiger, Strukturen und Funktionen noch da, aber es gibt auch Klassen mit privaten und geschützten (protected) Mitgliedern (members), sowie die Möglichkeit, Funktionen zu überladen, Vorgabeparameter (default Parameter), Konstruktoren und Destruktoren, vom Programmierer definierte Operatoren, Referenzen und vieles mehr. In aller Kürze: C++ ist eine wesentlich umfangreichere Sprache als C und liefert dadurch erheblich mehr Angriffsfläche für Fehler, schlechte Implementierungen und ineffizienten Code. Allerdings bietet C++ mächtige Konstrukte, um gut lesbaren, bequem zu wartenden und effizienten Code zu programmieren: Alles eine Frage der Implementierung und der Fähigkeiten des Programmierers. Insofern funktioniert C++ nicht anders als andere Sprachen, es ist nur umfangreicher.

Der folgende C++-Crashkurs wendet sich in erster Linie an C-Program-

mierer, denn viele Elemente dieser Sprache müssen vorausgesetzt werden. Allerdings sollten auch Programmierer, die sich in anderen Programmiersprachen bewegen, in der Lage sein, dem Text ohne größere Probleme zu folgen: Die meisten Beispiele sind so gewählt, dass auch ein Basic- oder Pascal-Programmierer damit klarkommen sollte.

■ Klassen definieren die Form der Objekte

C++ ist eine objektorientierte Sprache, was sich an verschiedenen Stellen bemerkbar macht. Das erste und wichtigste Element in C++ ist das der ‚Klasse‘. Ein Seitenblick auf andere Sprachen verdeutlicht das Konzept: Oft will man innerhalb eines Programms eine Datenstruktur logisch zusammenfassen. In C würde man die abstrakte Definition einer Textzeile durch eine Struktur ausdrücken:

```
struct Line {  
    char* pc;  
    int cb;  
}
```

Eine Zeile wird durch einen Zeiger auf "char" - vereinfacht gesagt durch ein Array aus Zeichen - sowie durch eine Längenangabe beschrieben. Um mit einer solchen Zeile innerhalb eines Pro-



gramms zu arbeiten, implementiert der C-Programmierer verschiedene Funktionen: Diese erhalten Namen wie "PrintLine(Line* pLine)" zum Ausdrucken oder Anzeigen einer Zeile, "ReadLine(Line* pLine)" zum Einlesen einer Zeile aus einer Textdatei und so weiter. Alle diese Funktionen bekommen einen Zeiger auf eine Variable vom Typ Line. Dieser zeigt auf die Daten, mit denen die Funktion arbeiten soll. (Siehe dazu "Bsp1.c".)

■ Eine Klasse wird definiert

In C++ werden die Daten (die "C" Struktur) und die zugehörigen Funktionen zusammengefasst - das Resultat ist die Klassendefinition:

```
class Line {
    char* m_pc;
    int m_cb;
    bool ReadLine( FILE hSource);
    int PrintLine();
};
```

Bei einer Klasse handelt es sich um einen Mechanismus zur Ordnungsfindung: Zusammengehörige Datenstrukturen und die Funktionen, die diese Datenstrukturen manipulieren, werden dadurch mit einem Namen nutzbar.

Klassen eignen sich nicht nur zum Aufräumen im Quellcode. Eine Klasse definiert auch einen neuen Typ, der voll in die Sprache integriert ist. In vielen Programmiersprachen, und auch in C++, gibt es primitive Datentypen wie "Int"; "Char" oder "Double". Wenn man eine Variable vom Typ "int" - also Integer - benutzen möchte, schreibt man in C und C++ folgendes:

```
int a;
```

Danach gibt es eine Variable mit dem Namen "a" und dem Typ "int" - dieser Variable kann man einen Wert zuweisen:

```
a = 12;
```

■ Klasse als Typ

Die Definition einer Klasse erzeugt einen neuen Typ, der wie die eingebauten Typen verwendet werden kann. So legt der Ausdruck

```
Line a;
```

eine Variable vom Typ "Line" an. Auf die Daten innerhalb dieser Klasse (Daten-Member) kann man genauso zugreifen, wie man in C auf Elemente einer Struktur zugreifen kann. Wenn man

den char-Zeiger setzen möchte, schreibt man folgendes:

```
a.m_ps = "Hallo C++";
```

Dies ist eigentlich nicht der Sinn der Sache - normalerweise abstrahiert man Daten-Member in C++, so dass die Implementierung der Klasse verändert werden kann, wenn das zu einem späteren Zeitpunkt notwendig wird. Zu diesem Zweck definiert man Zugriffsfunktionen, mit denen die Daten erfragt und gesetzt werden können, wie etwa

```
SetLength() und GetLength()
```

um die Länge der Zeile zu setzen oder zu erfragen. (Der Zugriff auf die Daten kann mit weiteren Methoden auf verschiedenen Stufen feingeregelt werden - doch dazu später mehr.)

Zusätzlich zu diesem aus C bekannten Konzept hat eine Klasse Funktionen, die im Zusammenhang mit Klassen "Methoden" genannt werden: Im Beispiel sind dies die Methoden ReadLine() und PrintLine(). Die Funktionskörper dieser Funktionen müssen ebenfalls implementiert werden - und zwar entweder innerhalb der Klassendefinition oder an anderer Stelle. Für dieses Beispiel gehen wir davon aus, dass diese Implementierung bereits vorliegt. Um eine Textzeile auszugeben, wird folgendes Statement verwendet:

```
Line a;
// hier wird das "Line" Objekt im
// Verlaufe des Programms
// mit Daten gefüllt
a.PrintLine();
```

■ Konstruktor und Destruktor

Wichtig bei Methoden: Jede Klasse hat zwei sehr spezielle Methoden, die Konstruktor und Destruktor genannt werden. Der Konstruktor (davon kann es verschiedene Spielarten geben) dient der "Erzeugung" und Initialisierung einer Instanz vom Typ einer Klasse. Er wird aufgerufen, wenn eine Klasse instanziiert wird. Innerhalb der Klassendefinition erkennt man den Konstruktor daran, dass er den gleichen Namen wie die Klasse selbst hat. Die verschiedenen Spielarten eines Konstruktors ergeben sich aus der unterschiedlichen Herangehensweise, wie man eine Instanz einer Klasse erzeugt. Für das Line-Beispiel wäre dies die Konstruktion einer "leeren" Line und die Konstruktion einer Line anhand eines konstanten Strings:

```
class Line
{
    Line();
    Line( const char*);
};
```

Die Implementierungen der beiden Methoden hätten in etwa folgenden Aufbau:

```
Line::Line()
{
    // "leere" Line
    m_pc = 0;
    m_cb = 0;
}

Line::Line( const char* psz)
{
    int cb = strlen( psz) + 1;
    m_pc = new char[ cb];
    strcpy( psz, m_pc);
    m_cb = cb;
}
```

Wie man am zweiten Beispiel für den Konstruktor erkennt, ist hier für die Konstruktion der Klasse die dynamische Anforderung von Speicher erforderlich. (In C++ wird dynamischer Speicher mit "new" angefordert - anders als in C mit malloc sind die von "new" gelieferten Zeiger typbehaftet.)

In diesem Zusammenhang stellt sich die Frage, wo dieser Speicher wieder freigegeben wird: Dazu - bzw. generell zum Freigeben belegter Ressourcen (zum Beispiel offener Datei-Handles) - ist der Destruktor einer Klasse zuständig. Genau wie der Konstruktor hat der Destruktor den gleichen Namen wie die Klasse selbst, wird aber durch eine vorgestellte Tilde (~) gekennzeichnet:

```
Line::~Line()
{
    if( m_pc) delete[] m_pc;
}
```

Bei einer Klasse handelt es sich also um eine zusammengefasste Version der aus C bekannten Strukturen mit den zugehörigen Funktionen sowie der Möglichkeit, die Daten sinnvoll zu initialisieren (Konstruktor) und Ressourcen freizugeben (Destruktor). Wie schon erwähnt verhält sich eine Klasse in C++ wie ein eingebauter Datentyp bzw. die Instanz einer Klasse (ein Objekt vom Typ einer Klasse) so wie eine Instanz eines eingebauten Datentyps.

■ Klassen und Operatoren

Dazu müssen verschiedene Dinge gewährleistet sein. So ist es in C und in C++ möglich, Daten direkt Variablen zuzuweisen. Um einer Integer-Variablen einen Wert zuzuweisen, verwendet man folgenden Ausdruck:



```
int a;  
a = 12;
```

Wenn sich die "Line"-Klasse aus unserem Beispiel ebenso verhalten soll, wäre ein Ausdruck wie der folgende wünschenswert:

```
Line a;  
  
a = "Hallo C++";
```

Leider kann der C++-Compiler von Haus aus relativ wenig mit solch einem Ausdruck anfangen. Zwar handelt es sich offenbar um eine Zuweisung eines konstanten Strings an eine Variable vom Typ "Line", doch was soll mit den Daten geschehen? Werden sie kopiert oder nur der Zeiger? Auf welche Daten-Member wird sich die Zuweisung auswirken? All dies sind Fragen, die geklärt werden müssen.

Ebenso ist mit eingebauten Typen folgender Ausdruck zulässig:

```
int a,b,c;  
a = b = c = 12;
```

Dies bedeutet, dass auch der folgende Ausdruck möglich sein sollte:

```
Line a,b,c;  
a = b = c = "Hallo C++";
```

Die angesprochenen Fälle behandelt C++ dadurch, dass es dem Programmierer möglich ist, seine Klasse(n) flexibel an alle Bedürfnisse anzupassen - auch an die eben geschilderten. Bei normalen "eingebauten" Typen verhält es sich so, dass der Compiler weiß, was er mit Operatoren wie dem Zuweisungsoperator (=) anfangen soll. Damit diese Unterstützung auch für eigenen Klassen funktioniert, ist es dem Programmierer möglich, eigene Methoden zu definieren, welche die Aufgaben von Operatoren übernehmen. So kann man für die Line-Klasse einen Zuweisungsoperator definieren, auch alle anderen Operatoren wie der Additions- oder der Subtraktionsoperator können selbst programmiert werden. Man nennt diese Vorgehensweise "Überladen" eines Operators, sie wird in Büchern genau geschildert.

■ Klassenbibliotheken gibt es vorgefertigt

Bei C++ setzt sich das Konzept der Klasse auf anderer Ebene fort: Ein C++-Programmierer wird nur selten tatsächlich eine Line-Klasse (oder eine String-Klasse) implementieren, sondern statt dessen eine bereits fertige Klasse aus einer Klassenbibliothek verwenden.

Eine Klassenbibliothek ist eine

Sammlung komplett fertiger Klassen, die direkt verwendet werden kann. Bei C gibt es ein ähnliches Konzept - dort ist es die Funktionssammlung, die in Form der C-Runtime-Bibliothek mit jedem ernstzunehmenden Compiler ausgeliefert wird. Diese Runtime-Bibliothek gibt es auch in C++, zusätzlich verfügen die meisten Compiler über Klassenbibliotheken für verschiedenste Zwecke. Eine der am häufigsten benutzten - wenn auch heiß diskutierten - Klassenbibliotheken ist die MFC-Bibliothek (Microsoft Foundation Classes). Sie wird von praktisch allen gängigen kommerziellen Windows-Programmen als Grundlage verwendet und deckt die wichtigsten Elemente der Windows-Programmierung ab. So gibt es darin Klassen, die Fenster abbilden, solche, die beim Drucken benötigt werden, und Klassen, die für die Arbeit mit OLE/COM gedacht sind. Die Verwendung der MFC wird in den Workshops zu C++ an anderer Stelle in diesem Sonderheft behandelt.

■ Vererbung und Basisklassen

Das Konzept der Klassen und Klassenbibliotheken bringt ein weiteres Feature von C++ ans Licht: Natürlich will man einmal implementierte Funktionalität weiterverwenden, am besten so, dass sie erweiterbar ist. Dazu bietet C++ die "Vererbung" an, eine für objektorientierte Sprachen typische Herangehensweise.

Hierbei geht es darum, dass eine "neue" Klasse von einer anderen "abgeleitet" wird - dadurch "erbt" diese Klasse die Fähigkeiten der Klasse, von der sie abgeleitet wurde. Als Beispiel dient die Line-Klasse von zuvor: Angenommen, die Line-Klasse stammt aus einer Klassenbibliothek oder aus einer eigenen - schon älteren - Implementierung. Diese fertige Klasse ist ausgetestet, fehlerfrei und erfüllt alle Ansprüche, die an eine Line-Implementierung gestellt werden. Nun benötigt der Programmierer eine weitere Version einer solchen Klasse - nur soll diese zusätzlich zum Text und der Längenangabe noch einen Hash-Code enthalten. Damit wird bei der Benutzung sichergestellt, dass die Daten innerhalb der Zeile nicht korumpiert wurden, weil diese über eine serielle Verbindung übertragen werden sollen.

Es bieten sich unterschiedliche Möglichkeiten an: So kann der Originalcode um die neue benötigte Funktionalität erweitert werden. Die bisherige Line-



Klasse wird um zwei Methoden - zum Beispiel GetHashCode() und BuildHash() - erweitert werden. Eine zweite Möglichkeit ist das Kopieren der Original-Implementierung und die Erweiterung der Kopie um die neue Funktionalität; diese erweiterte Implementierung kann für das neue Projekt verwendet werden.

Beide Herangehensweisen bergen jedoch große Nachteile: Im ersten Fall müssten alle Projekte, die die "alte" Implementierung von "Line" verwenden, neu übersetzt und einschließlich möglicher Seiteneffekte wieder getestet werden. (Was passiert zum Beispiel, wenn die Daten der Line-Klasse gespeichert werden? Schließlich hat die neue Implementierung weitere Daten-Member. Es eröffnet sich das Problem, dass zuvor gespeicherte Daten konvertiert werden müssen.) Beim zweiten Ansatz hat man das Problem, dass auch bei einer noch so gut getesteten Klasse irgendwann ein Fehler auftritt - natürlich direkt, nachdem die Kopie der Klasse angelegt wurde: Nun sind zwei unterschiedliche Codebasen mit mehr oder minder gleicher Funktionalität zu warten.

Der Ansatz von C++ geht einen anderen Weg: Man belässt die bisherige Klasse sowohl in ihrer jetzigen Art als auch an ihrer aktuellen Position und leitet statt dessen eine neue Klasse ab. Die neue Klasse kann um die benötigte Funktionalität erweitert werden, ohne dass sich dies in irgendeiner Form auf die vorhandene Implementierung (oder auf die Stellen, an denen diese benutzt wird) auswirkt.

■ Vererben kann gefährlich werden

Beim Vererben von Klassen existieren verschiedene Modi, die im wesentlichen festlegen, in welcher Art die "abgeleitete" Klasse auf die Daten der "Basisklas-



se" zugreifen darf. Welche Auswirkungen diese Modi haben, ist - für dieses Beispiel nicht wichtig, füllt das Wort "public" in der folgenden Codesequenz unseren Zweck zunächst:

```
class LineWithHash : public Line
{
    long m_Hash;
    long GetHash();
    void BuildHash();
    bool CompareHash();
};
```

Die abgeleitete Klasse verfügt nun sowohl über die Fähigkeiten der Basis-klasse als auch über die erweiterten neuen Fähigkeiten. Es ist also möglich, folgenden Ausdruck zu verwenden:

LineWithHash a;

```
a.PrintLine(); // Funktionalita
et der basis-klasse
a.BuildHash(); /// neue Funktionalitaet
```

Die Vererbung ist eines der komplizier-testen Features von C++ bzw. eines, das recht schnell zu sehr komplizierten Situationen führen kann. Nachdem es sich hier nur um eine Einführung handelt, wird auf besondere Problematiken nicht weiter eingegangen. Es soll jedoch erwähnt werden, dass die Vererbung auch mit mehreren Objekten auf einer Ebene durchgeführt werden kann. Eine Klasse erbt dabei von mehreren Basis-klassen. In der Theorie hört sich das un-

geheuer praktisch an - man stelle sich aber eine Klasse Namens "Automobil" vor, die gleichzeitig von den Klassen "Verkehrsmittel", "Motorfahrzeug" und "Transporter" abgeleitet ist. Alle Basisklassen haben verständlicherweise eine Methode "Hubraum" und eine Methode "MaxAnzahlInsassen". Welche Implementierung wird nun verwendet? Multiple Vererbung hat noch jede Menge weitere Fallen auf Lager: Bevor man selbige einsetzt, sollte man sich in C++ sehr sicher im Sattel fühlen.

■ Vererbung ist Gewinn

Die Vererbung von Klassen hat einen sehr positiven (und gewollten) Neben-

BEISPIEL 1

```
/*
 bsp1.c - minimalistisches beispiel fuer die verwendung
 einer struktur in 'c' mit zwei funktionen die diese
 struktur verwenden.
*/

#include <stdio.h>
#include <conio.h>
#include <malloc.h>

typedef struct TAGLine {
    char* pc;
    int cb;
} Line;

void InitLine( Line* pLine);
void PrintLine( Line* pLine);
void ReadLine( Line* pLine);
void DeleteLine( Line* pLine);

#define M_READ 1
#define M_PRINT 2
#define M_QUIT 3
int Menu()
{
    int c;
    puts("Line ein(1)esen, Line (a)usgeben, (e)nde\n");
    while( c = getch())
    {
        if( c == '1') return M_READ;
        if( c == 'a') return M_PRINT;
        if( c == 'e') return M_QUIT;
    }

    return M_QUIT;
}

int main()
{
    Line l;
    int m;

    m = Menu();
    while( M_QUIT != m)
    {
        switch( m)
        {
            case M_READ:
                ReadLine( &l);
                break;
            case M_PRINT:
                PrintLine( &l);
                break;
        }
    }
}
```

```
        m = Menu();
    }

    DeleteLine( &l);
}

void PrintLine( Line* pLine)
{
    if( pLine->cb)
    {
        int i;
        for( i=0; i<pLine->cb; i++)
            putchar( pLine->pc[ i]);
        putchar( '\n');
    }
}

void ReadLine( Line* pLine)
{
    char buffer[128];
    int i;
    char c;

    if( pLine->cb) puts("ACHTUNG: Line noch nicht
    gelöscht!\n");

    while( c = getch())
    {
        i++;
        if( i == 127) puts("Zu viele ZEichen...\n");
        else
        {
            buffer[ i] = c;
        }
    }

    pLine->pc = malloc( i);
    pLine->cb = i;
}

void InitLine( Line* pLine)
{
    pLine->pc = 0;
    pLine->cb = 0;
}

void DeleteLine( Line* pLine)
{
    if( pLine->cb)
    {
        free( pLine->pc);
        pLine->cb = 0;
    }
}
```



effekt: Man kann Zeiger auf eine Basis-klasse verwenden, um Funktionalität in abgeleiteten Klassen aufzurufen. Was das soll, erläutert das folgende Beispiel, allerdings diesmal mit einem Satz an Geometrie-Klassen.

Angenommen, geometrische Objekte (Kreis, Viereck, Dreieck) sollen im Quellcode abgebildet werden. Und angenommen, im Zuge des Programms sollen diese Elemente am Bildschirm dargestellt werden. Mit normalem C-Code würde man drei Strukturen defi-

nieren. Passend zu diesen Strukturen gäbe es drei Ausgabefunktionen:

```
DrawViereck( Rechteck* p);
DrawKreis( Kreis* );
DrawDreieck( Dreieck* );
```

Diese Herangehensweise macht die Verwaltung der einzelnen Objekte aufwendig: Hat man mehrere Objekte von jedem Typ, so benötigt man im Prinzip eine Liste für die Verwaltung von Dreiecken, eine für Vierecke und so weiter. Um dann alle Elemente anzuzeigen,

BEISPIEL 2

```
// Bsp2.cpp : Defines the entry point for the console
application.
//

#include "stdafx.h"
#include <stdio.h>
#include <string.h>

class Line {
public:
    Line();
    Line( const char* );
    ~Line();

    const Line& operator+=( const Line& l);
    const Line& operator+=( const char* psz);

    void Print();

    // die folgenden member sollten private sein,
    // doch das geht fuer beispiel zu weit
    char* m_pc; // die zeichen
    int m_cb; // die laenge

    // achtung: eigentlich benoetigte diese klasse
    // auch noch einen operator=.... !
};

// operator== - auf gleichheit testen
bool operator==( const Line& l1, const Line& l2)
{
    if( ! strcmp( l1.m_pc, l2.m_pc)) return true;
    return false;
}

// addition und zuweisung (mit line)
const Line&
Line::operator+=( const Line& l)
{
    m_cb += l.m_cb;
    char* pTemp = new char[ m_cb];
    strcpy( pTemp, m_pc);
    strcat( pTemp, l.m_pc);
    delete[] m_pc;
    m_pc = pTemp;
    return *this;
}

// addition und zuweisung (mit cont char)
const Line&
Line::operator+=( const char* psz)
{
    int cb = strlen( psz) + 1;
    m_cb += cb;
    char* pTemp = new char[ m_cb];
    strcpy( pTemp, m_pc);
    strcat( pTemp, psz);
    delete[] m_pc;
    m_pc = pTemp;
    return *this;
}
```

```
Line::Line()
{
    m_pc = 0;
    m_cb = 0;
}

Line::Line( const char* psz)
{
    m_cb = strlen( psz) + 1;
    m_pc = new char[ m_cb];
    strcpy( m_pc, psz);
}

Line::~Line()
{
    puts("Im Destruktor!\n");
    if( m_pc) delete[] m_pc;
}

void
Line::Print()
{
    // primitves error-handling
    if( ! m_pc) puts( "FEHLER: LEERE ZEILE\n");
    else puts( m_pc);
}

void Test()
{
    puts("1. Erzeugen von ein paar Line-Objekten\n");
    Line a( "hallo, ich bin zeile a");
    Line b( "hallo, ich bin zeile b");
    Line c( "hallo, ich bin zeile c");
    Line d( "hallo, ich bin zeile a"); // d ist iden-
    tisch mit a

    puts("2. Vergleich von Line-Objekten\n");
    if( a == b ) puts("Line a ist gleich Line b\n");
    else puts( "Line a ungleich Line b\n");
    if( a == d) puts("Line a ist gleich line d\n");
    else puts( "Line a ist ungleich line d\n");

    puts("3. Operator += verwenden\n");
    a += b;
    c += "Konstanter Text";
    puts("Hier kommt a:\n");
    a.Print();
    puts("Hier kommt c:\n");
    c.Print();

    puts("4. Das Programm wird beenden, die Lines
    verlieren scope und ihre destruktoren werden
    aufgerufen:\n");
}

int main(int argc, char* argv[])
{
    Test();
    return 0;
}
```



müsste zunächst über die Liste der Dreiecke, dann über die der Vierecke und schließlich über die Liste der Kreise iteriert werden.

In C++ löst man ein derartiges Problem durch eine Basisklasse und eine Liste, die Elemente vom Typ der Basisklasse aufnehmen kann.

```
class Geo
{
    Geo();
    virtual void Draw() = 0;
};
```

Diese Basisklasse Geo verfügt über einen Konstruktor sowie über die Methode "Draw()". Diese ist - und das ist neu für diese Einführung - als "= 0" ge-

kennzeichnet. Diese Kennzeichnung besagt, dass die Methode "Draw()" innerhalb der Klasse Geo nicht implementiert wird - deshalb können Instanzen von Geo auch nicht erzeugt werden. Klassen, die mindestens eine Funktion = 0 enthalten, sind nur als Basisklassen zum Ableiten zu gebrauchen. Solche Klassen nennt man "abstrakte Basisklassen". Allerdings: Zeiger auf Objekte vom Typ der Basisklasse können sehr wohl verwendet werden, und dies passiert weiter unten im Beispiel. (Eine Basisklasse muss nicht notwendigerweise abstrakt sein, auch für das Beispiel wäre es nicht erforderlich gewesen, Draw = 0 zu definieren. Allerdings konnte so

BEISPIEL 3

```
// Bsp3.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <afxtempl.h>

class Geo {
public:
    Geo() {} ;
    virtual void Draw() = 0;
};

class Viereck : public Geo {
public:
    Viereck() { /* tut nichts */ }
    void Draw() { puts("Viereck!"); }
};

class Dreieck : public Geo {
public:
    Dreieck() { /* tut nichts */ }
    void Draw() { puts("Dreieck!"); }
};

class Kreis : public Geo {
public:
    Kreis() { /* tut nichts */ }
    void Draw() { puts("Kreis!"); }
};

int main(int argc, char* argv[])
{
    // liste die die mfc klassen verwendet.
    CTypedPtrList<CPtrList, Geo*> list;

    Viereck v1;
    Viereck v2;
    Kreis k1;
    Kreis k2;
    Dreieck d1;
    Dreieck d2;

    list.AddTail( &v1);
    list.AddTail( &k1);
    list.AddTail( &d1);
    list.AddTail( &v2);
    list.AddTail( &k2);
    list.AddTail( &d2);

    for( POSITION pos = list.GetHeadPosition(); pos;)
    {
        Geo* pGeo = list.GetNext( pos);
        pGeo->Draw();
    }

    return 0;
}
```



der Begriff der abstrakten Klassen leicht eingeführt werden.) Nachdem Geo-Objekte nicht instanziiert werden können, müssen die tatsächlichen Geometrie-Elemente nun von Geo abgeleitet werden. Alle erhalten auch eine "echte" Draw-Funktion, die in diesem Beispiel der Einfachheit halber nur einen passenden Text ausgeben:

```
class Dreieck : public Geo {
    Dreieck();
    void Draw() { printf("Drei
eck"); }
};

class Viereck : public Geo {
    Viereck();
    void Draw() { printf("Vier
eck"); }
};
```

Für die Verwaltung der Elemente wird eine Klasse vom Typ "GeoList" benötigt; diese Liste kann Zeiger auf Elemente von Typ "Geo" aufnehmen, und zwar mit der Methode "AddTail()".

Nachdem die "echten" Geometrie-Objekte alle auch vom Typ "Geo" sind - schließlich wurden sie davon abgeleitet -, können sie auch in dieser Liste verwaltet werden. Der folgende Code ist also möglich:

```
Dreieck d;
Viereck v;
Kreis k;
GeoList l;
l.AddTail( &d);
l.AddTail( &v);
l.AddTail( &k);
```

Es gibt nun die Möglichkeit, drei unterschiedliche Objekte in einer gemeinsamen Liste zu verwalten. Praktischerweise ist diese Liste eine Liste aus Geo-Objekten - und diese haben, wie bei der Klassendefinition von "Geo" angegeben, eine Draw()-Methode. Iteriert man nun über die Liste, so erhält man bei jedem Iterationsschritt einen Zeiger auf ein Geo-Objekt. Dieser zeigt in Wirklichkeit einmal auf ein Dreieck, einmal auf ein Viereck und einmal auf einen Kreis. Um alle Elemente - obwohl sie von unterschiedlichem Typ sind -

auszugeben, wäre folgender Code möglich. (Die Iterationsmöglichkeit der Liste muss man sich dazudenken)

```
while( ! l.AtEnd()) // solange
nicht ende der liste
{
    Geo* p = l.GetNext();
    //nächsten Geo-Zeiger aus der
    Liste holen
    p->Draw(); // und Objekt an-
    zeigen.
}
```

Allein diese Möglichkeit macht C++ viel produktiver als C. Der Umstieg lohnt sich.

■ Überladene Operatoren (Operator Overloading)

Es wurde bereits angesprochen, dass sich eine C++-Klasse für den Compiler ganz so darstellt wie die eingebauten Typen "int" oder "char". Damit dies gelingt, müssen bestimmte Voraussetzungen gewährleistet sein. So müssen die normalen C++-Operatoren auf diese Klassen anwendbar sein.

Dabei muss der Programmierer bedenken, wie bestimmte Operatoren funktionieren sollen. Als Beispiel sei der "="-Operator (Vergleich) genannt. Angenommen, man hat zwei Instanzen vom Typ "Line". Wann sind diese identisch? Dafür gibt es schließlich mehrere Möglichkeiten:

- Zwei Line-Objekte können als identisch betrachtet werden, wenn sie gleich lang sind und den gleichen Inhalt haben.
- Zwei Line-Objekte können als identisch betrachtet werden, wenn sie auf den gleichen Speicherbereich zeigen.
- etc.

Letztlich muss diese Entscheidung vom Programmierer der Klasse getroffen werden, denn der Programmierer ist für die Implementierung der Operatoren zuständig. Ein möglicher Vergleichsoperator für die Line-Klasse (in diesem Fall einer, der Gleichheit bei identischem Inhalt annimmt) hat folgenden Aufbau:

```
bool operator==( const Line& L1,
const Line& L2)
{
    if( ! strcmp( L1.m_pc,
L2.m_pc)) return true;
    return false;
}
```

Ein selbstdefinierter Additions-Zuweisungsoperator (+=) für Line-Objekte könnte hingegen folgenden Aufbau haben:

```
Line operator+=( const Line& l)
{
    strcat( m_pc, l->m_pc);
    m_cb = strlen( m_pc);
    return *this;
}
```

Das Interessante an der Definition eigener Operatoren ist, dass man damit das Verhalten einer Klasse im Zusammenspiel mit anderen Typen festlegen kann. So liegt es nahe, einen Additionsoperator zu definieren, mit dem Elemente vom Typ "const char*" auf Elemente vom Typ "Line" addiert werden können.

Wie man allerdings bereits am vorhergehenden Beispiel sieht, ist auch hier viel Vorsicht und Feingefühl geboten: Bereits bei der Addition von Strings ist nicht mehr eindeutig, was im Programmcodem gemeint ist, wenn irgendwo ein Statement wie das folgende auftaucht:

```
Line a;
Line b;
//..... hier passiert etwas mit den
beiden Variablen
Line c = a+b;
```

Um die letzte Zeile zu verstehen, muss man sich im Prinzip die Definition des Operators + ansehen. Zwar ist es bei Line-Objekten naheliegend, dass die darin enthaltenen Zeilen aneinandergehängt werden, doch ob dies tatsächlich passiert, ist letztlich davon abhängig, wie der Operator implementiert wurde. Grundsätzlich spricht nichts dagegen, dass das Resultat des Operators die bitweise Addition der Strings ist - oder einfach der konstante String "Hallo". Beim Überladen von Operatoren lässt man besser Feingefühl und Vorsicht walten.

Lehrbücher zu C++ oder für Umsteiger von C füllen ganze Bücherregale. Dementsprechend ist klar, dass längst nicht alle Features, sämtliche Vor- und Nachteile in diesem Crashkurs angesprochen werden konnten. Vielleicht hat der Beitrag aber Lust auf mehr gemacht - die zugehörigen Quellcodebeispiele können als erste Startpunkte dienen. Wer mehr lernen will und Hilfe sucht, findet diese im Internet unter http://www.nickles.de/category_21.html.

UR



QUELLCODES

Die kompletten Beispiele: Bsp1, Bsp2 und Bsp 3 finden Sie unter www.pc-magazin.de/download/Special_21/Crashkurs/