



Objekte und Klassen

Das Beispiel in C++

Bei den **bisherigen** Projekten haben Sie eher die **C-nahen** Elemente von C++ **verwendet**, nun kommen die **C++-Elemente** ins Spiel.

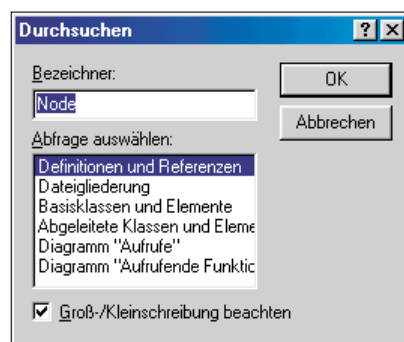
THOMAS WÖLFER

Bisher haben Sie rein funktionsorientiert gearbeitet: Alle Aufgaben wurden in separaten Funktionen abgearbeitet, und dabei hatten die einzelnen Funktionen ein gewisses Maß an Wissen über die verwendeten Daten. Zum Beispiel wurde ein "struct" Namens "Node" verwendet, dessen Member alle öffentlich zugänglich waren. Das hat seine Vorteile – aber auch Nachteile. Ein Nachteil ist zum Beispiel der, dass ein Fehler im Quellcode schnell dazu führen konnte, dass eines der Member einer der beteiligten Structs überschrieben wurde: Das "Struct" kann sich nicht dagegen wehren.

Ein weiterer Nachteil ist der, dass die verwendeten Funktionen und die verwendeten Daten nicht logisch gruppiert waren. Das erkennt man daran, dass zwar eine verkettete Liste implementiert wurde – aber nirgendwo eine Variable vom Typ "Liste" zum Zuge kam. Die komplette Implementierung fand in Form von Speicherverwaltung per *new* und *delete* sowie dem Verändern der Member der Node-Struktur statt.

Das kann man wesentlich besser machen – und zwar mit C++-Klassen. Im

Wesentlichen ist eine Klasse dabei ein Element, das sich aus Daten und Funktionen zusammensetzt. Dabei versucht man zum einen die Daten vor dem Zugriff von



DER QUELLCODE-BROWSER bietet Ihnen eine ganze Reihe zusätzlicher Ansichten zum Quellcode. Zum Beispiel können Sie untersuchen, wo welche Symbole definiert wurden.

außen zu schützen, zum anderen bemüht man sich darum, dass die Funktionen der Klasse alles abdecken, was an zugehöriger Funktionalität für die Arbeit mit den Daten geboten sein muss. (Viele Hintergründe und tiefergehende Erklärungen dazu finden Sie im PC-Magazin Special 21 "C++ und Java".)

Eine Klasse verhält sich dabei in C++ so, wie ein eingebauter Datentyp. Zumindest kann man eine Klasse

so programmieren, dass sie das tut – und zwar Vergleiche, Zuweisungen und eingebaute Operatoren eingeschlossen.

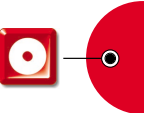
Das würde für dieses Beispiel allerdings ein wenig zu weit führen, daher sind die vorgestellten Klassen eher einfach gehalten und nutzen nur sehr wenige der in C++ gebotenen Möglichkeiten. Die Instanz einer Klasse nennt man ein Objekt, und die in einer Klasse eingebetteten Funktionen nennt man Methoden des Objekts.

Ein Beispiel für eine sinnvolle Klasse ist die Klasse der Strings, also Zeichenketten. (Nicht umsonst gibt es in praktisch jeder C++-Klassenbibliothek für diesen Zweck fertige, wieder verwendbare Klassen.) Eine ganz einfache String-Klasse könnte zum Beispiel den Aufbau haben, den Sie in Listing 1 vorfinden.

LISTING 1

```
class String
{
public:
    String( char* s)
    {
        m_pszData = new char[ strlen
            ( s)+1];
        strcpy( m_pszData, s);
    }
    ~String()
    {
    }
    bool IsEqual( char* s)
    {
        return ! strcmp( s, m_pszData);
    }

private:
    char* m_pszData;
};
```

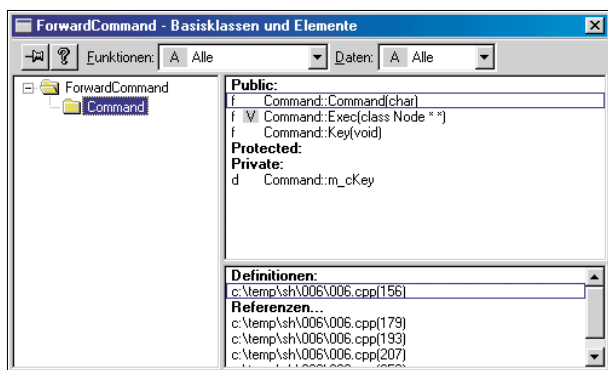


Sie sehen hier zunächst einmal das bisher noch nicht verwendete Schlüsselwort "class". Das tut im Wesentlichen das Gleiche, wie das Schlüsselwort "struct", das Sie schon kennen – nur wird keine Struktur, sondern eine Klasse definiert. Dann sehen Sie zwei weitere neue Schlüsselwörter, und zwar "public" und "private", wobei das zweite etwas weiter unten in der Klassendefinition verwendet wird. Mit diesen beiden Schlüsselwörtern können Sie die Member der Klasse in verschiedene Bereiche unterteilen: Solche, die von außerhalb der Klasse benutzbar sein sollen und solche, bei denen das nicht der Fall ist. Um das zu verdeutlichen nochmals ein kurzer Rückblick auf die "structs". Hier waren alle Members von außen einfach im Zugriff.

```
struct Node
{
    Node* pNext;
    char* pLine;
};

void foo()
{
    Node n;
    n.pLine = 0;
}
```

Sie konnten also mit dem "Punkt"-Operator einfach auf die einzelnen Member der Struktur zugreifen – egal, von welcher Stelle. Bei Klassen ist das anders:



WELCHE KLASSE DIE BASISKLASSE für welche andere Klasse ist, können Sie schnell mit dem Quellcode-Browser herausfinden.

Hier können Sie nur auf die Members zugreifen, die als "public" markiert sind. Die anderen Members sind nur innerhalb der Klasse selbst verwendbar. Bei der String-Klasse könnten Sie also nicht einfach Folgendes hinschreiben, denn das würde der Compiler bemängeln:

```
void foo()
{
    String s;
    s.m_pszData = 0;
}
```

Konstrukoren und Destruktoren

Die einfache String-Klasse hat von oben noch drei Methoden: Eine, die den Namen der Klasse trägt, eine mit dem Namen der Klasse und einer vorangestellten Tilde (~) sowie eine mit dem Namen *IsEqual()*.

Eine Funktion, die den Namen der Klasse trägt, nennt man Konstruktor. Der Konstruktor ist die Methode, die dann aufgerufen wird, wenn Sie eine Instanz der Klasse erzeugen. Wenn Sie also beispielsweise

```
String s("Hallo String");
```

in Ihrem Quellcode verwenden, wird der Konstruktor der String-Klasse aufgerufen. Dabei ist es möglich, mehrere Konstruktoren in einer Klasse zu haben, solange diese unterschiedliche Parameter erhalten. Im Falle des String-Beispiels kopiert der Konstruktor den übergebenen String einfach in das private Member namens *m_pszData*.

Die Methode mit dem gleichen Namen wie die Klasse, die zusätzlich eine vorangestellte Tilde hat, nennt man Destruktor. Der Destruktor wird dann aufgerufen, wenn die Instanz des Objektes den Scope verliert – das heißt, wenn das Objekt seine Gültigkeit verliert. Das ist in C++ mehr oder minder identisch mit C, nämlich dann, wenn der durch die geschweiften Klammern markierte Code-Block, in dem das Objekt erzeugt wurde, verlassen wird. Im Destruktor müssen Sie sich darum kümmern, dass Ressourcen, die vom Objekt verwendet wurden, wieder frei-

gegeben werden, denn nach dem Aufruf des Destruktors ist das Objekt nicht länger vorhanden – und kann sich dann auch nicht mehr ums Aufräumen kümmern.

Die Methode *IsEqual()* ist hingegen eine ganz normale Methode – im Rahmen des Beispiels liefert diese Methode einfach einen bool'schen Wert, der angibt, ob der String mit einem bestimmten Text inhaltlich identisch ist. Sie könnten die String-Klasse al-

so beispielsweise wie in Listing 2 verwenden.

LISTING 2

```
void foo()
{
    String s("ende");

    char sz[128];
    printf("Text eingeben");
    scanf(" %s", &sz[0]);
    if(s.IsEqual(sz))
        printf("s ist mit dem eingegebenen
        ➔ Text identisch.");
}
```

Klassen haben noch eine weitere Besonderheit: Man kann von ihnen erben. Erbt eine Klasse von einer anderen, ist die Funktionalität, die in der Basisklasse (so nennt man die von der geerbt wird) automatisch in der abgeleiteten Klasse enthalten. Das Beste dabei: Man kann Methoden so definieren, dass man einen Zeiger auf eine Basisklasse zum Aufruf von Funktionen in einer abgeleiteten Klasse verwenden kann. (Was das genau bedeutet, erfahren Sie etwas später im Beitrag.)

Das File-Viewer Beispiel in C++

Man kann nun eine ganze Menge an Dingen in Klassen kapseln – aber für eine C++-Implementierung des File-Viewers brauchen Sie nicht sehr viel. Benötigt werden:

- Eine Node-Klasse für die verkettete Liste,
- eine Liste-Klasse,
- eine Kommando-Klasse; das Menü setzt sich dann aus Kommandos zusammen sowie
- von der Kommando-Klasse abgeleitete Klassen, die die tatsächlichen Kommandos implementieren

(Den fertigen Quellcode finden Sie auf der CD zu diesem Sonderheft.)

Zunächst einmal zur Listen-Klasse und der zugehörigen Node-Klasse. Zur besseren Übersicht finden Sie den vereinfachten Quellcode zu diesen beiden Klassen im Kasten: "Listen und Knoten".

Die Knoten-Klasse kommt mit den gleichen Daten-Members daher, die Sie auch schon in der Node-Struktur vorgefunden haben: Es gibt einen Zeiger



LISTEN UND KNOTEN

```
(class Node
{
public:
    Node()
    {
        m_pszLine = 0;
        m_pPrev = m_pNext = 0;
    }

    void SetPrev( Node* p)
    {
        m_pPrev = p;
    }

    void SetNext( Node* p)
    {
        m_pNext = p;
    }

    void SetText( char* pszLine)
    {
        if( m_pszLine) delete[] m_pszLine;

        m_pszLine = new char[ strlen( pszLine) + 1 ];
        strcpy( m_pszLine, pszLine);
    }

    char* GetLine()
    {
        return m_pszLine;
    }

    Node* GetNext()
    {
        return m_pNext;
    }

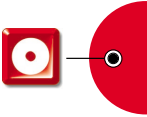
    Node* GetPrev()
    {
        return m_pPrev;
    }

    ~Node()
    {
        delete[] m_pszLine;
    }
private:
    char* m_pszLine;
    Node* m_pPrev;
    Node* m_pNext;
};

class List
{
public:
    List()
    {
        m_pRoot = 0;
        m_pCurrent = 0;
    }

    ~List()
    {
        Node* n = m_pRoot;

        while( n->GetNext())
        {
```



LISTEN UND KNOTEN (FORTSETZUNG)

```

        Node* nDelete = n;
        n = n->GetNext();
        delete nDelete;
    }
}

void Add( Node* p)
{
    if( ! m_pCurrent)
    {
        m_pRoot = p;
        m_pCurrent = p;
        m_pCurrent->SetPrev( 0);
        m_pCurrent->SetNext( 0);
    }
    else
    {
        m_pCurrent->SetNext( p);
        p->SetPrev( m_pCurrent);
        p->SetNext( 0);
        m_pCurrent = p;
    }
}

Node* GetNext()
{
    // immer nur bis zum letzten
    if( m_pCurrent->GetNext())
    {
        m_pCurrent = m_pCurrent->GetNext();
        return m_pCurrent;
    }

    return 0;
}

Node* GetPrev()
{
    if( m_pCurrent->GetPrev())
    {
        m_pCurrent = m_pCurrent->GetPrev();
        return m_pCurrent;
    }

    return 0;
}

Node* GetCurrent()
{
    return m_pCurrent;
}

Node* GetRoot()
{
    return m_pRoot;
}

void SetRootAsCurrent()
{
    m_pCurrent = m_pRoot;
}

private:
    Node* m_pCurrent;
    Node* m_pRoot;
};

```



auf die Daten (also den Text), einen Zeiger auf den vorhergehenden Knoten und einen Zeiger auf den folgenden Knoten. Anders als bei der Node-Struktur sind diese Members aber hier "private" und damit von außen nicht zu erreichen.

Dafür gibt es aber die passenden Methoden mit den Namen *GetLine()*, *GetNext()* und *GetPrev()*, die die entsprechenden Informationen liefern. Außerdem hat die Node-Klasse noch

Methoden zum Setzen dieser Members, die die Namen *SetText()*, *SetNext()* und *SetPrev()* haben. Ferner gibt es noch einen Konstruktor, der die Zeiger auf 0 setzt damit diese einen bekannten Wert haben und einen Destruktor, der sich um die Freigabe des Speichers für den Text kümmert. Das führt dazu, dass Sie keine eigene Funktion zur Speicherfreigabe mehr programmieren müssen – im fertigen Beispielprogramm werden Sie auch sehen, dass diese Funktion dort fehlt.

Damit ist die Node-Klasse vollständig. Im Wesentlichen handelt es sich also um ein der Node-Struktur sehr ähnliches Konstrukt, bei dem die Zugriffe auf die Daten per Funktion erfolgen, statt direkt.

Die Listen-Klasse mit dem Namen "List" ist schon aufwändiger, denn diese Klasse enthält die komplette Logik der verketteten Liste. Zunächst einmal hat die Listen-Klasse auch einen Konstruktor, der die Zeiger – die Member der Listen-Klasse sind – auf 0 setzt und einen Destruktor, der sich um die Speicherfreigabe kümmert.

Dabei hat die Liste zwei Zeiger als Member: Einen Node-Zeiger namens "root", der immer auf den Anfang der Liste zeigt und einen weiteren namens "current", der beim Iterieren über die Liste den aktuellen Zeiger enthält.

Dann gibt es noch die Methode *Add()*, mit der Sie einen neuen Knoten in die Liste einfügen. Dabei wird zunächst überprüft, ob es bereits einen "Current"-Zeiger gibt. Ist das nicht der Fall, handelt es sich bei der neu hinzugekommenen Node um die erste.

In diesem Fall wird der Root- und der Current-Zeiger gesetzt, ansonsten werden die Prev/und Next-Zeiger der aktuellen und der neuen Node passend gesetzt. Sie brauchen sich danach nicht mehr darum zu kümmern, dass die einzelnen Nodes in Ihrer Liste richtig verzweigt sind, denn das erledigt die *Add()*-Methode an einer zentralen Stelle besser.

Ferner gibt es noch die Methoden *GetNext()* und *GetPrev()*, die auf Basis des "Current"-Zeigers die jeweils nächste bzw. vorige Node liefern oder 0, und zwar dann, wenn es keine nächste oder vorherige Node gibt.

Das macht das Programmieren der bereits von vorherigen Beispielen bekannten Funktion *ReadFile()* deutlich einfacher (siehe dazu Listing 3).

Auch das Anzeigen der Datei wird deutlich einfacher: Alles was man tun muss, ist über die Listen-Instanz zu iterieren (siehe Listing 4).

So viel zu den Listen – bleiben noch die Kommandos fürs Menü. Hier bietet es sich an, eine "Command"-Basisklasse zu definieren, von der die tatsächlichen Kommandos dann abgeleitet werden. (Wenn eine Klasse von einer anderen erbt, dann sagt man dass die Klasse von der Basisklasse abgeleitet wurde.) Die Command- und die davon

DIE KLASSEN FÜRS MENÜ: KOMMANDOS

```
// klasse fuer das menu
class Command
{
public:
    Command( char c)
    {
        m_cKey = c;
    }

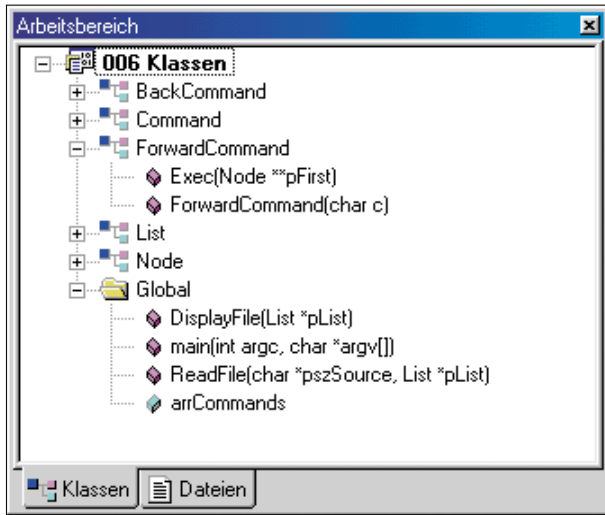
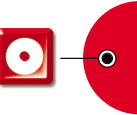
    virtual void Exec( Node** pFirst)
    {
        // default tut nichts
    }

    char Key()
    {
        return m_cKey;
    }

private:
    char m_cKey;
};

class ForwardCommand : public Command
{
public:
    ForwardCommand( char c) : Command( c)
    {
    }
    void Exec( Node** pFirst)
    {
        // scroll forward
        if ( (*pFirst)->GetNext() )
            (*pFirst) = (*pFirst)->GetNext();
    }
};

class BackCommand : public Command
{
public:
    BackCommand( char c) : Command( c)
    {
    }
    void Exec( Node** pFirst)
    {
        // scroll forward
        if ( (*pFirst)->GetPrev() )
            (*pFirst) = (*pFirst)->GetPrev();
    }
};
```



DIE KLASSENANSICHT IM ARBEITSBEREICH machte bei den bisherigen Projekten keinen Sinn. Nun finden Sie aber dort alle von Ihnen definierten Klassen in einer übersichtlichen Ansicht zusammengefasst.

LISTING 3

```
List l;
while( ! feof( pSource))
{
    char sz[1024];
    fgets( sz, 1024, pSource);
    Node* n = new Node();
    n->SetText( sz);
    l.Add( n);
}
```

LISTING 4

```
void Display( List* l)
{
    l->SetRootAsCurrent();
    while( Node* n = l->GetNext())
        printf( n->GetLine());
}
```

LISTING 5

```
// array aus kommandos
Command* arrCommands[] =
{
    new ForwardCommand( 'v'),
    new BackCommand( 'r')
};
int cntCommands =
    sizeof(arrCommands)/sizeof(Command*);
for( i=0; i<cntCommands; i++)
{
    if( arrCommands[i]->Key() == c)
        arrCommands[i]->Exec( &pFirst);
}
```

abgeleiteten Klassen finden Sie im Kasten: "Die Klassen fürs Menü: Kommandos."

Dazu gibt es aber noch einiges anzumerken. In der "Command"-Klasse finden Sie ein neues Schlüsselwort, und zwar "virtual". Damit markieren Sie eine Methode in einer Basisklasse als virtuell – Sie können dann diese Methode in einer abgeleiteten Klasse implementieren, und per Zeiger auf die Basisklasse aufrufen.

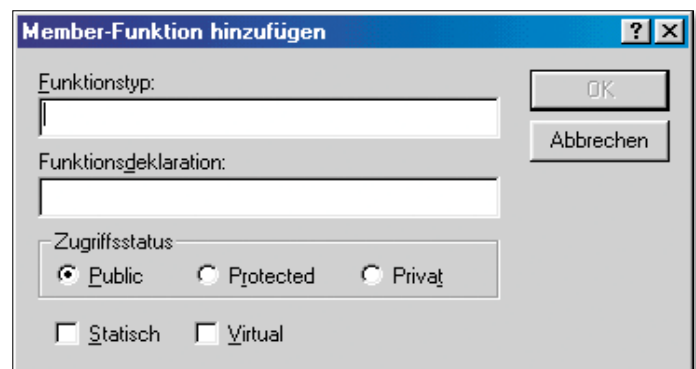
Genau so ist auch das Menü implementiert. Die Basisklasse hat die virtuelle Methode *Exec()*, die dann in den beiden abgelei-

und keine "reinen" Command-Objekte sind (siehe dazu Listing 5).

Die Schleife läuft dabei über die Anzahl der Elemente im Array. (Es ist zu beachten, dass die Ermittlung der Anzahl durch die Größe eines Command-Zeigers zu teilen ist, denn das Array enthält keine Kommandos, sondern Zeiger auf solche.)

Mit dem Zeiger auf das Kommando-Objekt kann dann ermittelt werden, ob das Objekt zur gedrückten Taste passt. Dazu wird die Methode *Key()* des Objekts aufgerufen, deren Implementierung sich in der Basisklasse befindet. Passt die Taste, wird die *Exec()*-Methode aufgerufen – nachdem diese in der Basisklasse aber als virtuell markiert ist, wird nicht die Implementierung der Basisklasse, sondern die der abgeleiteten Klassen aufgerufen: eine praktische Sache, die Zeiger auf Basisklassen.

Mit diesem Beitrag sind Sie nun bei der objektorientierten Programmierung mit C++ angelangt – und noch immer sind Zeiger eines der wichtigsten Elemente bei der Arbeit. Eines wird aber



ÜBER DIE KLASSENANSICHT im Arbeitsbereichs-Fenster können Sie auch Methoden zu Klassen hinzufügen, ohne die zugehörige Quellcode-Datei zu öffnen. Das ist zwar manchmal praktisch – aber meist macht man dies trotzdem besser selbst im Editor.

teten Klassen die tatsächliche Funktionalität enthält. Das Menü selbst wird genau wie in den vorherigen Beispielen einfach als Array implementiert, allerdings als Array aus Zeiger auf Command-Objekte. Tatsächlich befinden sich aber im Array Objekte, die von der Command-Klasse abgeleitet wurden

bei den zukünftigen Beispielen einfacher: Sie werden aufhören, die gesamte benötigte Funktionalität selbst zu programmieren, denn dazu sind Klassenbibliotheken da, und davon gibt es mehr als man meinen möchte. UR

