



Wie man kapselt, was man kapselt

Eine File-Klasse

Klassen dienen in C++ unter anderem der **Kapselung von Funktionalität** in logisch zusammengehörenden **Einheiten**.

THOMAS WÖLFER

Eine gegebene Klasse sollte die komplette Funktionalität enthalten, die für die Arbeit mit einer Instanz dieser Klasse (beziehungsweise mit dem zugrunde liegenden Konzept) notwendig ist. Dazu ist aber immer etwas Vorarbeit notwendig – man muss zunächst einmal ermitteln, was die benötigte Funktionalität eigentlich ist. In diesem Beitrag erstellen Sie eine Datei-Klasse, die das Konzept von Dateien in der Win32 API kapselt.

Mit Dateien kann man ziemlich viel machen, und dementsprechend umfangreich ist auch eine Kapselung der zu Dateien gehörenden Funktionalität. Allerdings gibt es bereits eine Menge anfertigen und vollständigen Datei-Klassen, so enthält die MFC zum Beispiel die CFile-Klasse sowie eine ganze Reihe davon abgeleiteter Klassen für die Arbeit mit Dateien. Aus diesem Grund wird im Zuge dieses Beispiels auch nicht alles implementiert – statt dessen erfahren Sie aber alles über die relevanten Aspekte und über Dinge, auf die Sie bei der Implementierung eigener Klassen achten müssen.

■ Die Datei, das unbekannte Wesen

Um eine Datei-Klasse zu implementieren, muss man exakt wissen, was eine Datei, von C++ aus gesehen, ist. Innerhalb der Win32 API ist eine Datei ein Datenstrom, aus dem gelesen und in den geschrieben werden kann. So wird etwa auch die serielle Schnittstelle von Win32 als Datei angesehen. Ob man das in der eigenen Datei-Klasse auch so haben will, ist die Frage.

Beim Benutzen einer Datei kann man nun verschiedene interessante Dinge mit ihnen tun: Man kann Daten aus einer Datei lesen und Daten hineinschreiben. Ferner kann man Dateien anlegen, öffnen, schließen und löschen. Man kann Dateien aber auch kopieren, oder nur einen Teil der Daten daraus lesen – das bedeutet, es muss auch eine Art Positionsbestimmung für den Inhalt von Dateien geben: Die Position innerhalb einer Datei misst man meist in Byte.

Man kann auch diverse Informationen über eine Datei ermitteln. Der Pfad zur Datei kann in verschiedene Einheiten unterteilt werden. Das Laufwerk, auf dem sich die Datei befindet, ist genauso eine solche Informationseinheit wie die Datei-Erweiterung, der Titel oder der vollständige Pfad. Man kann Dateien übrigens auch suchen und zwar auf der Festplatte, im LAN, im Internet – und eben auch sonst an allen Stellen, an denen Dateien abgespeichert sein können.

Alle die genannten Dinge haben die allermeisten Arten von Dateien gemeinsam. Dann gibt es aber auch Eigenschaften von Dateien, die nicht auf alle Dateien anwendbar sind. So werden HTML-Dateien beispielsweise mit Hilfe des http-Protokolls transportiert, während normale Dateien von der Festplatte mit der normalen Win32 API bewegt werden können. Ferner gibt es auch Dateien, die per FTP-Protokoll transportiert werden müssen. Eine ganz einfache Unterscheidung bei Dateien besteht auch im Inhalt der Datei: Handelt es sich um Text, ist dieser unter Umständen anders zu behandeln, als das bei binären Dateien der Fall ist: Textdateien können beispielsweise zeilenweise

gelesen werden, Binärdateien immer nur blockweise.

Betrachtungen dieser Art kann man noch eine ganze Reihe anstellen – was dabei aber herauskommt, ist immer das Eine: Es gibt ein gewisses Maß an Gemeinsamkeit bei Dateien, aber auch eine ganze Menge Unterschiede. Für die Implementierung einer Datei-Klasse bedeutet das im Wesentlichen, dass man eine grundlegende Basis-Klasse für Dateien zu implementieren hat, die die im Wesentlichen gleichen Eigenheiten von Dateien implementiert, aber für abgeleitete Klassen genug Spielraum lässt, um Eigenschaften oder Methoden aus der Basisklasse durch eigene Implementierungen zu ersetzen. Im Klartext bedeutet das: Man braucht eine Basis-Klasse mit virtuellen Methoden, die eine einigermaßen sinnvolle Implementierung haben.

Was dabei die "sinnvolle" Implementierung ist, ist Ansichtssache. Angesichts der Tatsache, dass die Datei-Klasse vermutlich meist auf einem Windows-System eingesetzt werden soll, ist es nahe liegend die Win32-API-Betrachtungsweise von Dateien für die Default-Implementierung zu verwenden: Man geht also zunächst einmal davon aus, dass Dateien im Dateisystem des lokalen Rechners behandelt werden und überlässt die Implementierung von HTML-Dateien oder FTP-Dateien einer abgeleiteten Klasse. Auch die Unterscheidung zwischen einer Text- und einer Binärdatei steckt man in das Ableitungssystem: Eine Textdatei kann auch binär gelesen werden, also ist die binäre Implementierung der Teil, der in die Basisklasse kommt.

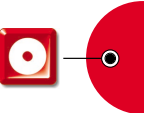
■ Eine kleine Datei-Klasse

Aus den bisher eingesammelten Informationen über eine Datei kann man bereits eine Klasse zusammenbauen. Zunächst einmal braucht man ganz sicher einen Konstruktor:

```
class File
{
    File();
```

Dabei fällt aber auf, dass man ein Objekt vom Typ File vielleicht auch anhand eines File-Handles einer Datei erzeugen will, die mit den C-Funktionen zum Öffnen von Dateien bereits geöffnet wurde. Diese Funktionen liefern als Handle einen "int" – man braucht also auch einen Konstruktor, der einen Integer-Parameter erhält:

```
File( int hFile);
```



Schließlich wird man File-Objekte aber ganz sicher auch mit dem Namen und dem Pfad einer Datei erzeugen wollen. In diesem Fall würde der Konstruktor die Datei gleich öffnen. Dazu muss er aber auch die Flags übergeben bekommen, die die Art und Weise des Öffnens spezifizieren: Soll die Datei zum Lesen oder zum Schreiben geöffnet werden – und soll die Datei eventuell zuvor gelöscht oder angelegt werden? Dieser Konstruktor hätte dann folgendes Aussehen:

```
File( const char* pszPath,
    UINT flags);
```

Die Implementierung dieses Konstruktors wirft ein weiteres Bild auf die Probleme beim Kapseln von Funktionalität in Klassen, darum wird dessen Implementierung gleich abgebildet. Doch zuvor braucht man eine weitere Funktion.

Nachdem ein Datei-Objekt auch mit einem Konstruktor ohne Parameter erzeugt werden kann, ist es offensichtlich notwendig, dass man auch eine *Open()*-Funktion implementiert, mit der dann das Datei-Objekt mit einer tatsächlich auf der Festplatte vorhandenen Datei verbunden wird. Dabei ist es ebenfalls notwendig zu spezifizieren, wie die Datei geöffnet werden soll. Die *Open*-Methode sieht also folgendermaßen aus:

```
bool Open( const char*
    pszPath, UINT flags);
```

Die Methode muss offensichtlich einen bool'schen Wert zurückliefern, denn beim Öffnen der Datei könnte etwas schief gehen: etwa dann, wenn die Datei nicht vorhanden ist oder dann, wenn die Datei zum Schreiben geöffnet werden soll, aber bereits von einem anderen Prozess zum Schreiben geöffnet wurde. *Open()* ist ohnehin etwas kompliziert – darum finden Sie eine Beispiel-Implementierung dazu im Kasten: "Die Open Methode". Diese *Open()*-Methode kann nun im zuvor angesprochenen Konstruktor verwendet werden, und dabei tut sich ein weiteres Problem auf (siehe Listing 1).

Das Problem ist sofort ersichtlich: Ein Konstruktor liefert als Returnwert entweder einen Zeiger auf ein Objekt seines Typs oder null – aber auf gar keinen Fall einen bool'schen Wert. Bool'scher Wert ist aber genau das, was geliefert werden müsste, denn sonst könnte man nicht überprüfen, ob das Öffnen der Datei in-

LISTING 1

```
File( const char* pszPath, UINT flags)
{
    bool fWorked = Open( pszPath, flags);
}
```

nerhalb des Konstruktors funktioniert hat.

Dieses Dilemma löst ein C++-Feature, das an keiner anderen Stelle im Sonderheft erwähnt wurde: Exceptions. Eine Exception (Ausnahme) kann an beliebigen Stellen im Programm aus-

auch "Eine Exception werfen". Diese muss dann im Quellcode entsprechend behandelt – man sagt dazu "gefangen" werden. Tut man das nicht selbst, wird das Programm beendet. Der einfachste Fall eine Exception zu fangen, besteht darin, einfach mehrere zu fangen (siehe Listing 2).

■ Lesen und schreiben

Aus einer Datei muss natürlich gelesen werden und in die Datei muss geschrieben werden können. Dazu bieten sich Funktionen mit den Namen *Read* und *Write* an (siehe Listing 3).

LISTING 2

```
File* pFile = 0;
try
{
    pFile = new File("C:\\autoexec.bat", mode::read);
}
catch( ...)
{
    // eine exception ist aufgetreten und muss behandelt werden
}
```

gelöst werden. Dabei werden Exceptions meist auch in Klassen gekapselt, so dass es "FileExceptions", MemoryException" und dergleichen mehr gibt. Man betrachte dazu das folgende Statement:

```
File aFile( "c:\\autoexec.bat",
    mode::open);
```

Hat dabei alles geklappt, ist alles wunderbar – man kann das Objekt mit dem Name *aFile* dann für Dateioperationen verwenden. Hat aber etwas nicht funktioniert, ist das Objekt in sich ungültig und darf nicht weiter verwendet werden. Erkennen tut man das daran, dass im Konstruktor eine Exception ausgelöst wurde. Man nenn das

Nun ist es so, dass man bei der Implementierung solcher Funktionen natürlich auf die Win32 API zurückgreift – und die hat dummerweise auch einige Fehler. So einer kann zum Beispiel dann auftreten, wenn man versucht *WriteFile()* mit einem 0 Byte großen Puffer aufzurufen. Nun kann man natürlich

LISTING 3

```
UINT Read( void* pZiel, UINT cb);
void Write( const void* pSource, UINT cb);
```

auf dem Standpunkt stehen, so etwas sollte sowieso nicht vorkommen – und die Eingangsparameter der Funktion entsprechend mit *ASSERT()* testen, oder aber man lässt einen Aufruf mit dem Parameter zu und umgeht den Fehler in der

LISTING 4

```
void CFile::Write(const void* lpBuf, UINT nCount)
{
    ASSERT_VALID(this);
    ASSERT(m_hFile != (UINT)hFileNull);

    if (nCount == 0)
        return; // avoid Win32 "null-write" option
    ...
}
```



API auf andere Weise. Im Fall der File-Klasse bietet sich letzteres an (siehe Listing 4 auf der vorherigen Seite).

Genauso können nun auch die anderen Methoden der Klasse definiert werden. Das ist im Wesentlichen Tipparbeit und daher nicht weiter erwähnenswert. Allerdings gibt es noch zwei spezielle Methoden, die in einer File-Klasse enthalten sein sollten, und daher an dieser Stelle noch extra angesprochen werden sollen.

Die erste ist eine Funktion aus der Gruppe *Löschen/Umbenennen/Suchen von Dateien*. Als Beispiel wird hier die Löschen-Methode aufgeführt. Wenn man eine Datei löschen möchte, ist ein

LISTING 5

```
class File
{
public:
    static bool Remove( const char* pszPath);
};
```

eigenständiges File-Objekt meist gar nicht erwünscht. Schließlich macht es wenig Sinn zunächst ein File-Objekt zu erzeugen, nur um dann gleich die darunterliegende Datei im Dateisystem zu entfernen. Überhaupt gibt es oft Methoden, die zwar logisch prima in eine bestimmte Klasse passen, aber mit den

in der Klasse gekapselten Daten nichts weiter anzufangen wissen. Solche Methoden macht man innerhalb der Klasse "static". Die Methoden sind dann zwar Teil der Klasse – aber man braucht keine konkrete Instanz der Klasse, um die Methode zu verwenden. Angenommen die Klasse hätte das folgende Aus-

DIE OPEN-METHODE

Solange man sich das Klassen-Interface ausdenkt, ist die Welt noch gut, schön und übersichtlich. Eine *Open()*-Methode öffnet einfach eine Datei und liefert true, wenn das geht. Das klingt nicht sonderlich aufwändig – doch das täuscht. Was alles zu beachten ist, können Sie an der hier vorliegenden Beispiel-Implementierung sehen, die auf der tatsächlichen Implementierung von CFile in MFC basiert:

Listing 6

```
bool File::Open( const char* pszPath, UINT nFlags)
{
```

Zunächst einmal sollte man für die Debug-Version der Klasse passende Tests einbauen: Ist das Objekt in sich noch in Ordnung, und passen die eingehenden Parameter? Das sind Dinge, die sichergestellt sein sollten, bevor man überhaupt irgendetwas anderes tut. Die Parameter sind dann sinnvoll, wenn es sich bei pszPath nicht um einen Null-Pointer, sondern um einen gültigen String handelt, und wenn nFlags gültige Muster haben. Für die File-Basisklasse darf zum Beispiel der "textMode"-Flag nicht gesetzt sein. Die dazugehörigen Statements sehen dann folgendermaßen aus:

Listing 7

```
ASSERT_VALID(this);
ASSERT(AfxIsValidString(lpszFileName));
ASSERT((nOpenFlags & typeText) == 0);
```

Dann muss überprüft werden, ob die Modi fürs Lesen und Schreiben sowie für Share richtig gesetzt sind, und schließlich muss dazu der passende Win32 Modus gefunden werden:

Listing 8

```
ASSERT((modeRead|modeWrite|modeReadWrite) == 3);
DWORD dwAccess = 0;
switch (nOpenFlags & 3)
{
case modeRead:
    dwAccess = GENERIC_READ;
    break;
case modeWrite:
    dwAccess = GENERIC_WRITE;
    break;
case modeReadWrite:
    dwAccess = GENERIC_READ|GENERIC_WRITE;
    break;
default:
```

```
...ASSERT(FALSE); // ungültiger share mode
..}

..// map share mode
DWORD dwShareMode = 0;
switch (nOpenFlags & 0x70) // jetzt exclusive
{
default:
    ...ASSERT(FALSE); // ungültiger share mode?
case shareCompat:
case shareExclusive:
    dwShareMode = 0;
    break;
case shareDenyWrite:
    dwShareMode = FILE_SHARE_READ;
    break;
case shareDenyRead:
    dwShareMode = FILE_SHARE_WRITE;
    break;
case shareDenyNone:
    dwShareMode = FILE_SHARE_WRITE|FILE_SHARE_READ;
    break;
..}
```

Dann müssen die Erzeugungs-Flags aus der File-Klasse auch die passenden Win32 Flags gemappt werden. Das sieht mehr oder minder genauso aus wie der oben stehende Code und ist darum nicht abgebildet.

Erst dann kommt man zu der Stelle, an der die Datei tatsächlich mit der entsprechenden Win32 API geöffnet wird:

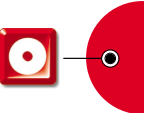
Listing 9

```
HANDLE hFile = ::CreateFile(lpszFileName, dwAccess,
dwShareMode, &sa,
dwCreateFlag, FILE_ATTRIBUTE_NORMAL, NULL);
```

Schließlich muss noch überprüft werden, ob das Öffnen auch funktioniert hat – hat es das nicht, muss entsprechend reagiert werden. Nur im anderen Fall, also wenn die Datei im gewünschten Modus geöffnet werden konnte, wird TRUE geliefert.

Listing 10

```
if (hFile == INVALID_HANDLE_VALUE)
{
    // entsprechend reagieren
}
```



sehen (siehe Listing 5 auf der vorherigen Seite).

In diesem Fall könnten Sie die `Remove()` Methode im Quelltext auf die folgende Art verwenden

```
void foo()
{
    File::Remove(
        "c:\\autoexec.bat");
};
```

■ Operatoren

Wie schon häufiger erwähnt hat C++ einen recht großen Sprachumfang – zumindest im Vergleich mit C. Das macht sich auch in diesem Sonderheft bemerkbar, denn längst nicht alle Features der Sprache finden Erwähnung. (Für tiefergehende Informationen können Sie aber das Referenzwerk auf der CD zum Sonderheft verwenden.) Doch ein Feature soll noch erwähnt werden, weil es in die File-Klasse so gut hineinpasst: Das Überladen von Operatoren. In C++ können Sie Operatoren selbst definieren: Dabei erhalten die ganz normalen Operatoren für bestimmte Klassen eine besondere Bedeutung – auf diese Weise kann man zum Beispiel einen Operator `+` definieren, mit dem Strings verkettet werden können.

Damit ist es auch möglich, eine eigene Typumwandlung in Klassen einzubauen. Das ist sehr praktisch, wie sie gleich sehen werden. In der Win32 API werden Files immer als `HANDLE` auf Files betrachtet – dazu gibt es den Datentyp `HFILE`. Nun wäre es sehr schön, wenn die eigene Klasse auch immer direkt als Parameter für die dateibezogenen Funktionen der Win32 API verwendet werden könnte. Von Haus aus geht das natürlich nicht, denn diese Funktionen erwarten einen `HFILE`, aber Objekte vom Typ `File` sind eben genau nur vom Typ `file`.

Dafür kann man aber einen Operator programmieren, der sich dann um die Umwandlung kümmert, wenn Sie notwendig ist:

```
operator HFILE() const;
```

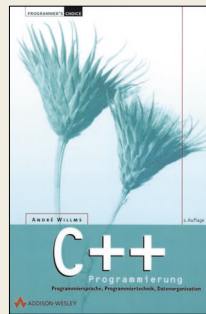
So viel zu File-Klassen und den allgemeinen Überlegungen, denen Sie folgen müssen, wenn Sie eigene Klassen entwerfen. Egal, ob Sie diese Hinweise beachten oder nicht, auf eines sollten Sie auf jeden Fall Wert legen: Gestalten Sie Ihre Implementierungen so robust wie möglich – und das bedeutet konkret: Testen Sie grundsätzlich und immer alle Eingangsparameter, dann haben Sie später weniger Ärger. UR

Bücher

C++ PROGRAMMIERUNG

Programmiersprache, Programmieretechnik, Datenorganisation

Das Buch wendet sich an Leser, die noch wenig C++-Programmiererfahrung haben.



Zum Einstieg wiederholt es knapp aber vollständig die Basics der Programmierung von Ablaufplan über Schleife bis zur Struktur. Dann geht es an die Objektorientierung, die Klassen, Stacks und Queues, Klassen-Scha-

blonen und was man so an Grundlagen als Objektorientierter Programmierung benötigt. Danach werden in C++ File-Handling, Listen, Vererbung, Rekursion, Such- und Sortierverfahren, Bäume, und Exceptions geboten. Es gibt viele Übungen und Kontrollfragen. Das Buch zeigt anhand der Programmiersprache C++, wie man die Grundaufgaben der professionellen Programmierung löst und liefert dazu die tieferen Einblicke in die Funktionalitäten von C++.

André Willms: C++-Programmierung - Programmiersprache, Programmieretechnik, Datenorganisation, 2. Auflage. Addison-Wesley, 34,95 Euro, 418 Seiten, mit CD, ISBN 3-8273-1627-8

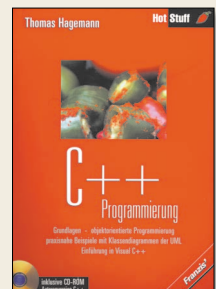
C++-PROGRAMMIERUNG

Grundlagen, objektorientierte Programmierung

Dieser Band fängt richtig von vorne an mit der Geschichte von C und C++. Er behandelt die Datentypen, Variablen, Kontrollstrukturen, Felder, Operatoren, Funktionen Schritt für Schritt bis zu ersten Programmen in C++. Die Besonderheiten der C-Compiler-Systeme und die objektorientierte Programmierung werden gezeigt, Streams und weitere Sprachelemente. In Beispielanwendungen werden die erworbenen Kenntnisse dann umgesetzt. Das alles mit der Unterstützung von Visual C++,

dessen Autoren-edition auf der CD zu finden ist. Ein schönes Einsteigerbuch mit ausführlichem Glossar und Stichwortverzeichnis im Anhang.

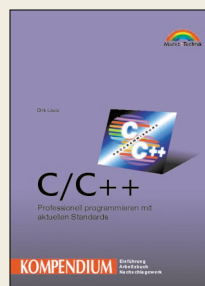
Thomas Hagemann: C++ Programmierung, Franzis' Verlag, 21,95 Euro, 384 Seiten, mit CD, ISBN 3-7723-5464-5



C/C++ KOMPENDIUM

Einführung, Arbeitsbuch, Nachschlagewerk

Auf der CD zu diesem Heft finden Sie ein Probekapitel aus diesem Buch. Der Band erfüllt die Ankündigungen des Untertitels recht gut: Man kann ohne Vorkenntnisse loslegen und wird erst mal über den ANSI-Standard aufgeklärt, der den "legalen" Sprachumfang von C und C++ beschreibt. Dann werden die einzelnen Sprachelemente näher beleuchtet. Präprozessor und Ex-



ceptions werden behandelt, Funktionen und die Programmentwicklung. Danach geht es in die Objektorientierung, von Vererbung bis Polymorphie werden die Grundlagen aufgezeigt. Es wird gezeigt, wie man Laufzeitbibliotheken aufbaut und es werden weiterreichende Informatik- und Programmierkenntnisse angeboten. Das Buch bietet viel Gelegenheit, im Programmieren fit zu bleiben. Im umfangreichen Referenzteil kann man bei Bedarf die exakte Definition der C++-Sprachelemente nachschlagen.

Dirk Luis: C/C++ Kompendium - Einführung, Arbeitsbuch, Nachschlagewerk, Markt+Technik, 49,95 Euro, 1100 Seiten, mit CD, ISBN 3-8272-6335-2