



## Windows Registry durchkämmen

Die Windows **Registry** steht in dem Ruf, ein harter Brocken zu sein. Mit ein wenig Aufwand lässt sich jedoch ein simpleres **Interface in C++** realisieren, mit dem der Umgang mit der Registry viel einfacher von der Hand geht.



# Simplicissimus

THOMAS WÖLFER

Die Windows Registry ist einer der wichtigsten Bestandteile von Windows 9.x, NT und Windows 2000. Leider auch einer der am schwierigsten zu handhabenden, auch programmtechnisch. Der Grund dafür ist einfach: Die Win32 Registry API ist nicht gerade aufgeräumt, was im Wesentlichen darauf zurückzuführen ist, dass sich die Baumstruktur der Registry nicht besonders für eine Abbildung in Form von C-Funktionen eignet. Das Interface zur kompletten Win32 API ist

### QUICK INDEX

- **Einfach nur im Prinzip**  
Die Baumstruktur der Registry ist in C nicht einfach nachzuempfinden, in C++ aber schon.
- **Der Konstruktor des Xregistry**  
Der Konstruktor berücksichtigt die verschiedenen Windows-Varianten.
- **GetValue() ist überladen**  
GetValue muß sich natürlich auf die verschiedenen Situationen einstellen.
- **EnumValues**  
Wird an vielen Stellen benötigt.
- **Der Browser: ein MFC-Programm**  
Ganz normal dialogbasiert, durchwandert den Baum rekursiv.

jedoch eine C- und keine C++-Schnittstelle - dementsprechend geduldig muss der Programmierer sein, wenn er versucht, die Registry API zu verwenden.

Das muss aber nicht so sein. Mit ein wenig Aufwand lässt sich ein wesentlich einfacheres Interface zur Registry in C++ realisieren, und genau solch ein Interface stellt der folgende Beitrag zur Verfügung. Die fertige Klasse ermöglicht sehr einfache Verwendungen der Registry. Um einen Registry-Eintrag vom Typ DWORD auszulesen, genügen folgende Zeilen:

```
DWORD dwValue;  
XRegistryKey key(  
HKEY_CURRENT_USER, "Software\\  
Hersteller\\Programme\\Setting");  
key.GetValue( dwValue);
```

Genauso einfach ist es, mit der vorgestellten Klasse einen Registry-Eintrag zu schreiben oder über einen Teil der Registry-Äste zu iterieren. Dabei wird Sorge dafür getragen, dass die unter Windows 9.x, NT und Windows 2000 teilweise unterschiedlichen Registry-API-Aufrufe automatisch richtig verwendet werden. Die vorgestellte Klasse kümmert sich um alle diese Belange.

### ■ Einfach nur im Prinzip

Im Prinzip ist die Registry relativ einfach aufgebaut. Letztendlich gibt es nur einen Baum, der ausschließlich aus Items

besteht. Jedes Item stellt sowohl einen Ast als auch einen Container für Einträge dar. In der Registry API wird ein atomarer Teil der Registry "Key" genannt. Dieser kann weitere Keys enthalten. Dabei ist ein Key entweder ein Ast innerhalb der Registry oder aber ein Blatt, das ein Name/Value-Paar darstellt.

Um diese Struktur mit C++ abzubilden, bietet sich eine Klasse vom Typ "RegistryKey" an. Mit dieser Klasse können einzelne Einträge ausgelesen und gesetzt werden; außerdem kann über alle darin befindlichen Einträge (seien es nun Äste oder Name/Value-Paare) iteriert werden. Damit die Abbildung komplett wird, macht es Sinn, darüber hinaus eine Klasse "XRegistry" zu implementieren. Diese dient als Container für alle Elemente vom Typ "RegistryKey".

Zusätzlich definiert das Beispiel einige Hilfsklassen, wie eine XRegValueInfo-Klasse, die den Typ eines Schlüssels ermittelt. (In der Registry API ist es vorgesehen, dass Schlüssel von einem bestimmten Typ sein können. So existieren Schlüssel für Binärdaten, solche für Strings und solche für DWORDs.)

Im folgenden werden die wichtigsten Elemente der beteiligten Klassen vorgestellt. Darüber hinaus findet sich ein kleiner Registry-Browser, der unter Verwendung der vorgestellten Klassen mit wenigen Zeilen Code eine Schnellüber-



sicht über einen Teil der Registry bietet. Sowohl den Quellcode zum Browser als auch die Registry-Klassen und den Browser als ausführbares Programm finden Sie im Internet. Die wesentlichen Elemente der XRegistryKey-Klasse sind wie nachfolgend definiert (Listing 1).

Im Wesentlichen besteht ein XRegistryKey aus einem Konstruktor und einem Destruktor sowie aus Methoden zum Enumerieren, Auslesen, Löschen und Setzen von Werten. Darüber hinaus gibt es einige wenige Hilfsfunktionen. Hier die Details der Implementierung.

## ■ Der Konstruktor des Xregistry

Key erledigt einen Großteil der Arbeit. Insbesondere kümmert er sich um die unterschiedliche Behandlung der Registry in Windows 9.x und Windows NT bzw. Windows 2000: Unter NT und 2000 müssen bei der Arbeit mit der Registry Zugriffsrechte berücksichtigt werden. Unter Windows 9.x ist dies nicht der Fall. (Im abgedruckten Quellcode sind aus Gründen der Übersichtlichkeit Fehlerbehandlungen und Debug-Codesequenzen nicht enthalten (Listing 2). Zunächst legt der Konstruktor von XRegistryKey die gewünschten Zugriffsmodi fest. Diese müssen sich unter NT und Windows 9.x unterscheiden, denn der Kompletzzugriff ist unter NT ohne bestimmte Rechte des angemeldeten Users nicht möglich. Unter Windows 9.x hingegen kann immer KEY\_ALL\_ACCESS, der Kompletzzugriff, angefordert werden. Für diese Unterscheidung werden zunächst zwei Variablen definiert, woraufhin der XRegistryKey-Konstruktor die Hilfsfunktion CheckIfNT() verwendet (siehe dazu den Kasten: NT oder nicht?), um die Zugriffsflags, die tatsächlich verwendet werden sollen, zu bestimmen. Unter NT sind dies andere als unter Windows 9.x.

Danach wird mit der RegOpenKeyEx() API der Registry-Schlüssel tatsächlich geöffnet und der zurückgelieferte Fehlercode in der Member-Variable m\_idErr gespeichert. Das ist wichtig, damit im Destruktor überprüft werden kann, ob der RegistryKey wieder geschlossen werden muss oder nicht. Konnte der Schlüssel nicht geöffnet werden, versucht der Konstruktor anschließend, den Schlüssel anzulegen. Diese Funktionalität wäre zwar in einer separaten Funktion besser aufgehoben, aber die vorliegende Implementierung macht es später bei der

Benutzung einfacher, neue Werte in der Registry einzutragen. Soviel zum Konstruktor der Klasse. Der Destruktor ist deutlich einfacher.

Dort wird einfach die oben erwähnte Member-Variable m\_idErr überprüft: Enthält sie den Wert ERROR\_SUCCESS (diese bedeutet: kein Fehler), so wurde der Key im Konstruktor erfolgreich geöffnet und muss nun geschlossen werden:

```
if( m_idErr == ERROR_SUCCESS)
    RegCloseKey( m_key);
```

## ■ GetValue() ist überladen

Die wichtigste Member-Funktion ist GetValue(). Von dieser Methode existie-

ren verschiedene Varianten, jeweils eine für die unterschiedlichen in der Registry ablegbaren Datentypen. Strings, DWORDS und andere Daten können in der Registrierungsdatenbank abgelegt werden - und alle diese Datentypen sollen auch erfragbar sein. Nachdem die XRegistry-Klasse mehrere unterschiedliche Methoden zur Verfügung stellt, können bei der Benutzung der Klasse später folgende Ausdrücke verwendet werden:

```
CString str;
Reg.GetValue("PfadZumGewünschten
Schlüssel", str);
```

oder

```
DWORD dw;
Reg.GetValue("PfadZumGewünschten
Schlüssel", &dw);
```

### LISTING 1

```
class XRegistryKey
{
    XRegistryKey( HKEY tree, const char* pszSubkey);
    ~XRegistryKey();
    LONG Error() const; // Get
    error code
    BOOL EnumValues( XRegValueInfo
    List& rList); // get all named
    values >from this key
    BOOL DeleteValue( const char* pszName); // delete a named
    value

    BOOL GetValue( const char*
    pszName, DWORD* pData); // query a named REG_DWORD value
    // hier folgen weitere Methoden zum Auslesen von Daten

    BOOL SetValue( const char*
    pszName, DWORD dwData); // set a named REG_DWORD value
    // hier folgen weitere Methoden zum Setzen von Daten

private:
    BOOL Open();
    BOOL Close();
    LONG m_idErr;
    HKEY m_key;
};
```

### LISTING 2

```
XRegistryKey::XRegistryKey( HKEY tree, const char* pszSubKey)
{
    REGSAM samDesiredNT = KEY_QUERY_VALUE |
    KEY_ENUM
    RATE_SUB_KEYS |
    KEY_EXECUTE |
    KEY_SET_VAL
    LUE;
    REGSAM samDesiredW95
    = KEY_ALL_ACCESS;

    REGSAM samDesired;
    if( XRegistry::CheckIfNT()) samDesired = samDesiredNT;
    else samDesired = samDesi
    redW95;

    m_idErr = RegOpenKeyEx( tree, pszSubKey, 0, samDesired, &m_key);

    if( m_idErr != ERROR_SUCCESS)
    {
        XRegistry reg;
        XRegistryKey* pKey = reg.CreateKey( tree, pszSubKey, "");
        if( ! pKey) m_idErr = reg.Error();
        else
        {
            delete pKey;
            m_idErr = RegOpenKeyEx( tree, pszSubKey, 0, samDesired, &m_key);
        }
    }
}
```



Je nachdem, was für ein Datum ausgelesen werden soll, muss eine der verschiedenen GetValue()-Spielarten verwendet werden.

Hier eine der GetValue()-Methoden - in diesem Fall die für DWORDs - als Beispiel (Listing 3).

Da der Zugriff auf die Registry notorisch fehleranfällig ist, ist die GetValue()-Funktion mit Sicherungscode gespickt. Zum Teil ist dieser nur in der Release-Variante eingebaut (und zwar dann, wenn \_DEBUG nicht definiert ist), ausführliche Meldungen erscheinen zusätzlich in der Debug-Variante.

Zunächst wird mit RegQueryValueEx() überprüft, ob der Schlüssel, der ausgelesen werden soll, ein Datum von dem Typ enthält, der erfragt wurde. Im Beispiel ist das ein DWORD, die anderen GetValue()-Implementierungen überprüfen entsprechend die anderen Datentypen. Entspricht der Typ des Schlüssels nicht dem erfragten Typ (dwType != REG\_DWORD), wird in der Debug-Version der Klasse eine Fehlermeldung in das Output-Fenster des Debuggers ausgeworfen. Anschließend, und zwar sowohl im Release, als auch im Debug-Build, wird die Methode bei einem unpassenden Typ beendet und FALSE zurückgeliefert.

War der Datentyp hingegen korrekt, so wird erneut RegQueryValueEx() bemüht. Diesmal nicht, um den Datentyp herauszufinden, sondern um die Daten tatsächlich auszulesen. Der Erfolg oder Misserfolg der Leseoperation wird per Return-Code zurückgeliefert.

Soviel zum einfachen Lesen von Werten aus der Registry.

Ein interessanter Aspekt ist auch das Aufzählen von Elementen innerhalb eines Schlüssels. Dazu stellt die XRegistryKey-Klasse die Methode EnumValues() zur Verfügung.

### EnumValues

Diese Methode erhält als Parameter eine Referenz auf eine Liste vom Typ XregValueInfoList. Diese Liste ist im Headerfile XRegistry.h definiert und dient ausschließlich dem Transport von Registry-Einträgen in Form einer Liste.

Benötigt wird eine solche Enum()-Funktion an vielen Stellen. Das einfachste Beispiel ist ein Registry Browser, mit dem Äste der Registry angezeigt werden können. Um herauszufinden, wel-

che Elemente sich innerhalb eines Astes befinden, bleibt dem Programmierer nichts anderes übrig, als diese Schritt für Schritt aufzuzählen.

Innerhalb von EnumValues() verwendet die XRegistryKey-Klasse die

Win32 API RegEnumValues(), die genau diesen Zweck erfüllt.

RegEnumValues() kann so lange aufgerufen werden, bis die Funktion ERROR\_NO\_MORE\_ITEMS liefert. Ist dies der Fall, stehen keine weiteren Ein-

### LISTING 3

```

BOOL
XRegistryKey::GetValue( const char* pszName, DWORD* pData)
{
    DWORD dwType;
    DWORD cbData;

    m_idErr = RegQueryValueEx( Key(),
                               pszName,
                               NULL,
                               &dwType,
                               NULL,
                               &cbData);

    #ifdef _DEBUG
    if( dwType != REG_DWORD)
        TRACE("WARNING (XRegistry (5)): Attempt to read non REG_DWORD into DWORD\n");
    #endif

    if( dwType != REG_DWORD) return FALSE;

    if( m_idErr == ERROR_SUCCESS)
    {
        m_idErr = RegQueryValueEx( Key(),
                                   pszName,
                                   NULL,
                                   &dwType,
                                   (LPBYTE)pData,
                                   &cbData);
    }

    return m_idErr == ERROR_SUCCESS;
}

```

### NT ODER NICHT?

Beim Programmieren für Win32 ist eines immer wieder lästig: Zwar sehen die allermeisten APIs für alle Windows-Plattformen (Windows 9.x, NT und 2000) gleich aus, doch es gibt kleine Unterschiede im Verhalten. Nicht zuletzt ist dafür das Sicherheitssystem von NT und 2000 verantwortlich, denn dieses ist bei den "kleinen" 32-Bit-Windows-Varianten nicht verfügbar. Dementsprechend gestalten sich viele Win32 APIs intern unterschiedlich: Unter NT und 2000 muss der Sicherheitskontext gewahrt werden, bei den kleineren Systemen hingegen hat jeder Prozess immer Zugriff auf alles.

Aus diesem Grund ist ein Versionstest in nahezu jedem größeren Win32-Programm notwendig - sei es, um die für einen API-Aufruf benötigten Parameter "richtig" zusammenzubauen, oder um schlicht festzustellen, ob eine bestimmte API tatsächlich auf der Plattform verfügbar ist, auf der das Programm läuft. Bei der XRegistry-Klasse ist dies nicht anders, denn zum Öffnen von Registry-Schlüsseln sind unter NT und 2000 bestimmte Rechte erforderlich, die bei 9.x nicht existieren. Aus diesem Grund ent-

hält die XRegistry-Klasse eine Testmethode, die die aktuelle Plattform ermittelt: CheckIfNT().

```

OSVERSIONINFO os;
os.dwOSVersionInfoSize = sizeof(
OSVERSIONINFO);
GetVersionEx( &os);
if( os.dwPlatformId == VER_PLATFORM_WIN32_NT) return TRUE;
else if( os.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) return FALSE;
return FALSE;

```

CheckIfNT() verwendet dazu die OSVERSIONINFO-Struktur aus der Win32 API sowie die zugehörige Funktion GetVersionEx(). Handelt es sich bei der aktuellen Plattform um NT oder Windows 2000, liefert GetVersionEx() unter der dwPlatformId das Symbol VER\_PLATFORM\_WIN32\_NT, ansonsten liefert die Funktion VER\_PLATFORM\_WIN32\_WINDOWS. Ein besserer Test ist in der tatsächlichen Implementierung zu finden, denn unter Umständen ist die aktuelle Plattform etwas anderes, zum Beispiel Windows CE oder Win32s. Bei einer "richtigen" Anwendung müsste auch auf diese Betriebssystemversionen reagiert werden.



## SOURCEN...

...im Internet unter [www.pc-magazin.de/download/special\\_2/registry/](http://www.pc-magazin.de/download/special_2/registry/)

träge im abgefragten Registry-Schlüssel zur Verfügung. Für jeden gelieferten Wert alloziert EnumValues() eine neue Instanz eines XRegValueInfo-Objekts, füllt dieses mit den soeben gelieferten Werten und trägt das neue Objekt dann in der Liste ein, die die Funktion als Parameter erhält. Auf diese Weise kann die aufrufende Funktion einfach eine Liste aller Elemente innerhalb eines Registry-Schlüssels anfordern. Die aufrufende Funktion ist dafür zuständig, dass die einzelnen RegValueInfo-Objekte wieder zerstört und freigegeben werden.

Soviel zu den Details der XRegistry-Implementierung und den zugehörigen Klassen. Wer mehr wissen will: use the source!

## Der Browser: ein MFC-Programm

Abschließend noch ein paar Worte zum Beispielprogramm, dem Registry Browser. Bei diesem handelt es sich um ein ganz normales dialogboxbasierendes MFC-Programm. Sämtliche Teile des Programms wurden mit dem VC++ Wizard automatisch erstellt. Anschließend waren lediglich drei Änderungen notwendig: Zum einen wurde der Header der XRegistry-Klasse im Projekt-Headerfile inkludiert und die Implementierungsdatei der Klasse in das Projekt aufgenommen. Zum anderen wurde die OnInitDialog()-Methode des Dialogs um einen einzigen Funktionsaufruf erweitert: InsertTree(). Bei dieser Funktion handelt es sich um die dritte Änderung des Standardprojekts.

InsertTree() ist eine rekursive Funktion, die die XRegistry-Klasse verwendet. Der initiale Aufruf der Funktion liefert den Startwert, bei dem die Rekursion über den Registry-Baum begonnen werden soll. Für jeden neu gefundenen Eintrag wird InsertTree() erneut aufgerufen.

Die Funktion selbst verwendet eine weitere Hilfsklasse, XregKeyInfoList. Diese entspricht weitgehend XregValueInfoList, nur dient sie der Aufnahme von Schlüsseln statt der Aufnahme von Werten. Iteriert wird mit XRegistryKey::EnumKeys(). Für jeden gefundenen Eintrag wird das Tree-Control auf dem Dialog um einen neuen Eintrag er-

weitert (Listing 4). Wie man sieht, ist die Iteration über komplette Registry-Äste mit der XRegistryKey-Klasse und ihren Hilfsklassen ein Kinderspiel. Die komplizierte Struktur der Win32 API ist voll-

ständig verborgen. Der Programmierer braucht sich nur noch um das tatsächliche Problem zu kümmern, statt die Win32-Dokumentation zu durchforsten.

UR

### LISTING 4

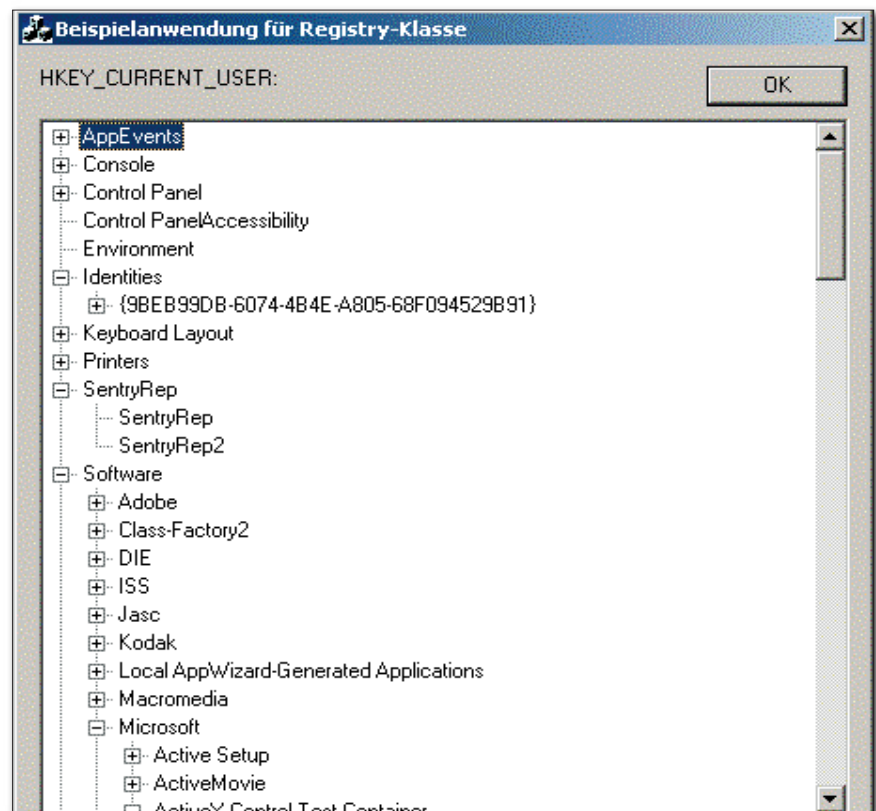
```
void CRegistryDlg::InsertTree( const char* pszPath, HTREEITEM hParent)
{
    XRegKeyInfoList rLst;
    XRegistryKey keyStart( HKEY_CURRENT_USER, pszPath);
    if( keyStart.Error() ==
    ERROR_SUCCESS)
    {
        if( m_reg.EnumKeys( keyStart, rLst))
        {
            while( ! rLst.IsEmpty())
            {
                XRegKeyInfo* pInfo = rLst.Remo
                CString strPath;

                if( ! pszPath) strPath = pInfo
                >m_strName;

                else strPath = pszPath + CString("\\") +
                CString strInsert = pInfo->m_str
                Name;

                HTREEITEM hInsertAfter = TVI_LAST;
                HTREEITEM hInserted = m_tree.Ins
                ertItem( strInsert, hParent, hIns
                ertAfter);

                delete pInfo;
                InsertTree( strPath, hInserted);
            }
        }
    }
}
```



**DIE XREGISTRY-KLASSE** ist mächtig genug, um einen kleinen Registry Browser mit nur wenigen Zeilen Quellcode zu generieren.