L U K U 6

Tietojen tallentaminen

Tässä luvussa

Tietoja voidaan tallentaa pysyvästi tallennusvälineelle, tyypillisesti kiintolevylle, joka tallentaa tietoihin tehdyt muutokset sovelluksen käyttökertojen välillä.

Tässä luvussa opit, kuinka tallennat sovelluksesi tiedot ja asetukset. Opit, kuinka MFC:n **CFile**-luokkaa käytetään tiedostojen luku ja kirjoitusoperaatioissa (I/O); kuinka MFC-sovelluskehyksen serialisointiominaisuuksia käytetään sovelluksen rakenteellisen tiedon tallentamiseen ja lukemiseen ja kuinka yksittäisiä tietoja, kuten sovelluksen asetuksia, voidaan tallettaa järjestelmärekisteriin.

Tietoja voidaan tallentaa myös sovelluksen ulkopuolisiin tietokantoihin. Tietokan-tayhteyksiä käsitellään luvussa 7.

Ennen kuin aloitat

Ennen tämän luvun aloittamista sinun tulisi lukea luvut 2-5 ja suorittaa niihin liittyvät harjoitukset.

Oppitunti 1: Tiedosto-I/O

Tässä luvussa kerrotaan MFC:n sisältämistä tiedosto-I/O-luokista. Opit myös tuntemaan **CFile**-luokan sisältämät tiedoston käsittelyn perusfunktiot ja kuinka käytetään siitä periytettyä luokkaa **CStdioFile**, joka sisältää puskuroidun sarjatiedonkäsittelyn vaatimat toiminnot.

Tämän oppitunnin jälkeen:

- Osaat luoda uusia tiedostoja ja avata olemassaolevia tiedostoja käyttämällä **CFile**-luokan jäsenfunktioita.
- Tiedät, kuinka teksti ja binaaritiedostoja käsitellään **CStdioFile**-luokan avulla.
- Tiedät, kuinka tiedostojen käsittelyn aikana esiintyviä virheitä käsitellään.

Oppitunnin arvioitu kesto: 25 minuuttia

CFile-luokka

MFC:n **CFile**-luokka toteuttaa binaaritiedostojen käsittelyssä tarvittavat toiminnot kapseloimalla käyttöjärjestelmän tiedostokahvan ja tarjoamalla metodit tiedostojen avaamista, lukemista, kirjoittamista ja sulkemista varten. **CFile** käsittelee tiedos-toja suoraan ilman puskurointia. **CFile**:stä periytetty **CStdioFile**-luokka toteuttaa MFC:ssä puskuroidun tiedostojen käsittelyn.

CFile on **CMemFile** ja **CSharedFile** -luokkien kantaluokka. **CMemFile** tukee muistiin tallennettuja tiedostoja (in-memory files), jotka tallennetaan levyn sijasta RAM-muistiin toimintojen nopeuttamiseksi. **CSharedFile** sisältää tuen jaetuille muistiin talletetuille tiedostoille.

Tiedostojen avaaminen ja sulkeminen

CFile-luokan avulla voidaan tiedostojen avaaminen tehdä useammilla tavoilla. Luokan muodostimen avulla voidaan määritellä avattava tiedosto. Tämä muodostin antaa sinulle mahdollisuuden muodostaa **CFile**-objektin ja liittää sen tiedostoon yhdellä komennolla. Tiedoston määrittely suoraan muodostimessa sisältää riskin. Jos tiedostoa ei ole, syntyy virhe.

On yleensä parempi edetä vaiheittain. Ensin luodaan **CFile**-objekti ja sen jälkeen yhdistetään luotu objekti tiedostoon. Tämä lähestymistapa on joustavampi virheiden käsittelyssä ja auttaa myös selkiyttämään koodissasi esiintyvän tiedosto-objektin ja levyllä fyysisesti olevan tiedoston yhteyttä.

Avattaessa tiedostoa vaiheittain luodaan ensin **CFile**-objekti ilman parametrejä. Seuraavaksi kutsutaan objektin **CFile::Open()**-funktiota, jonka avulla määri-

tellään tiedoston sijainti ja määritellään liput, jotka osoittavat tiedoston käsittelyn ja jakamisen tilat. Esimerkiksi tutkitaan seuraavaa listausta, jossa tiedosto avataan vaiheittain:

```
CString strPath = “MyFile.bin”;
CFile MyFile;
BOOL bResult = MyFile.Open(strPath, CFile::modeRead);
if(!bResult)
    AfxMessageBox(strPath + “ could not be opened”);
```

Jos tiedosto määritellään ilman hakupolkua **Open**-komento etsii tiedostoa ensin nykyisestä kansiosta ja sen jälkeen järjestelmän polkumäärittelyn mukaisista kansioista. Voit myös määritellä sijainnin suhteessa nykyiseen kansioon tai käyttää kansion UNC-nimeä (Universal Naming Convention). UNC-nimi määrittelee tiedoston sijainnin koneesta riippumattomalla tavalla ja muotoutuu seuraavasti: *\\Palvelimen nimi\Verkko jako\Tiedoston polku*.

Kun määrittelet avattavan tiedoston sijaintia, älä tee mitään oletuksia tiedoston sijainnista. Esimerkiksi polun määrittäminen seuraavalla tavalla olettaa, että MyFile.bin-tiedosto on C:\\Program Files\\MyApp-kansiossa:

```
CString strPath = C:\\Program Files\\MyApp\\MyFile.bin;
```

Tämä olettamus ei pidä paikkaansa, jos sovellus on asennettu johonkin muualle kuin C-asemalle. Sovellus tai sen asennusohjelma voi tallettaa käyttämiensä tiedostojen sijainnit ympäristömuuttujiin tai rekisteriin (katso tämän luvun oppitunti 3).

Open()-funktion toinen parametri on bittijono, joka määrittelee tiedoston käsittelyyn vaikuttavat tilat. Bittijonon arvot määritellään lueteltuina vakioina **CFile**-luokan määrittelyissä, mikä merkitsee sitä, että lippujen täytyy olla luokan määrit telyissä hyväksyttyjä. Käsittelytila määrää onko tiedosto avattu vain lukemista, vain kirjoittamista vai sekä lukemista että kirjoittamista varten. Jakotila määrittää voiko toinen prosessi käsitellä tiedostoa sen ollessa avoinna. Voit luoda uuden tiedoston käyttämällä **CFile::modeCreate**-arvoa.

Open()-funktiolle välitettäviä bittijonoja voidaan yhdistellä käyttämällä biteittäistä **OR**-operaattoria (|). Yleensä tulisi määritellä sekä käsittelytila että jakotila. Seuraava koodi esimerkiksi avaa MyFile.bin-tiedoston lukemista ja kirjoittamista var-ten:

```
MyFile.Open(“MyFile.bin”, CFile::modeReadWrite|CFile::shareDenyWrite);
```

Jos tiedosto MyFile.bin saadaan onnistuneesti avattua tällä funktion kutsulla, kaikilta muilta prosesseilta estetään tämän tiedoston käsittely. Jos tiedostoa MyFile.bin ei löydy, se luodaan.

Taulukossa 6.1 on yhteenveto **CFile**-luokassa määritellyistä käsittely- ja jakotiloista.

Taulukko 6.1 Käsittely- ja jakotilat

Lipun arvo	Toiminto
CFile::modeCreate	Luo uuden tiedoston. Jos tiedosto on jo olemassa, se korvataan tyhjällä tiedostolla.
CFile::modeNoTruncate	Voidaan käyttää **CFile::modeCreate**-arvon yhteydessä varmistamaan, että määrätyn tiedoston ollessa jo olemassa se avataan korvaamisen sijaan. Eli tiedoston avaaminen on varmistettu, koska jos tiedostoa ei ole olemassa, se luodaan. Tämä arvo voi olla käyttökelpoinen sellaisten asetustiedostojen avaamisessa, joiden olemassaolosta ei olla varmoja.
CFile::modeRead	Tiedosto avataan vain lukemista varten.
CFile::modeReadWrite	Tiedosto avataan luku/kirjoitus-tilassa.
CFile::modeWrite	Tiedosto avataan vain kirjoittamista varten.
CFile::shareDenyNone	Ei estä muita prosesseja lukemasta tiedostoa tai kirjoittamasta siihen.
CFile::shareDenyRead	Estää muita prosesseja lukemasta tiedostoa.
CFile::shareDenyWrite	Estää muita prosesseja kirjoittamasta tiedostoon.
CFile::shareExclusive	Estää muilta prosesseilta sekä tiedoston lukemisen että siihen kirjoittamisen.

Tiedostovirheet

On helppo nähdä, että hyvin moni asia voi aiheuttaa ongelmia **CFile::Open()**-funktiolle. Yritys avata tiedostoa, jota ei ole olemassa (jos **CFile::modeCreate** lippua ei ole määritelty) aiheuttaa **Open()**-funktion toiminnan epäonnistumisen. Toisen prosessin yksinoikeudella avaaman tiedoston avaamisyritys aiheuttaa myös virheen. Lukematon määrä ympäristötekijöitä voi aiheuttaa virheitä työskenneltäessä tiedostojen kanssa.

CFile::Open()-funktion käyttö esitelleessä esimerkissä, funktio palauttaa Boolean-arvon, joka ilmoittaa onnistumisesta tai epäonnistumisesta. Jos törmätään virheeseen, halutaan yleensä tietää, miksi operaatio epäonnistui. Voit välittää tämän tiedon käyttäjälle ja ehdottaa tapaa, jolla ongelma korjataan.

MFC sisältää **CFileException**-luokan (periytetty **CException**-kantaluokasta), joka esittää tiedonsiirtovirheen tilan. MFC-poikkeusluokka sisältää jäsenmuuttujia ja -luokkia, joiden avulla voit saada tietoja virheestä, joka aiheutti poikkeuksen.

CFile:n muodostimen versio, joka yrittää avata tiedoston, nostaa poikkeuksen **CFileException** virheen sattuessa. Jos käytät tätä muodostinta, sinun tulisi sulkea se *try/catch* poikkeustenkäsittelylohkoon, kuten seuraavassa:

```
try
{
    CFile MyFile("MyFile.old", CFile::modeRead);
}
catch(CFileException * fx)
{
    TCHAR buf[255];
    fx->GetErrorMessage(buf, 255);
    CString strPrompt(buf);
    AfxMessageBox(strPrompt);
}
```

Poikkeusten käsittelystä on kerrottu enemmän luvun 13 oppitunnilla 1.

CFile::Open()-funktio ei nosta poikkeusta. Sen sijaan sille voidaan antaa kolmas valinnainen parametri, joka on osoitin **CFileException**-objektiin. Jos tiedoston avaaminen epäonnistuu, **CFileException**-objekti varustetaan tiedoilla virheen luonteesta. Tätä tietoa voidaan hyödyntää jäljempänä koodissa välitettäessä tietoa käyttäjälle, kuten seuraavassa:

```
CFile MyFile;
CFileException fx;

if(!MyFile.Open("MyFile.old", CFile::modeRead, &fx))
{
    TCHAR buf[255];
    fx.GetErrorMessage(buf,  255);
    CString strPrompt(buf);
    AfxMessageBox(strPrompt);
}
```

Tiedostojen sulkeminen

CFile-luokalla on **Close()**-metodi avoimen tiedoston sulkemista varten. Seuraava koodi on esimerkki **Close()**-funktion käytöstä:

```
BOOL MyFileFunction()
{
    CFile MyFile;
    if(!MyFile.Open("MyFile.old", CFile::modeRead))
    {
        AfxMessageBox("Cannot open MyFile.old");
        return FALSE;
    }
    // Do something with the file . . .
    MyFile.Close();
    return TRUE;
}
```

Tässä esimerkissä **MyFile.Close()**-funktion kutsuminen ei ole täysin välttämätöntä. **CFile**:n tuhooja, jota kutsutaan objektin poistuessa, kutsuu **Close()**-funktiota, jos se huomaa, että näin ei ole jo tehty. On kuitenkin hyvän ohjelmoin- titavan mukaista aina kutsua **CFile::Open()**-funktioon liittyvää **CFile::Close()**-funktiota.

Voit käyttää **Close()**-funktiota **CFile**-objektin irrottamiseen tiedostosta ennen kuin käytät objektia uudelleen toisen tiedoston avaamiseen. Esimerkiksi seuraava koo-di luo kolme uutta tiedostoa, file1.txt, file2.txt ja file3.txt nykyisen sovelluksen kansioon. Kaikki kolme tiedostoa luodaan samaa **CFile**-objektia käyttämällä:

```
CString  strFiles[3] = { “file1.txt”, “file2.txt”, “file3.txt” };
CFile file;

for(int i = 0; i < 3; i++)
{
      file.Open(strFiles[i], CFile::modeCreate);
      file.Close();
}
```

Tiedostojen lukeminen ja kirjoittaminen

CFile sisältää **Read()** ja **Write()** -funktiot, jotka kutsuvat Microsoft Windows API-funktioita **ReadFile()** ja **WriteFile()**, tarjoten suorat puskuroimattomat kirjoitus- ja lukutoiminnot. Koska suora tiedostojen käsittely on aika konstikasta, C:n suoritusaikainen kirjasto sisältää *stream I/O* -funktiot. (Esimerkiksi puskurit ja osoittimien siirtymät täytyy määritellä yksikköinä, jotka ovat levyn sektorikoon kerrannaisia.) Stream I/O -funktioiden avulla voit käsitellä levyllä olevia tietoja yksittäisistä merkeistä suuriin tietorakenteisiin. Nämä funktiot sisältävät myös suorituskykyä parantavat I/O-puskurit. Stream I/O-funktioihin kuuluvat esimerkiksi **fopen()**, **fseek()**, **fread()** ja **fwrite()**. C++-ohjelmoijana saatat tuntea stream I/O:n käytön iostream-luokkien kautta.

MFC:n stream file I/O toteutetaan **CStdioFile**-luokan kautta. **CStdioFile**n versiot **Read()** ja **Write()** -funktioista käyttävät runtime stream I/O -funktioita. Jollet erityisesti tarvitse matalan tason levynkäsittelyä, sinun tulisi käyttää **CStdio File**-luokkaa, joka on joustava ja helppokäyttöinen.

Levytiedostot yhdistetään **CStdioFile**-objektiin, joka voidaan avata *tekstitilassa* (text mode) tai *binaaritilassa* (binary mode). Tekstitilassa vaunupalautus/rivinvaihto (CR/LF) -pari saa erityiskäsittelyn. Kun kirjoitat rivivaihtomerkin (0x0A) tekstitilassa olevaan **CStdioFile**-objektiin, kirjoitetaan tiedostoon merkkipari CR/LF (0x0D, 0x0A). Kun luet tietoja tiedostosta, joka on tekstitilassa, tavupari 0x0D, 0x0A muutetaan yhdeksi 0x0A tavuksi. Avataksesi **CStdioFile**-objektin

tekstitilassa, sinun täytyy lähettää **CFile::typeText**-lippu **Open()**-funktiolle seuraavaan tapaan:

```
CStdioFile inFile("MyFile.txt", CFile::modeRead | CFile::typeText);
```

Avattaessa **CStdioFile**:n binaaritilassa käytetään **CFile::typeBinary**-lippua. Binaa-ritilassa rivinvaihtoja ei muuteta.

Tietojen lukemiseen **CStdioFile** tiedosto-objektista voidaan käyttää joko **Read()-**funktiota tai **ReadString()**-funktiota.

Read() ottaa osoittimen puskuriin, joka sisältää tiedostosta luetut tiedot ja etumerkittömän kokonaisluvun (UINT), joka ilmaisee luettujen tavujen määrän. **Read()**-funktio palauttaa luettujen tavujen määrän. Jos vaadittua tavumäärää ei voida tiedoston päättymisen vuoksi lukea, ilmoitetaan todellisuudessa luettujen tavujen määrä.

Jos virheitä ilmenee, nostetaan **CFileException**-poikkeus. Kun kirjoitat tekstitilassa olevaan tiedostoon, käytä **ReadString()**-funktiota, joka on samanlainen kuin **Read()**, paitsi että:

- Lukeminen pysähtyy rivinvaihtoon.
- Lopussa oleva tyhjä merkki liitetään puskuriin.
- Puskurin osoitin palautetaan. Tämä osoitin sisältää NULL arvon, jos tiedoston loppuun tultiin lukematta yhtään tietoa.

ReadString() tarjoaa helpon tavan yksittäisen rivin lukemiseen tekstitiedostosta merkkijonoon. Siitä on olemassa versio, joka palauttaa onnistumisen tai epäonnistumisen osoittavan BOOL-arvon.

Write() on toiminnaltaan saman kaltainen kuin **Read()**. Se ottaa osoittimen puskuriin, joka sisältää tiedon kirjoitettavat tavut ja luettavien tavujen määrän. Kirjoitettujen tavujen määrää ei **Write()**-funktiota käytettäessä palauteta. Jos virheitä ilmenee, eikä kaikkia määrättyjä tavuja voida kirjoittaa, nostetaan **CFileException**-poikkeus. Kun kirjoitat tekstitilassa olevaan tiedostoon, käytä **CStdioFile::WriteString()**-funktiota, joka kirjoittaa rivinvaihtomerkin.

Seuraava koodi havainnollistaa, kuinka **CStdioFile**-luokkaa käytetään levyllä olevien tiedostojen lukemiseen ja kirjoittamiseen. Koodi avaa MyFile.bin-tiedoston binaaritilassa ja lukee tietoa 10 bitin lohkoissa. Jokainen lohko kirjoitetaan uudeksi riviksi luotuun Output.txt-tiedostoon, kuten seuraava esimerkki osoittaa:

```
try
{
    CStdioFile inFile("MyFile.bin", CFile::modeRead |
        CFile::typeBinary);
    CStdioFile outFile("outfile.txt", CFile::modeCreate |
```

```
            CFile::modeWrite | CFile::typeText);

        const UINT linelength = 10;
        TCHAR strBuf[16];

        while(inFile.ReadString(strBuf, linelength))
        {
            _tcscat(strBuf, _T("\n"));
            outFile.WriteString(strBuf);
        }
}

catch(CFileException * fx)
{
        TCHAR buf[255];
        fx->GetErrorMessage(buf, 255);
        AfxMessageBox(buf);
}
```

Hajasaantitiedostot

Tekstitiedostoja luetaan ja kirjoitetaan usein järjestyksessä rivi kerrallaan. Kun luetaan binaaritiedostoja, täytyy usein käyttää *hajasaantia* (random access). Hajasaanti antaa mahdollisuuden siirtyä suoraan tiettyyn paikkaan tiedostossa ja välittömästi hakea siellä sijaitsevat tiedot.

Avoin tiedosto ylläpitää *tiedosto-osoitinta* (file pointer), joka määrittää seuraavan luettavan bitin tai sijainnin, johon seuraava bitti kirjoitetaan. Kun tiedosto avataan, tiedosto-osoitin sijoitetaan tiedoston alkuun. Voit muuttaa osoittimen sijaintia käyttäen **CFile::Seek()**-funktiota. **CFile::Seek()** siirtää tiedoston osoitinta määrätyn siirtymän verran tiedoston alusta tai lopusta tai suhteessa nykyiseen sijaintiin. Luku- ja kirjoitusoperaatiot siirtävät tiedosto-osoitinta luettavien tai kirjoitettavien bittien määrää vastaavalla matkalla.

Oppitunnin yhteenveto

MFC-luokka **CFile** sisältää suoran puskuroimattoman I/O:n levyllä oleviin tiedostoihin. **CFile** on myös puskuroidun stream I/O:n C:n suorituksenaiakaisten kirjastojen avulla tarjoavan **CStdioFile**-luokan kantaluokka. **CStdioFile**-luokka antaa mahdollisuuden avata tiedostoja *tekstitilassa* (text mode) (jossa rivin vaihdot muutetaan CR/LF pareiksi) tai *binaaritilassa* (binary mode) (jossa rivin vaihtoja ei muunneta). Jos tarvitset erityisesti suoraa levyllä olevien tiedostojen käsittelyä, voit käyttää **CFile**-luokkaa. Normaalisti tulisi tiedostojen käsittelyssä tulisi käyttää **CStdioFile**-luokkaa.

CFile:stä johdetut objektit joko yhdistetään luontivaiheessaan levyllä olevaan tiedostoon (käyttämällä sopivaa muodostinta), tai ne luodaan vaiheittain niin, että ensin luodaan **CFile**:stä johdettu objekti ja tämän jälkeen kutsutaan sen **Open()**-funktiota. **Open()**-funktiolle voidaan välittää sekä avattavan tiedoston polku, että lippuarvo, joka määrittää tiedoston käsittely- ja jakotilat. Käsittelytila määrittelee, avataanko tiedosto vain lukemista tai kirjoittamista varten vai lukemista ja kirjoittamista varten, sekä luodaanko uusi tiedosto. Jakotila määrittää, millä tasolla muut prosessit voivat tiedostoa käyttää sen ollessa avattuna.

Monet **CFile**-jäsenfunktiot nostavat **CFileException**-tyyppisiä MFC-poikkeuksia. **CFileException**-objektien avulla voidaan selvittää poikkeuksen aiheuttaneesta virheestä saadut tiedot. **Open()**-funktio ottaa poikkeusobjektin parametrinä.

Avattujen tiedostojen lukemiseen ja kirjoittamiseen voit käyttää **CFile:**n **Read()** ja **Write()** -funktioita, tai voit hyödyntää **CStdioFile::ReadString()** ja **CStdioFile ::WriteString()** -funktioita. Sekä **Read()** että **ReadString()** sijoittavat luetut kirjaimet puskuriin, jonka olet funktiolle toimittanut. **ReadString()** keskeyttää lukemisen tavatessaan rivinvaihtomerkin ja lisää puskurissa olevan tekstin loppuun null-merkin.

Käytä **Write()**-funktiota puskurissa olevan tekstin tiedostoon kirjoittamiseen. Kun kirjoitat tekstitilassa olevaan tiedostoon, käytä **CStdioFile::WriteString()**-funktiota, joka kirjoittaa rivinvaihtomerkin CR/LF-parina.

Hajasaantitilassa avattu tiedosto ylläpitää *tiedosto-osoitinta* (file pointer), joka määrittää seuraavan luettavan tai kirjoitettavan tavun sijainnin. Käytä **CFile:: Seek()**-funktiota halutessasi siirtää tiedosto-osoitinta määrätyn siirtymän verran tiedostossa ennen kirjoitus- tai lukuoperaatiota.

Oppitunti 2: Sovelluksen tietojen serialisointi

Tietojen tallennuksen toteuttamiseen liittyy erityisiä ongelmia oliopohjaisia sovelluksia ohjelmoitaessa. Ohjelmoijien täytyy pohtia, kuinka sovelluksen objektien rakenne ja niiden väliset suhteet saadaan säilytettyä tallennettaessa ja haettaessa tietoja. Tämä saattaa olla ongelmallista, koska tiedostoissa tieto varastoidaan yleensä rakenteettomana bittivirtana. Tämän ongelman korjaamiseksi MFC:n sovelluskehyksessä on mahdollista käyttää *serialisointia* (serialization), jonka avulla sovelluksen tietojen objektirakenne voidaan säilyttää tallennettaessa ja avattaessa tietoja.

Oppitunnin jälkeen:

- Tiedät, kuinka MFC sovelluskehys toteuttaa serialisoinnin.
- Osaat käyttää ylikuormitettuja << ja >> operaattoreita sisäisten sekä MFC:n tyyppien serialisointiin.
- Osaat tehdä luokasta serialisoituvan.
- Osaat serialisoida MFC-kokoelman.

Oppitunnin arvioitu kesto: 50 minuuttia

MFC:n tuki serialisoinnille

MFC:ssä on **CObject**-luokassa sisään rakennettu tuki serialisoinnille. Kaikkien serialisointia käyttävien luokkien tulee periytyä **CObject**-luokasta ja niiden täytyy ylikuormittaa **CObject::Serialize()**-funktio. **Serialize()**-funktion tehtävä on arkistoida luokan valitut tietojäsenet ja tallentaa tai palauttaa ne MFC-luokan **CArchive**-objektista.

CArchive-objekti toimii välittäjänä objektin ja tallennusvälineen välissä. **CArchive**-objekti on aina yhteydessä **CFile**-objektiin. **CFile**-objekti edustaa yleensä levyllä olevaa tiedostoa, mutta se voi edustaa myös muistissa olevaa tiedostoa. Voit esimerkiksi serialisoida Windowsin leikepöydällä olevat tiedot yhdistämällä **CArchive**-objektin **CSharedFile**-objektiin. Lisäksi **CArchive**-objektissa on tyyppiturvallinen puskurointimekanismi serialisoituvien objektien lukemiseksi ja kirjoittamiseksi **CFile**-objektiin.

Yhtä **CArchive**-objektia käytetään joko tietojen tallentamiseen tai lataamiseen, muttei koskaan molempiin. **CArchive**-objektin elinaika rajoittuu yhteen siirtoon, eli joko objektien tiedostoon kirjoittamiseen tai objektien lukemiseen tiedostosta. Tietojen serialisointi tiedostoon ja lukeminen tiedostosta vaatii erillisten

CArchive-objektien luomista. **CArchive**-objektin tila, eli käytetäänkö sitä lukemiseen vai kirjoittamiseen, voidaan määrittää **CArchive::IsStoring()**-funktion boolean-paluuarvon perusteella.

CArchive-luokassa määritellään sekä lisäys- (<<) että poisto-operaattorit (>>). Näitä operaattoreita käytetään vastaavalla tavalla kuin C++:n normaalien tietovirtaluokkien vastaavia operaattoreitakin seuraavan koodin esittämällä tavalla:

```
if (ar.IsStoring())
{
      ar << m_string;
}
else
{
      ar >> m_string;
}
```

Voit käyttää lisäys- ja poisto-operaattoreita tietojen tallentamiseen ja lukemiseen **CArchive**-objektista. Taulukossa 6.2 on luettelo tietotyypeistä ja objekteista, joiden yhteydessä lisäys- ja poisto-operaattoreita voidaan käyttää.

Taulukko 6.2 Tietotyypit ja objektit, joiden kanssa lisäys- ja poisto-operaattorit toimivat

CObject*	SIZE ja CSize	float
WORD	CString	POINT ja CPoint
DWORD	BYTE	RECT ja CRect
double	LONG	CTime ja CTimeSpan
int	COleCurrency	COleVariant
COleDateTime	COleDateTimeSpan	

Serialisointiprosessi

Luvun 3 oppitunnilla 4 opit, että sovelluksen tiedot tallennetaan sovelluksen dokumenttiobjektiin. Sovelluksen tiedot serialisoidaan levyllä olevaan tiedostoon ja tiedot palautetaan tiedostosta dokumenttiobjektiin. Dokumentin tiedostotyyppi yhdistetään sovellukseen määrittelemällä tiedostontarkennin **Advanced Options** -dialogissa AppWizardin vaiheessa 4 (katso luvun 2 oppitunti 1).

Dokumenttiobjekti suorittaa sovelluksen tietojen serialisoinnin käyttäjän antamien tiedostokomentojen seurauksena. Sovelluskehys luo sopivan tyyppisen (riippuen siitä tallennetaanko vai ladataanko tietoja) **CArchive**-objektin ja välittää sen parametrinä dokumenttiobjektin **Serialize()**-funktiolle.

AppWizard tekee dokumenttiluokkaan tyngän funktiosta **Serialize()**. Tietojen hakemiseen tai tietojen tallentamiseen liittyvä koodi täytyy lisätä tähän funktioon. Yksinkertaisia tietojäseniä voidaan tallentaa ja ladata käyttämällä operaattoreita << ja >>. Jos dokumenttiobjekti sisältää monimutkaisempia objekteja, joilla on oma serialisointikoodinsa, täytyy kutsua näiden objektien omia **Serialize()**-funktioita ja välittää niille viittaus nykyiseen arkistoon.

Ajatellaan esimerkiksi TestApp-sovellusta, jonka dokumenttiluokassa on kolme tietojäsentä, kuten seuraavassa koodiesimerkissä nähdään:

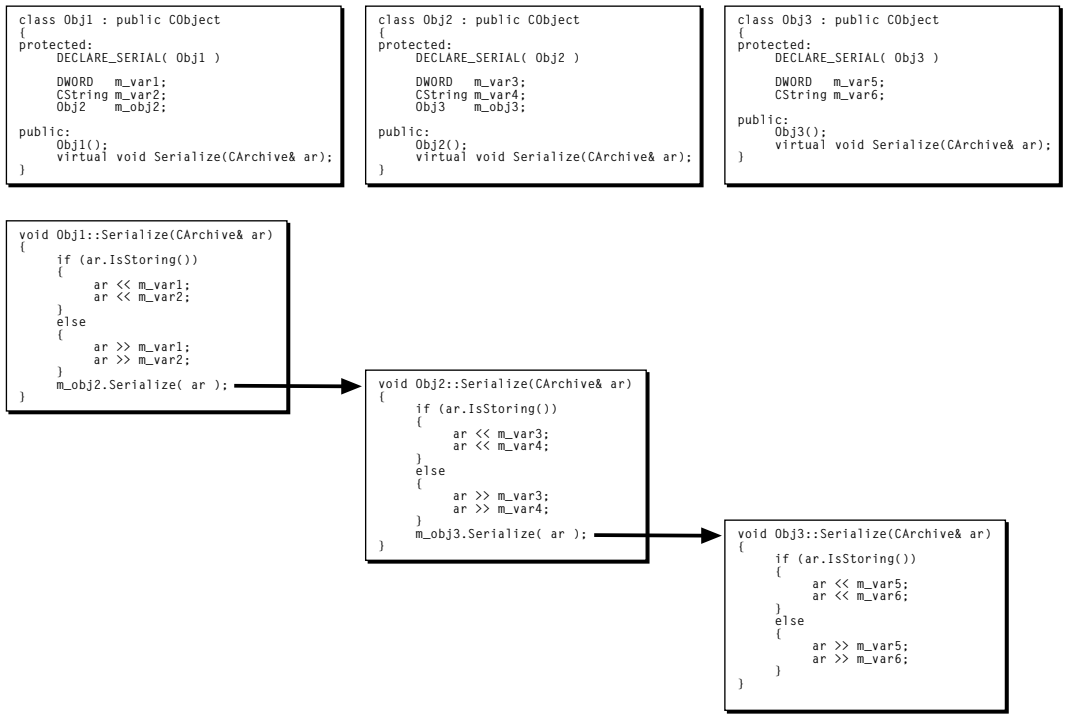
```
Class CtestAppDoc
{
    CString m_string;
    DWORD m_dwVar;
    MyObj m_obj;
}
```

Oletetaan, että **MyObj**-luokka on serialisoituva.

Seuraavassa nähdään esimerkki **Serialize()**-funktiosta, joka voitaisiin kirjoittaa **TestApp**-dokumenttiluokalle:

```
void CTestAppDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_string;
        ar << m_dwVar;
    }
    else
    {
        ar >> m_string;
        ar >> m_dwVar;
    }
    m_obj.Serialize(ar);
}
```

Huomaa, kuinka **MyObj::Serialize()**-funktiota kutsutaan testilohkojen ulkopuolella, koska se sisältää omat päättelynsä, joilla määrätään luetaanko vai kirjoitetaanko tietoja. Kuvassa 6.1 havainnollistetaan tapaa, jonka avulla tätä tekniikkaa voidaan soveltaa rekursiivisesti.



Kuva 6.1 Sisäkkäisten objektien serialisointi

Niin kauan kuin serialisointirutiinit pysyvät yhdenmukaisina, voidaan monimutkaisia objektirakenteita tallentaa ja palauttaa levyltä sovellukseen. Erityisesti on kiinetettävä huomioita siihen, että **Serialize()**-funktion tallennus- ja lukuhaarat täsmäävät — eli toisin sanoen, että kaikki objektit palautetaan sovellukseen samassa järjestyksessä, jossa ne on tallennettu.

Serialisointirutiinit palauttavat objektirakenteen entiselleen, kun rakenne ladataan levyltä. Tämä voidaan tehdä, koska tallennettaessa tiedostoon objektin arvon lisäksi tallennetaan myös sen tyyppi. Kun objekti palautetaan muistiin, tallennetun informaation perusteella päätetään millaisia objekteja luettavia arvoja varten täytyy luoda. Serialisointirutiinit hoitavat objektien luomisen automaattisesti. Jotta tämä toiminto toimisi oikein on kaikkiin serialisoitaviin luokkiin sisällytettävä oletusmuodostin (muodostin, jolla ei ole argumentteja).

Seuraavassa harjoituksessa opit mitä vaiheita tarvitaan yksinkertaisen sovelluksen tietojen serialisoinnin toteuttamiseen.

Sovelluksen tietojen serialisointi

Luvun 5 oppitunnilla 1 lisäsit kaksi sovelluksen tiedot sisältävää jäsenmuuttujaa, CMyAppDoc::m_nLines ja CMyAppDoc::m_string, MyApp-sovellukseen. Seuraavaksi tähän projektiin lisätään koodi, joka serialisoi nämä tietoyksiköt dokument-titiedostoon.

► **MyApp-sovelluksen tietojen serialisointi**

1. Etsi AppWizardin luoma **Serialize()**-funktio luokasta **CMyApp**. **Serialize()**-funktioon luotu koodi näyttää seuraavalta:

```
void CMyApp::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

2. Korvaa //TODO-kommentit koodilla, joka suorittaa tallentamisen ja tietojen noutamisen takaisin muuttujiin. Valmiin koodin tulisi näyttää seuraavalta:

```
void CMyAppDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_nLines;
        ar << m_string;
    }
    else
    {
        ar >> m_nLines;
        ar >> m_string;
    }
}
```

3. Etsi **CMyAppDoc::OnDataEdit()**-funktio. Lisää funktion loppuun välittömästi **UpdateAllViews()**-kutsun jälkeen seuraava rivi:

```
SetModifiedFlag();
```

Funktion tulisi tämän jälkeen näyttää kokonaisuudessaan seuraavalta:

```
void CMyAppDoc::OnDataEdit()
{
    CEditDataDialog aDlg;

    aDlg.m_nLines = m_nLines;
    aDlg.m_strLineText = m_string;

    if(aDlg.DoModal())
    {
        m_nLines = aDlg.m_nLines;
        m_string = aDlg.m_strLineText;

        UpdateAllViews(NULL);
        SetModifiedFlag();
    }
}
```

CDocument::SetModifiedFlag()-funktiota kutsumalla ilmoitetaan sovelluksen kehykselle, että sovelluksen tietoja on muutettu. Tämä funktio saa sovelluskehyksen muistuttamaan käyttäjää tallentamisesta ennen dokumentin sulkemista.

4. Ylikuormita ClassWizardia käyttämällä **CDocument::DeleteContents()**-funktio **CMyAppDoc**-luokassa. Korvaa automaattisesti muodostetun koodin kommentti seuraavilla riveillä:

```
m_nLines = 0;
m_string = "";
```

Funktion tulisi näyttää nyt kokonaisuudessaan seuraavalta:

```
void CMyAppDoc::DeleteContents()
{
    m_nLines = 0;
    m_string = "";

    CDocument::DeleteContents();
}
```

Toisin kuin MDI-sovellukset, jotka luovat uuden dokumenttiobjektin aina uutta dokumenttia avattaessa, SDI-sovellus luo vain yhden dokumenttiobjektin, joka uudelleenkäytetään aina, kun uusi dokumentti luodaan tai avataan. Funktio **DeleteContents()** poistaa dokumenttiobjektissa olevat tiedot ennen sen uudelleenkäyttöä. SDI-sovellusta ohjelmoitaessa on tehtävä **DeleteContents()**-funktio, joka asettaa dokumenttiobjektin kaikkien jäsenten arvoksi nollan tai tyhjän arvon. Muussa tapauksessa edellisen dokumentin tiedot sekoittuvat uuteen dokumenttiin.

► **MyApp serialisoinnin kokeileminen**

1. Käännä ja käynnistä MyApp-sovellus.
2. Valitse **Edit**-toiminto **Data**-valikosta, kirjoita näytettävä teksti ja kuinka monesti teksti toistetaan. Sulje **Edit Document Data** -dialogi napauttamalla **OK**.
3. Valitse **File**-valikosta **New**. Ruudulle avautuu dialogi, joka kehottaa sinua tallentamaan muutokset nimettömään MyApp-dokumenttiin ennen uuden dokumentin avaamista. Dialogi avautui koska ohjelmassa kutsuttiin funktiota **SetModifiedFlag()** sen jälkeen, kun muutit sovelluksen tietoja.
4. Napauta **Yes**. Tallenna dokumentti **Save As** -dialogia käyttäen nimellä **MyFile.mya** nykyiseen kansioon.
5. Toinen nimeämätön tiedosto avautuu näytölle. Tekemäsi **DeleteContents()**-funktio on tyhjentänyt MyFile.mya-dokumentin tiedot.
6. Avaa MyFile.mya-tiedosto valitsemalla **Open**-toiminto **File**-valikosta, tai valitsemalla **MyFile.mya File**-valikon viimeksi käsiteltyjen tiedostojen luettelosta.
7. Sovelluksen tiedot ladataan takaisin ja näytetään siinä muodossa kuin ne olivat tallennettaessa, sekä sovellusikkunassa että **Edit Document Data** -dialogin muokkausruuduissa.

Serialisoituvan luokan tekeminen

Olet jo nähnyt, kuinka dokumenttiobjekti voi sisältää objekteja, jotka toteuttavat oman serialisointinsa. Serialisoituvaa luokkaa tehtäessä täytyy suorittaa seuraavat vaiheet:

1. Periytä luokka **CObject**-luokasta tai **CObject**-luokasta periytetystä luokasta.
2. Tee luokkaan oletusmuodostin (jossa ei käytetä argumentteja).
3. Lisää MFC-makro **DECLARE_SERIAL** luokan määrittelyyn header-tiedostoon. **DECLARE_SERIAL** ja sen pari **IMPLEMENT_SERIAL** sisältävät luokkasi MFC ajonaikaisen infirmaatin. Nämä makrot sisältävät myös yleisen

>>-operaattorin, joka käyttää ajonaikaisia tietoja palauttaessaan luokan objek-teja arkistosta. **DECLARE_SERIAL** ottaa parametrinä luokan nimen.

Seuraavassa koodissa havainnollistetaan serialisoituvan luokan määrittelyä:

```
// MyClass.h
class CMyClass : public CObject
{
    DECLARE_SERIAL(CMyClass)

public:
    CMyClass() {;}  // Default constructor

    virtual void Serialize(CArchive& ar);

};
```

4. Lisää **IMPLEMENT_SERIAL**-makro luokan toteutustiedostoon (.cpp). **IMPLEMENT_SERIAL**-makro ottaa kolme parametriä: serialisoitavien luok-kien määrän, kantaluokan nimen ja skeeman numeron, kuten seuraavassa:

   ```
   IMPLEMENT_SERIAL(CMyClass, CObject, 1)
   ```

   Skeeman numero antaa mahdollisuuden dokumenttitiedostojen versioinnin toteuttamiseen. Sovelluksen dokumenttiobjektien rakenne saattaa muuttua versioiden välillä. Tällaiset muutokset aiheuttavat todennäköisesti ongelmia, jos käyttäjä yrittää avata sovelluksen aikaisemmalla versiolla tallennetun tiedoston uudella versiolla.

   Jokaiselle objektin rakennetta muuttavalle sovellusversiolle voidaan määrittää **IMPLEMENT_ SERIAL**-makrossa uusi skeeman numero. Näin voidaan avausrutiiniin liittää koodi, joka tutkii sovelluksen ja dokumenttien välisiä ver-sioristiriitoja ja ryhtyy tarvittaviin toimenpiteisiin, eli näyttää esimerkiksi virhe-ilmoituksen tai suorittaa dokumentin muunnoksen.
5. Tee luokkaan ylikuormitettu versio **CObject::Serialize()**-funktiosta.

MFC-kokoelmaluokkien serialisointi

MFC:n kokoelmaluokat **CArray**, **CList** ja **CMap** sisältävät omat versionsa **Serialize()**-funktiosta, jotka serialisoivat kaikki kokoelmaan kuuluvat elementit.

Oletetaan, että dokumenttiluokka sisältää joukon kokonaislukuarvoja seuraavassa koodiesimerkissä nähtävällä tavalla:

```
CList<int, int &> m_intList;
```

Tämä kokelma voidaan serialisoida lisäämällä seuraava rivi dokumentin **Serialize()**-funktioon:

```
m_intList.Serialize(ar);
```

Tämä rivi on kaikki, mitä yksinkertaisten tyyppien serialisointi vaatii. **CList:: Serialize()** kutsuu globaalia apufunktiomallia **SerializeElements()**, jonka esittely on seuraava:

```
template<class TYPE> void AFXAPI SerializeElements(CArchive& ar, TYPE* pElements, int
nCount);
```

Kääntäjä muodostaa tästä mallista kokoelman elementtien tyyppejä vastaaavan version. **SerializeElements()**-funktion tavanomainen toiminta on kokoelman (johon viittaa osoitin **pElements**) tietojen biteittäinen kopionti arkistoon tai arkistosta.

Tämä oletustoiminto sopii hyvin yksinkertaisilla objekteilla, mutta on ongelmallinen, jos objektirakenne on monimutkaisempi. Oletetaan, että dokumenttiluokalla on seuraavan esimerkkikoodin mukainen jäsen:

```
CList<CMyClass, CMyClass &> m_objList;
```

CMyClass määriteltäisiin seuraavasti:

```
class CmyClass
{
    DECLARE_SERIAL(CMyClass)

public:
    CMyClass() {;}
    int m_int;
    DWORD m_dw;
    CString m_string;
    virtual void Serialize(CArchive& ar);
}
```

Jos m_objList-kokoelma serialisoidaan lisäämällä seuraava rivi dokumenttiobjektin **Serialize()**-funktioon, seuraa ongelmia:

```
m_objList.Serialize(ar);
```

Virheitä syntyy, koska **CMyClass**-objekti sisältää **CStrings**-objektin; joka on monimutkaisia objekteja mukautettuine muistin käsittelyineen ja viittausten laskutapoineen.

m_objlist-kokoelmalle muodostettu oletus **SerializeElements()**-funktio yrittää lukea tai kirjoitaa kokoelman elementtejä biteittäin ja se tulee näin ohittaneeksi

CString-luokassa määritellyt mukautetut << ja >> operaattoreita vastaavat funktiot.

Tässä tapauksessa täytyy kirjoittaa oma versio **SerializeElements()**-funktiosta. Oletettaen, että **CMyClass** on tehty oikein serialisoituvaksi funktioksi, vastaava **SerializeElements()**-funktio saattaisi näyttää seuraavalta:

```
template <> void AFXAPI SerializeElements <CMyClass>
    (CArchive& ar, CMyClass * pNewMC, int nCount)
{
    for (int i = 0; i < nCount; i++, pNewMC++)
    {
        // Serialize each CMyClass object
        pNewMC->Serialize(ar);
    }
}
```

Huomio Yksinkertaisille **CString**-objekteille ei tarvitse tehdä erillistä versiota **SerializeElements()**-funktiosta, sillä MFC:n **CArchive** sisältää sellaisen.

Oppitunnin yhteenveto

MFC-sovelluskehys sisältää serilialisoinniksi kutsutun tekniikan, jonka avulla sovelluksen tietojen objektirakenne voidaan tallentaa ja palauttaa uudelleen käyttöön. Kaikkien serialisointia hyödyntävien luokkien tulee periytyä **CObject**-luokasta ja lisäksi ylikuormittaa **CObject::Serialize()**-funktio.

Serialize()-funktio varastoi ja palauttaa tallennettavat tiedot **CArchive**-luokan objektia käyttäen. Tämä objekti toimii serialisoitavan objektin ja tallennusmedian, joka on yleensä **CFile**-luokan objektiin kapseloitu levyasema, välillä. Lisäksi **CArchive**-objektissa on tyyppiturvallinen puskurointi serialisoitavien tietojen **CFile**-objektista lukemista ja sinne kirjoittamista varten. Tietojen tallentamista ja palauttamista varten täytyy luoda erilliset **CArchive**-objektit. Kun arkistointiob-jekti on luotu, sen roolia (joka selviää **CArchive::IsStoring()**-funktion paluuar-vosta) ei voida muuttaa.

Dokumenttiobjektin serialisointitoiminto käynnistyy, kun käyttäjä valitsee tallennus- tai avaamiskomennon. Sovelluskehys luo **CArchive**-objektin ja välittää sen parametrinä dokumenttiobjektin **Serialize()**-funktiolle.

AppWizard luo dokumenttiluokalle tynkäversion **Serialize()**-funktiosta. Tähän funktioon täytyy lisätä tallentamisessa ja lataamisessa tarvittava koodi. **CArchive** luokassa määritellään << ja >> -operaattorit, joita voidaan käyttää erilaisten C++ ja MFC-tietotyyppien tallentamiseen. Jos objekti sisältää muita

serialisoituvia objekteja, sinun täytyy kutsua näiden objektien omia **Serialize()**-funktioita. Ohjelmoitaessa serialisoituvaa luokkaa täytyy:

- Periyttää luokka **CObject**-luokasta.
- Tehdä luokalle oletusmuodostin.
- Lisätä **DECLARE_SERIAL**-makro luokan määrittelyyn ja **IMPLEMENT_SERIAL**-makro luokan toteutustiedostoon.
- Tehdä luokalle ylikirjoitettu **CObject::Serialize()**-funktio.

MFC kokoelmamalliluokan serialisointi tehdään yksinkertaisesti kutsumalla luokan serialisointifunktiota. On tärkeää muistaa, että kokoelmaluokat toteuttavat serialisoinnin kutsumalla **SerializeElements()**-funktiomallia, joka on luotu kokoelmaluokan elementtityyppiä varten. **SerializeElements()**-funktio suorittaa normaalisti kokoelmaan tallennettujen tietojen bitittäin kopioimisen arkistoon. Jos tämä toiminto ei sovellu kokoelman elementeille, täytyy **SerializeElements()** -funktiomallista tehdä oma toteutus.

Oppitunti 3: Rekisterin käsittely

Windowsin rekisteri on keskitetty, hierarkkisesti järjestetty tietokanta, joka sisältää käyttöjärjestelmän ja sovellusten pysyvät asetustiedot. Windows NT:ssä, Windows 95:ssä, ja Windows 98:ssa rekisteri tarjoaa turvallisen vaihtoehdon 16-bittisten Windows-versioiden .ini-tiedostoille. Tällä oppitunnilla opit, kuinka voit käyttää rekisteriä Windows-sovelluksen käyttäjän tekemien asetusten tallentamiseen ja lataamiseen.

Tämän oppitunnin jälkeen:

- Tunnet rekisterin perusrakenteen.
- Tiedät, kuinka MFC tukee rekisterin ohjelmallista käsittelyä ja kuinka voit käyttää näitä toimintoja sovelluksesi tietojen tallettamiseen ja hakemiseen.
- Tiedät tilanteet, joissa voit joutua käyttämään Win32 APIa rekisterin käsittelemiseen ja tiedät perusasiat funktioista, joiden avulla se tehdään.

Oppitunnin arvioitu kesto: 40 minuuttia

Rekisterin tiedot

Rekisteriä käytetään käyttöjärjestelmän kokoonpanotietojen ja sovellusten asetusten tallentamiseen. Sinne talletetaan kaikki tieto asennetun verkkokortin IP-osoitteesta ja Ohjauspaneelin maa-asetuksista Windowsin Laskin-ohjelman näyttötilaan (tieteellinen vai normaali). Seurauksena tästä rekisteri sisältää yleensä vähintään useita megatavuja tietoa.

Tämän suuren tietomäärän hallinnan helpottamiseksi rekisterin rakenne on loogisesti hierarkkinen. Vaikka rekisterin muodostavien tiedostojen fyysinen rakenne onkin erilainen Windows NT:ssä kuin Windows 95:ssä ja Windows 98:ssa, Win32 API kätkee nämä erot käyttäjiltä ja ohjelmoijilta tarjoamalla yhtenäisen käyttöliittymän tietojen tallentamista ja noutamista varten. Sinun tulisi kuitenkin tuntea rekisterin hierarkia ennen kuin yrität käsitellä rekisteriä ohjelmallisesti. Hyvä väline rekisteriin tutustumiseen on *rekisterieditori* (registry editor).

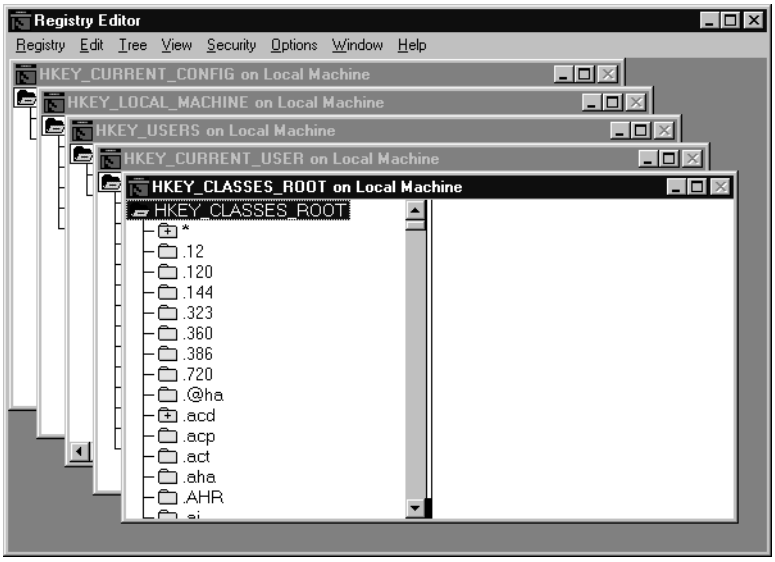
Windows 95 ja Windows 98 rekisterieditori on regedit.exe, joka sijaitsee Windows-kansiossa. Windows NT:n käyttäjät löytävät toisen rekisterieditorin — *regedt32.exe* — Winnt\System32-kansiosta. Regedt32.exe:ssä on muutamia lisä toimintoja, jotka ovat käyttökelpoisia Windows NT:ssä. (Se osaa käsitellä rekisteriavainten turvamäärityksiä ja mahdollistaa **REG_EXPAND_SZ** ja **REG_MULTI_SZ** -tietotyyppien käsittelemisen.) Useimmat käyttäjät pitävät regeditin hakuominaisuuksista, mutta Windows NT:n tutkimiseen regedt32:n vain-luku-tilassa tarjoaa aloittelevalle ohjelmoijalle lisäsuojan vahingossa tapahtuvaa rekisterin vaurioitumista vastaan.

Varoitus Rekisterin asetusten sopimaton muokkaaminen voi estää sovellusten käynnistämisen ja voi jopa johtaa käyttöjärjestelmän pysyvään toimimattomuuteen. Ennen rekisterin muokkaamista tulisi tehdä varmuuskopiot rekisteritiedostoista. Windows 95:ssä tulisi kopioida System.dat ja User.dat, jotka ovat Windows-kansiossa olevia piilotiedostoja. Windows NT:ssä tulisi tehdä päivitetty Emergency Repair Disk -levyke. Lisäohjeita saat käyttöjärjestelmäsi ohjeista.

► **Rekisterin tutkiminen (Windows NT)**

1. Valitse **Käynnistä**-valikosta **Suorita**.
2. Kirjoita **Avaa**-ruutuun **regedt32** ja napauta **OK**.

   Kuvan 6.2 mukainen rekisterieditori avautuu.



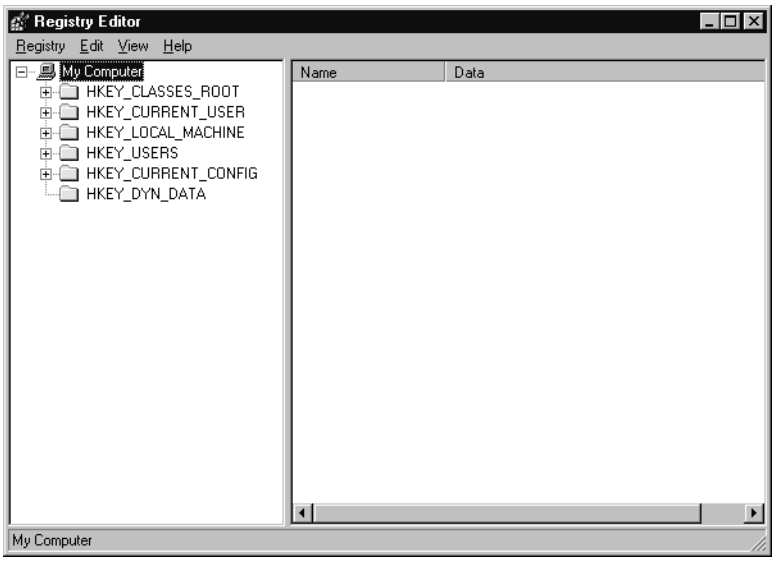
Kuva 6.2 Rekisterin tutkiminen RegEdt32:ta käyttäen

3. Valitse rekisterieditorin **Options**-valikosta **Read Only Mode**.

► **Rekisterin tutkiminen (Windows 95 ja Windows 98)**

1. Valitse **Käynnistä**-valikosta **Suorita**.
2. Kirjoita **Avaa**-ruutuun **regedit** ja napauta **OK**.

   Kuvan 6.3 mukainen rekisterieditori avautuu.



Kuva 6.3 Rekisterin tutkiminen RegEditiä käyttäen

Rekisterin rakenne

Rekisteri on hierarkkisesti järjestetty tietokanta. Hierarkian juuressa on joukko ennalta määrättyjä alihaaroja, jotka vastaavat järjestelmän organisaation yleisimpiä kategorioita. Huomaa, kuinka editorit esittävät alihaarat eri tavoin. **RegEdit** näyttää kaikki alihaarat samassa ikkunassa yhteisen juuren alapuolella. **RegEdit32** puo-lestaan näyttää jokaisen alihaaran omassa ikkunassaan.

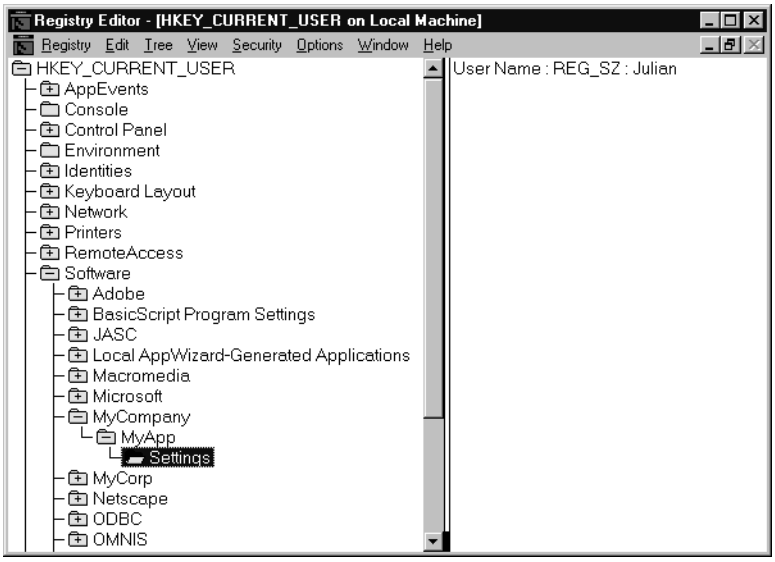
Taulukossa 6.3 on esitetty Windows-ympäristöjen viisi yleistä alihaaraa.

Taulukko 6.3 Windows 95, Windows 98 ja Windows NT:n alihaarat

Avaimen nimi	Tarkoitus
HKEY_CLASSES_ROOT	Sisältää ohjelmien kokoonpanotietoja. Sisältää tiedostojen tarkentimien ja sovellusten väliset yhteydet, vedä ja pudota -protokollat, tulostusasetukset ja COM-kokoonpanon tiedot.
HKEY_CURRENT_USER	Sisältää käyttäjästä riippuvia järjestelmän asetuksia. Alihaara luodaan käyttäjän kirjautuessa järjestelmään, jolloin siihen ladataan **HKEY_ USERS**-haaran käyttäjästä säilyttämät tiedot ja poistetaan käyttäjän kirjautuessa ulos.
HKEY_LOCAL_MACHINE	Sisältää tietokoneeseen asennettujen laitteiden ajureihin ja muuten järjestelmän laitteisiin liittyvät määritykset. Tähän osaan tehdyt merkinnät ovat kaikille tietokoneen käyttäjille yhteisiä.
HKEY_USERS	Sisältää kaikkien tietokoneeseen kirjautuvien käyttäjien tiedot. Sisältää sekä käyttäjäkohtaiset että yleiset tiedot. Yleiset asetukset koskevat kaikkia järjestelmän käyttäjiä.
HKEY_CURRENT_CONFIG	Sisältää tällä hetkellä tietokoneessa käytössä olevan laitteistoprofiilin tiedot.

Jos käytät Windows 95 tai Windows 98 -tietokonetta, näet lisäksi alihaaran **HKEY_DYN_DATA**, jota käytetään dynaamisen tiedon kuten suorituskykytilaston ja plug and play -informaation säilytykseen.

Alihaarassa on *avaimia*, jotka voivat toimia varastoina muille avaimille. Avaimet muistuttavat tiedostokansiota, koska niitä voidaan ketjuttaa ja koska niihin voidaan viitata samantyylisellä merkintätavalla. Kuvassa 6.4 nähdään tilanne, jossa RegEdt32:lla tutkitaan **HKEY_CURRENT_USER\Software\MyCompany \MyApp\Settings**-avainta.



Kuva 6.4 Rekisterin avainten tutkiminen

Avaimet voivat sisältää arvoja samoin kuin aliavaimiakin. Arvot ovat rekisterihierarkian päätesolmuja, joita käytetään rekisterin tietojen tallettamiseen. Kuvassa 6.4 nähdään *User Name* -arvo, joka on talletettu **Settings**-avaimeen.

Arvossa on kolme osaa — nimi, tietotyyppi ja arvo itse. Avain saattaa sisältää yhden nimeämättömän arvon, joka toimii avaimen oletusarvona. Selvyyden vuoksi rekisterin arvoille tulisi kuitenkin antaa nimet, jotka ilmaisevat selkeästi niiden merkityksen.

Windowsissa on määritelty erityinen joukko rekisteritietotyyppejä. Muutamia kaikkein yleisimpiä rekisteritietotyyppejä on esitetty taulukossa 6.4.

Taulukko 6.4 Rekisterin tietotyypit

Tietotyyppi	Käyttö
REG_DWORD	32-bittinen numero.
REG_BINARY	Mitä tahansa binaaritietoa.
REG_SZ	Null-arvoon päättyvä merkkijono.
REG_MULTI_SZ	Null-arvoon päättyvistä merkkijonoista koostuva taulukko, joka päättyy kahteen null-merkkiin.
REG_EXPAND_SZ	Null-arvoon päättyvä merkkijono, joka sisältää viittauksen ympäristömuuttujiin.

Täydellisen luettelon rekisteritietotyypeistä saat hakemalla hakusanalla "RegSetValueEx" — joka on Windows API -funktio, jota käytetään rekisterin arvon ja arvon tietotyypin asettamiseen — Visual C++ -ohjeessa.

MFC:n tuki rekisterille

MFC AppWizardin muodostama dokumentti/näkymä-sovellus luo rekisteriin seu-raavat merkinnät:

- Merkintä, joka auttaa Windowsin tiedostojen hallintaa yhdistämään sovelluksen dokumenttityypin sovellukseen. Tämä antaa käyttäjälle mahdollisuuden käynnistää sovelluksen kaksoisnapauttamalla dokumentin kuvaketta.
- Merkintä, joka määrittää sovelluksen *käyttäjäprofiilin* (user profile) sijainnin. Käyttäjäprofiilia käytetään varastoitaessa käyttäjän sovellukseen tekemiä asetuksia rekisteriin niin, että asetukset säilyvät käyttökerrasta toiseen.

Seuraava jakso kertoo, kuinka nämä rekisterimerkinnät tehdään ja kuinka niitä voi käyttää.

Dokumenttityyppien rekisteröinti

Rekisteröidäkseen sovelluksen dokumenttityypin Windowsin tiedostohallinnassa AppWizard lisää seuraavan koodirivin sovellusobjektin **InitInstance()**-funktioon:

```
RegisterShellFileTypes(TRUE);
```

CWinApp::RegisterShellFileTypes() käy läpi sovelluksen ylläpitämän dokumenttimallilistan ja lisää jokaista dokumenttimallia vastaavan yhdistämismerkin-nän rekisterin **HKEY_CLASSES_ROOT** alihaaraan.

RegisterShellFileTypes() lisää myös merkinnän, joka määrittää dokumenttityypin oletuskuvakkeen.

Koska dokumenttityypit tunnistetaan niiden tiedostotarkentimen perusteella, sinun täytyy muistaa määritellä dokumentin tiedostotarkennin AppWizardin vaiheen 4 Advanced Options -ruudussa. Jos päätteen määrittäminen ei onnistu, **RegisterShellFileTypes()**-funktiota ei kutsuta.

Kutsumalla **RegisterShellFileTypes()**-funktiota TRUE-parametrillä lisätään rekisteriin tarvittavat merkinnät **Print** ja **Print To** -komennoille, jotka antavat käyttäjälle mahdollisuuden tulostaa suoraan resurssienhallinnasta tai pudottamalla tiedoston kirjoittimen kuvakkeen päälle.

AppWizard luo rekisteritiedoston (.reg-tarkennin), jota asennusohjelman voi käyttää tehdessään asennusvaiheessa tarvittavia päivityksiä rekisteriin. Rekisteritiedoston käytöllä saavutetaan etua, koska sen avulla asennusohjelmaa voidaan ohjata poistamaan rekisterimerkinnät, jos ohjelma poistetaan järjestelmästä ja näin sovelluksia voidaan hallinnoida tehokkaammin. Jos jätät tiedostotarkentimen määrittelemättä, .reg-tiedostoa ei luoda.

Huomio Jos määrittelet sovelluksesi Compound Document Serveriksi, lisäkoodi OLE:en liittyvien rekisterimerkintöjen tekemiseksi liitetään sovelluksen **InitInstance()**-funktioon.

Sovelluksen käyttäjäprofiili

Sovelluksen käyttäjäprofiilitietojen sijainnin määrittämiseksi AppWizard lisää seuraavan rivin sovellusobjektin **InitInstance()**-funktioon:

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

CWinApp::SetRegistryKey()-funktiota kutsutaan antamalla parametriksi sovelluksen tietojen tallentamiseen käytetyn avaimen nimi. Tämä avain luodaan **HKEY_CURRENT_USER\Software**-avaimen alle. Oletuksena oleva avaimen nimi (**Local AppWizard-Generated Applications**) pitää korvata sopivammalla kaikkien sovelluksiesi käyttäjäprofiilien säilyttämiseen sopivalla nimellä. Tyypillisesti se on sovelluksen kehittäneen yrityksen nimi.

Huomio Jos poistat **InitInstance()**-funktiosta **SetRegistryKey()**-kutsun, sovelluskehys luo Windows-kansioon .ini-tiedoston ja käyttää sitä (käyttäjästä riippumattomien) profiilitietojen tallentamiseen..ini-tiedostot ovat 16-bittisen Windowsin jäänne ja niitä ei tulisi käyttää 32-bittisissä sovelluksissa.

Seuraavassa harjoituksessa määritellään MyApp-sovelluksen käyttäjäprofiilin tietojen tallentamiseen tarvittava rekisteriavain.

► **Käyttäjäprofiilin nimen asettaminen rekisteriin**

1. Avaa MyApp-projekti.
2. Avaa ClassViewissä **CMyAppApp**-luokan kuvake.
3. Aloita koodin editoiminen kaksoisnapauttamalla **InitInstance()**-jäsenfunktiota.
4. Etsi seuraava koodi, joka on noin rivillä 20:

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

5. Muuta riviä niin, että se näyttää seuraavalta:

```
SetRegistryKey(_T("MyCompany"));
```

6. Käännä ja käynnistä MyApp.

Varmista, että sovellus on luonut avaimen oikein, suorittamalla seuraavat toimenpiteet.

► **Käyttäjäprofiilin avaimen etsiminen**

1. Varmista, että olet kääntänyt ja käynnistänyt MyApp-sovelluksen ainakin kerran käyttäjäprofiilin avaimen määrittämisen jälkeen.
2. Sulje MyApp-sovellus. Valitse **Käynnistä**-valikosta **Suorita**.
3. Kirjoita **Avaa**-ruutuun **RegEdt32**, jos käytössä on Windows NT tai **RegEdit**, jos käytät Windows 95:ttä tai Windows 98:aa. Napauta **OK**.
4. Etsi **HKEY_CURRENT_USER**-alihaara.
5. Avaa **Software**-avain kaksoisnapauttamalla. Tarkista, että MyApp on lisännyt seuraavan avaimen:

```
HKEY_CURRENT_USER\Software\MyCompany\MyApp\Settings
```

Huomio **SetRegistryKey()**-funktio luo sovelluksen asetuksia varten rekisteriavaimen, joka on saman niminen kuin sovellus.

MFC-sovellukset kirjoittavat sovelluksen käyttäjäprofiiliin MFC:n profiilinkäsittelyfunktiota käyttäen. Näitä **CWinApp**-luokan jäsenfunktioita ovat:

- **WriteProfileString()**
- **GetProfileString()**
- **WriteProfileInt()**
- **GetProfileInt()**

Nämä funktiot tallentavat ja hakevat sekä merkkijono- että kokonaislukuarvoja rekisterissä olevasta sovelluksen profiiliavaimesta. Jokainen näistä funktioista ottaa kahtena ensimmäisenä parametrinä osan nimen (profiiliavaimen alihaara) ja arvon nimen. Jos osan tai arvon nimeä ei löydy, ne luovat sen. **WriteProfileString()** luo REG_SZ-arvon. **WriteProfileInt()** luo REG_DWORD-arvon. Kumpikin funktioista palauttaa BOOL-arvon, joka ilmaisee, onko kirjoitusoperaatio onnistunut. Molempien funktioiden kolmas parametri on talletettava tieto.

Seuraava rivi esimerkiksi:

```
WriteProfileString("Settings", "User Name", "Julian");
```

luo **HKEY_CURRENT_USER\Software\MyCompany\MyApp\Settings\User Name** arvon, johon yhdistetään teksti "Julian". Tämä arvo on kuvattuna kuvassa 6.4.

GetProfileString() ja **GetProfileInt()** palauttavat määrätyn arvon rekisteristä. Molempien funktioiden kolmas parametri on oletusarvo, joka palautetaan, mikäli pyydettyä arvoa ei löydy rekisteristä.

Sovelluksen käyttäjäprofiilin arvojen asettaminen

Tässä harjoituksessa opit, kuinka MFC:n profiilinkäsittelyfunktiota käytetään MyApp-sovelluksen tietojen tallettamiseen **HKEY_CURRENT_USER\Software\MyCompany\MyApp**-profiiliavaimeen, jonka loit edellisessä harjoituksessa. Tässä harjoituksessa käsitellään **Connect to Data Source** -dialogia, jonka olet tehnyt lukujen 4 ja 5 harjoituksissa.

Connect to Data Source -dialogissa on valintaruutu, jonka avulla käyttäjä voi valita, haluaako hän muodostaa yhteyden sovelluksen käynnistyessä.

► **Connect at Application Startup -toiminnon toteuttaminen**

1. Talleta Data Source Name, User ID, Access level ja **Attempt to connect at application startup** -valintaruudun viimeisin arvo rekisteriin. Nämä arvot tallentuvat käyttäjän painettua **Connect**-painiketta.
2. Alusta **Connect to Data Source** -dialogin kontrollit rekisteriin tallennettuja arvoja käyttäen.
3. Muuta **InitInstance()**-funktiota niin, että se tarkistaa **Attempt to connect at application startup** -valintaruudun tilan. Jos valintaruutu on valittu, anna käyttäjälle mahdollisuus yhteyden muodostamiseen avaamalla **Connect to Data Source** -dialogi.

► **Dialogin asetusten tallentaminen**

1. Etsi MyApp-projektista **CMyAppApp::OnDataConnect()**-funktio.

2. Etsi koodin haara, jossa lukee:

```
if(aCD.DoModal() == IDOK)
```

ja lisää seuraavat rivit **AfxMessageBox()**-funktiokutsun jälkeen:

```
WriteProfileString("Settings", "User ID", aCD.m_strUserID);
WriteProfileInt("Settings", "Access Level", aCD.m_nAccess);
WriteProfileInt("Settings", "Connect at Startup",
    aCD.m_bConnect ? 1 : 0);
WriteProfileString("Settings", "DSN", m_strDSN);
```

m_strDSN on **CMyAppApp**-luokan jäsenmuuttuja, jonka asettamisen suorittaa **CConnectDialog::OnOK()**-funktio.

► **Connect to Data Source -dialogin kontrollien alustaminen**

1. Poista seuraavat rivit **CMyAppApp::OnDataConnect()**-funktion alusta:

```
aCD.m_nAccess = 1;
aCD.m_bConnect = TRUE;
```

2. Korvaa poistetut rivit seuraavalla koodilla:

```
aCD.m_nAccess = GetProfileInt("Settings", "Access Level", 1);
aCD.m_strUserID = GetProfileString("Settings", "User ID", "");
aCD.m_bConnect = BOOL(GetProfileInt("Settings",
    "Connect at Startup", 0));
```

OnDataConnect()-funktion tulisi näyttää kokonaisuudessaan seuraavalta:

```
void CMyAppApp::OnDataConnect()
{
    CConnectDialog aCD;

    aCD.m_nAccess = GetProfileInt("Settings", "Access Level", 1);
    aCD.m_strUserID = GetProfileString("Settings", "User ID", "");
    aCD.m_bConnect = BOOL(GetProfileInt("Settings", "
        Connect at Startup", 0));

    if(aCD.DoModal() == IDOK)
    {
        CString strMessage;
        strMessage.Format("User %s logged in", aCD.m_strUserID);
        AfxMessageBox(strMessage);

        WriteProfileString("Settings", "User ID", aCD.m_strUserID);
        WriteProfileInt("Settings", "Access Level", aCD.m_nAccess);
        WriteProfileInt("Settings", "Connect at Startup",
```

```
                aCD.m_bConnect ? 1 : 0);
            WriteProfileString("Settings", "DSN", m_strDSN);
        }
    }
```

3. Etsi **CConnectDialog::OnInitDialog()**-funktio. Lisää seuraavassa esimerkissä lihavoituna kirjoitettu koodi:

```
BOOL CConnectDialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    m_lbDSN.AddString("Accounts");
    m_lbDSN.AddString("Admin");
    m_lbDSN.AddString("Management");
    CMyAppApp * pApp = dynamic_cast<CMyAppApp *>(AfxGetApp());
    ASSERT_VALID(pApp);
    CString strDSN = pApp->GetProfileString("Settings", "DSN", "");
    int index = m_lbDSN.FindStringExact(-1, strDSN);
    m_lbDSN.SetCurSel(index);
    OnUpdateUserid();
    return TRUE;
    // return TRUE unless you set the focus to a control
    // EXCEPTION: OCX Property Pages should return FALSE
}
```

Tämä koodi hakee tietolähteen nimen profiiliasetuksista ja käyttää sitä asettaessaan luettelon valintaa sopivaan kohtaan.

► **Connect to Data Source -dialogin avaaminen (jos tarpeen)**

1. Etsi **CMyAppApp::InitInstance()**-funktio. Lisää seuraava koodi funktion loppuun juuri return-komennon eteen:

```
if(GetProfileInt("Settings", "Connect at Startup", 0))
    OnDataConnect();
```

2. Käännä ja käynnistä MyApp-sovellus.
3. Valitse **Connect**-toiminto **Data**-valikosta, ja täytä **Connect to Data Source** -dialogi niin, että **Attempt to connect at application startup** -valintaruutu on valittuna.
4. Tallenna asetukset napauttamalla **Connect**.
5. Sulje ja käynnistä sovellus uudelleen ja varmista, että **Attempt to connect at application startup** -dialogi avautuu heti sovelluksen käynnistyttyä ja että talletetut tiedot on palautettu oikein. Voit myös tarkistaa rekisterieditorilla, että profiilin rekisteriarvot on luotu oikein.

Win32 API:n rekisterin käsittely

MFC:n rekisterifunktiot tekevät tietojen tallettamisen rekisteriin helpoksi, mutta ne eivät ole tarpeeksi joustavia kaikkiin tilanteisiin. MFC:n profiilifunktiot sallivat kirjoittamisen vain tietyn profiiliavaimen alle **HKEY_CURRENT_USER**-alihaa-rassa. Jos haluat tallettaa arvoja muualle rekisteriin, sinun täytyy käyttää Win32 API:n reksiterifunktiota. Saatat esimerkiksi haluta tallettaa kaikille tietokoneen käyttäjille yhteisiä sovelluksen asetuksia. Tämän tekemiseksi sinun täytyy luoda rekisteriavain **HKEY_LOCAL_MACHINE\SOFTWARE**-avaimen alle.

Lisäksi MFC-funktiot mahdollistavat vain merkkijono- ja kokonaislukuarvojen tallettamisen. Win32 API:n funktiot mahdollistavat kaikkien rekisteritietotyyppien käyttämisen.

Taulukossa 6.5 on lueteltu muutamia Win32 API:n rekisterifunktioita. Lisätietoja näiden funktioiden käytöstä saat hakemalla Visual C++:n ohjetiedostosta.

Taulukko 6.5 Win32 API rekisterifunktiot

Funktio	Toiminta
RegCreateKeyEx()	Luo avaimen rekisteriin. Jos avain on jo olemassa, funktio avaa sen.
RegOpenKeyEx()	Avaa olemassaolevan rekisteriavaimen.
RegCloseKey()	Sulkee **RegCreateKeyEx()** tai **RegOpenKeyEx()**-funktion avaaman rekisteriavaimen kahvan.
RegDeleteKey()	Poistaa Windows 95:ssä alihaaran ja kaikki sen alla olevat kohteet. Windows NT:ssä poistaa yksittäisen alihaaran, jonka alla ei voi olla muita kohteita.
RegSetValueEx()	Asettaa avoimen avaimen tietotyypin ja arvon.
RegQueryValueEx()	Hakee avoimen avaimen tietotyypin ja arvon.
RegDeleteValue()	Poistaa nimetyn arvon rekisteriavaimesta.

Oppitunnin yhteenveto

Windowsin rekisteri on keskitetty, hierarkkisesti järjestetty tietokanta, joka sisältää pysyvät käyttöjärjestelmän kokoonpanotiedot ja sovellusten asetukset. Sovelluksesi voi käyttää Win32 API -funktioita tietojen tallentamiseen ja tietojen noutamiseen rekisteristä. Voit muokata ja tutkia rekisteriin tallennettuja tietoja rekisterieditorin avulla.

Rekisteri on loogisesti jaettu viiteen ennalta määrättyyn alihaaraan, jotka vastaavat järjestelmän rakenteen yleisimpiä luokkia. Nämä luokat ovat:

- **HKEY_CLASSES_ROOT**
- **HKEY_CURRENT_USER**
- **HKEY_LOCAL_MACHINE**
- **HKEY_USERS**
- **HKEY_CURRENT CONFIG**

Alihaarat koostuvat avaimista, jotka toimivat säilöinä muille avaimille ja *arvoille* (values), jotka ovat tietojen tallettamiseen käytettyjä päätesolmuja. Arvolla on kolme osaa: nimi, tietotyyppi ja varsinainen arvo. Windows käyttää rekisterin arvoille erikoisesti tähän tarkoitukseen määriteltyä tietotyyppien joukkoa. Nämä tietotyypit ovat:

- **REG_DWORD**
- **REG_BINARY**
- **REG_SZ**
- **REG_MULTI_SZ**
- **REG_EXPAND_SZ**

MFC AppWizardin muodostama dokumentti/näkymä-sovellus kutsuu **CWinApp ::RegisterShellFileTypes()**-funktiota lisätäkseen rekisteriin merkinnän, joka yhdistää sovelluksen dokumenttityypin sovellukseen. **RegisterShellFileTypes()** lisää myös merkinnät, jotka yhdistävät oletuskuvakkeen tiedostoon ja merkinnät suoraan tiedostojen hallinnasta tapahtuvaa tulostusta varten.

Sovellus kutsuu **CWinApp::SetRegistryKey()**-funktiota lisätäkseen rekisteriin merkinnän, joka määrittää sovelluksen käyttäjäprofiilin (user profile) sijainnin. Käyttäjäprofiilia käytetään käyttäjän sovellukseen tekemien asetusten tallettamiseen niin, että ne voidaan säilyttää sovelluksen käyttökertojen välillä. Sinun tulisi muuttaa **SetRegistryKey()**-funktion määrittämä oletusnimi kaikkien sovelluksiesi käyttäjäprofiilien tallentamiseen sopivaksi nimeksi — esimerkiksi yhtiösi nimi käy hyvin. **SetRegistryKey()**-funktio luo profiiliavaimen alle avaimen, joka on samanniminen kuin sovelluksesi. Se toimii sovelluksesi tietojen perussijoitus-paikkana.

MFC-sovellus käsittelee sovelluksen käyttäjäprofiilia seuraavien profiilinkäsittelyfunktioiden avulla:

- **WriteProfileString()**
- **GetProfileString()**
- **WriteProfileInt()**
- **GetProfileInt()**

Näitä funktioita käytetään rekisterissä sovelluksen profiiliavaimessa olevien merkkijono- ja kokonaislukuarvojen tallettamiseen ja noutamiseen.

Jos MFC:n profiilinkäsittelyfunktiot eivät ole tarpeeksi joustavia, voit käyttää Win32 API:n rekisterifunktioita. Voit käyttää näitä funktiota esimerkiksi silloin, kun sinun täytyy kirjoittaa rekisteriin arvo, joka ei sijaitse määritellyn profiiliavaimen alla. Voit käyttää näitä funktioita myös rekisteriavainten poistamiseen ja lisäämiseen, sekä arvojen tallettamiseen ja hakemiseen. Yleisimmin käytetyt Win32 API:n reksiterifunktiot ovat:

- **RegCreateKeyEx()**
- **RegOpenKeyEx()**
- **RegCloseKey()**
- **RegDeleteKey()**
- **RegSetValueEx()**
- **RegQueryValueEx()**
- **RegDeleteValue()**

Laboratorio 6: Tallennettujen tietojen käsittely

Tässä laboratoriossa muutetaan STUpload-sovellusta niin, että se kykenee käsittelemään tietokoneen kiintolevyllä oleviin tiedostoihin talletettuja tiedostoja. Muokkaat sovellusta kahdessa vaiheessa. Ensin viimeistelet laboratoriossa 5 luomasi **CSTUploadDoc::LoadData()**-funktion. **CSTUploadDoc::OnDataImport()**-funktio kutsuu **LoadData()**-funktiota ladatakseen tekstitiedostossa olevat tiedot sovellukseen.

Toiseksi toteutat MFC:n standardin serialisoinnin STUpload-sovellukseen niin, että tekstitiedostosta tuodut tiedot voidaan talletaa STUpload-dokumenttitiedostoon.

Tietojen tuominen tekstitiedostosta

Laboratoriossa 5 loit väliaikaisen version **CSTUploadDoc::LoadData()**-funktiosta. Tällä hetkellä funktio yksinkertaisesti lisää muutamia kovakoodattuja osakkeiden hintatietoja **CStockDataList** objektiin **CSTUploadDoc::m_DocList**. Nyt viimeistellään **LoadData()**-funktio niin, että se lataa tietueet **m_DocList**-luetteloon **CSTUploadDoc::OnDataImport()**-funktion välittämästä **CStdioFile**-objektista. Ensin täytyy käyttää **CStdioFile**-objektia käyttäjän valitseman teks- titiedoston avaamiseen.

► **Tekstitiedoston avaaminen**

1. Muokkaa **CSTUploadDoc::OnDataImport()**-funktiota. Etsi koodiosuus, joka alkaa rivillä:

```
if(nID == IDOK)
```

2. Lisää **CStdioFile**-objektin määrittelyn perään seuraava koodi:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH6_01.cpp.)

```
CFileException fx;
if(!aFile.Open(aFileDialog.GetPathName(), CFile::modeRead |
      CFile::typeText, &fx))
{
      TCHAR buf[255];
      fx.GetErrorMessage(buf, 255);
      CString strPrompt(buf);
      AfxMessageBox(strPrompt);
      return;
}
```

Funktion tulisi näyttää nyt kokonaisuudessaan seuraavalta:

```
void CSTUploadDoc::OnDataImport()
{
    // String to customize File Dialog
    CString strFilter =
        Data Files (*.dat)|*.dat|All Files (*.*)|*.*||";

    CFileDialog aFileDialog(TRUE, NULL, NULL, OFN_HIDEREADONLY |
        OFN_OVERWRITEPROMPT,  strFilter);

    int nID = aFileDialog.DoModal();

    if(nID == IDOK)
    {
        CStdioFile aFile;

        CFileException fx;
        if(!aFile.Open(aFileDialog.GetPathName(),
            CFile::modeRead | CFile::typeText, &fx))
        {
            TCHAR buf[255];
            fx.GetErrorMessage(buf,  255);
            CString strPrompt(buf);
            AfxMessageBox(strPrompt);
            return;
        }
        LoadData(aFile);
    }
}
```

Ennen kuin vanha **LoadData()**-funktio korvataan uudella tehdään muutamia muutoksia, jotta uusi funktio voisi käyttää **Conflicting Records**-dialogia. Ensin sinun täytyy alustaa sovelluksesi rich edit -kontrollit kutsumalla **AfxInitRichEdit()**-funktiota.

► **Rich edit -kontrollien käyttöönotto**

1. Muokkaa **CSTUploadApp::InitInstance()**-funktiota. Lisää seuraavat rivit funktion loppuosaan ennen return-komentoa:

   ```
   AfxInitRichEdit();
   ```

2. Lisää DDX CConflictDialog::m_REditText jäsenmuuttuja. Tätä muuttujaa tullaan käyttämään CString-muuttujana, joka asettaa rich edit -kontrollissa näkyvän tekstin.

► **CConflictDialog::m_REditText-muuttujan lisääminen**

1. Avaa ClassWizard.
2. Valitse **Member Variables** -välilehti.
3. Lisää **CString**-muuttuja m_REditText **CConflictDialog**-luokkaan, joka on yhdistetty **IDC_DUPL_RICHEDIT** resurssitunnisteeseen.

Nyt voidaan tehdä uusi **LoadData()**-funktio.

► **LoadData()-funktion korvaaminen**

1. Lisää STUploadDoc.cpp-tiedoston alkuun muiden #include-lauseiden joukkoon seuraava rivi:

```
#include "ConflictDialog.h"
```

2. Etsi **CSTUploadDoc::LoadData()**-funktio. Poista koko funktio ja korvaa se seuraavalla koodilla:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH6_02.cpp.)

```
BOOL CSTUploadDoc::LoadData(CStdioFile &infile)
{
    // Check for NULL
    ASSERT(infile.m_hFile != NULL);

    // Hold data in temporary list of CStockData objects,
    // which we assign to CSTUploadDoc::m_DocList only
    // when we are sure load has been completed successfully
    CStockDataList TempList;
    // Additions are cumulative, so we need to copy in existing data
    TempList.AddHead(&m_DocList);

    // Line buffer
    CString strTemp;

    // Today's date
    COleDateTime Today = COleDateTime::GetCurrentTime();
    COleDateTime FileDate;
    CString strFileHeader;

    int addedCtr = 0;    // Count added items
    int discardedCtr = 0;      // Count discarded items

    BOOL bFirstLine = TRUE;
```

```
while(infile.ReadString(strTemp))
{
      BOOL bValidDate = FALSE;
      CString strFund;
      CString strDate;

      // Exclude blank lines
      if(strTemp.GetLength() == 0) continue;

      if(bFirstLine)
      {
            // Get Header information
            strFileHeader = strTemp.Left(18);
            strFileHeader.TrimRight();
            strDate = strTemp.Mid(18, 10);
      }
      else
      {
            strFund = strTemp.Left(8);
            strFund.TrimRight();
            strDate = strTemp.Mid(8, 10);
      }

      int nYear = atoi(strDate.Right(4));
      int nMonth = atoi(strDate.Left(2));
      int nDay = atoi(strDate.Mid(3, 2));

      COleDateTime aDate(nYear, nMonth, nDay, 0, 0, 0);

if(aDate.GetStatus() != COleDateTime::valid)
      {
            if(bFirstLine)
            {
                  // Cannot read file date - assume invalid
                  AfxMessageBox("Invalid File Format");
                  return FALSE;
            }
            else
            {
                  // Cannot read record date - discard line
                  discardedCtr++;
                  continue;
            }
      }
```

```
if(bFirstLine)
{
    // Get file date - loop back to top
    FileDate = aDate;
    bFirstLine = FALSE;
    continue;
}

double dPrice = atof(strTemp.Mid(19));

// Make a CStockData object and add it
// to our temporary array
CStockData aStData(strFund, aDate, dPrice);
CStockDataList::errorstatus err;
POSITION CurPos = TempList.AddSorted(aStData, err);

switch(err)
{
    // Discard identical entry
    case CStockDataList::duplicate_entry :

        discardedCtr ++ ;
        continue;

    // Same record, different price value
    case CStockDataList::conflicting_entry :
    {
        // Query if user wants to discard duplicate,
        // replace, or abort
        CConflictDialog aDialog;

        // Construct text to appear in rich edit
        // control
        CString strText = "Existing entry:\n\n";

        CStockData SDTemp = TempList.GetAt(CurPos);

        strText += SDTemp.GetAsString();
        strText += "\n\nReplacement entry:\n\n";
        strText += aStData.GetAsString();

        // Assign text to control variable
        aDialog.m_REditText = strText;
```

```
                    switch(aDialog.DoModal())
                    {
                        case IDABORT : // Abandon
                        return FALSE;

                        case IDCANCEL : // Discard new record
                        discardedCtr++ ;
                        continue;

                        case IDOK : // Replace existing record
                        TempList.SetAt(CurPos, aStData);
                    }
                }

            default:  // Ok
                addedCtr++ ;
        }
    }

    // If we got this far then this is a valid record

    CString strPrompt;
    strPrompt.Format(
        "Import of file %s complete:\nRecords loaded: %d \
        \nRecords discarded: %d  \
        \n\nHit OK to load data into document.",
        strFileHeader, addedCtr, discardedCtr);

    if(AfxMessageBox(strPrompt, MB_OKCANCEL) == IDOK)
    {
        // Update document data
        m_DocList.RemoveAll();
        m_DocList.AddHead(&TempList);

        // Update fund view
        CMainFrame * pWnd =
            dynamic_cast<CMainFrame *> (AfxGetMainWnd());

        if(pWnd)
        {
            pWnd->UpdateFundList(m_DocList);
            // Show fund window after loading new funds
            pWnd->SetFundsVisible(TRUE);
        }

        return TRUE;
    }
    else
        return FALSE;
    }
```

3. Tutki koodia ja varmista, että ymmärrät, kuinka:

- Väliaiakaista listaa käytetään ladattujen tietojen säilyttämiseen niin, että sovel-luksen tietoja ei muuteta ennen kuin lataaminen on suoritettu onnistuneesti loppuun ja käyttäjä on hyväksynyt tietojen tuomisen.
- Rutiini jakaantuu header ja data -rivien välillä.
- Header-rivi tulkitaan niin, että varmistutaan ladattavan tietotyypin oikeellisuudesta.
- **CString**-funktiota käytetään tietojen poimimiseen datariviltä.
- Rutiini käsittelee virheelliset ja useammin kuin kerran esiintyvät rivit, sekä ris-tiriitaiset merkinnät (sama osake ja päivämäärä, eri hinta). **CStockDataList ::AddSorted()**-funktiota kannattaa ehkä tarkastella lähemmin.

4. Käännä ja käynnistä STUpload. Kokeile **Data**-valikon **Import**-toimintoa ja lataa Ch6Test.dat-tiedosto oheisrompun kansiosta ..\Chapter 6\Data. Varmista, että lataaminen tapahtuu odotetulla tavalla. Sulje sovellus ja käynnistä se uudelleen. Lataa tällä kertaa conflict.dat-tiedosto ..\Chapter 6\Data-kansiosta. Tässä tiedostossa ovat virheelliset tiedot, joiden avulla voit kokeilla päällekkäisyyksien tarkistusrutiinia.

STUploadin serialisointi

STUpload-sovelluksen tiedot ovat yhdessä **CStockDataList**-objektissa, joka on kokoelma **CStockData**-objekteja. **CStockData**-objekti kapseloi osakkeen nimen, päivämäärän ja hinnan. Dokumentti sisältää myös CString-muuttujan, johon tallennetaan valittuna olevan osakkeen nimi. Tämä muuttuja täytyy myös serialisoida niin, että tallennettaessa valittuna ollut osake voidaan pitää edelleen valittuna, kun tiedot avataan uudelleen.

STUpload-sovelluksen tietojen serialisoimiseksi täytyy:

- Tehdä **CStockData**-luokasta serialisoituvat.
- Tehdä **SerializeElements()**-funktiomalli **CStockData**-elementtityypille (Tämä on välttämätöntä koska **CStockData** sisältää **CString**-tyyppisiä jäseniä.)
- Toteuttaa **CSTUploadDoc::Serialize()**-funktio.
- Toteuttaa **CSTUploadDoc::DeleteContents()**-funktio, joka tyhjentää dokumentin objekteissa olevat tiedot ennen uudelleenkäyttöä.
- Lisätä funktion **CDocument::SetModifiedFlag()** kutsu jokaiseen kohtaan, jossa tietoja muutetaan, jolloin sovelluskehys pystyy huomauttamaan käyttäjälle tietojen tallentamisesta ennen sovelluksen sulkemista.

CStockData-luokan serialisointi

CStockData-luokka, jota käytämme, on periytetty suoraan **CObject**-luokasta, ja sillä on oletusmuodostin. Sinun täytyy lisätä serialisointimakrot ja **Serialize()**-funktio.

► **Serialisointimakrojen lisääminen**

1. Avaa StockData.h-tiedosto **CStockData**-luokan määrittelyn muokkaamista varten.
2. Lisää seuraava rivi luokan määrittelyn **public**-osaan:

```
DECLARE_SERIAL(CStockData)
```

3. Avaa StockData.cpp-tiedosto. Lisää seuraava rivi tiedoston alkuun esikääntäjän komentojen perään:

```
IMPLEMENT_SERIAL(CStockData, CObject, 1)
```

► **Serialize()-funktion lisääminen**

1. Siirry takaisin StockData.h-tiedostoon.
2. Lisää seuraava määrittely **CStockData**-luokan määrittelyn **public**-osaan:

```
virtual void Serialize(CArchive& ar);
```

3. Siirry takaisin StockData.cpp-tiedostoon. Lisää tiedoston loppuun seuraava koodi:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH6_03.cpp.)

```
void CStockData::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strFund;
        ar << m_date;
        ar << m_dblPrice;
    }
    else
    {
        ar >> m_strFund;
        ar >> m_date;
        ar >> m_dblPrice;
    }
}
```

SerializeElements()-funktion ylikirjoitus

Seuraavaksi tehdään funktiomalli **SerializeElements()** elementtityypille **CStockData**. Funktio käy yksinkertaisesti läpi kokoelman tiedot ja kutsuu jokaisen **CStockData**-objektin **Serialize()**-funktiota.

► **SerializeElements()-funktion ylikuormitus**

1. Avaa StockDataList.h-tiedosto.
2. Lisää tiedoston loppuun, ennen #endif komentoa ja **CStockDataList**-luokan määrittelyn *jälkeen* seuraavat rivit:

```
template <> void AFXAPI SerializeElements <CStockData>
    (CArchive& ar, CStockData* pNewSD, int nCount);
```

3. Avaa StockDataList.cpp-tiedosto.
4. Lisää tiedoston loppuun seuraava koodi:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH6_04.cpp.)

```
template <> void AFXAPI SerializeElements <CStockData>
    (CArchive& ar, CStockData* pNewSD, int nCount)
{
    for (int i = 0; i < nCount; i++, pNewSD++)
    {
        // Serialize each CStockData object
        pNewSD->Serialize(ar);
    }
}
```

CSTUploadDoc::Serialize()-funktion toteuttaminen

Tässä vaiheessa olemme valmiit toteuttamaan dokumentin serialisointikoodin.

► **CSTUploadDoc::Serialize()-funktion toteutus**

1. Etsi **CSTUploadDoc::Serialize()**-funktio.
2. Korvaa funktio seuraavalla versiolla:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH6_05.cpp.)

```
void CSTUploadDoc::Serialize(CArchive& ar)
{
    m_DocList.Serialize(ar);

    if (ar.IsStoring())
    {
        ar << m_strCurrentFund;
    }
```

```
        else
        {
            ar >> m_strCurrentFund;

            // Update Select Fund window
            CMainFrame* pWnd = dynamic_cast<CMainFrame *>
                (AfxGetMainWnd());

            if(pWnd)
            // Will fail if running from icon or from
            // command line with file name argument
            {
                // Update and show fund window
                pWnd->UpdateFundList(m_DocList, m_strCurrentFund);
                pWnd->SetFundsVisible(TRUE);
            }
        }
    }
```

Kaikkien osakkeiden tiedot serialisoidaan yhdellä **CStockDataList::Serialize()**-kutsulla. **CSTUploadDoc::m_strCurrentFund**-muuttuja serialisoidaan myös.

Select Fund -ikkunan avaaminen

Huomaa koodi, joka avaa osakeikkunan, kun dokumentti ladataan. Ikkunan avaaminen ei ole mahdollista, kun sovellus käynnistetään kaksoisnapauttamalla dokumenttitiedostoa, koska pääikkunan osoitinta ei ole saatavilla. Voit kuitenkin ohjata sovelluksen avaamaan osakeikkunan sen jälkeen, kun pääikkuna on luotu, jos siinä vaiheessa havaitaan, että dokumentti on jo ladattu.

► **Select Fund -ikkunan avaaminen ohjelman käynnistyessä**

1. Etsi **CSTUploadApp::InitInstance()**-funktio.
2. Lisää seuraava koodi funktion loppuun ennen return-komentoa:

   (Tämä koodi on asennettu oheisrompulta tiedostoon CH6_06.cpp.)

```
CMainFrame * pFrameWnd =
    dynamic_cast<CMainFrame *> (m_pMainWnd);

ASSERT_VALID(pFrameWnd);

CSTUploadDoc * pDoc =
    dynamic_cast<CSTUploadDoc *>(pFrameWnd->GetActiveDocument());
```

```
ASSERT_VALID(pDoc);

if(pDoc->GetDocList().GetCount() > 0)
// Non-empty document at main window creation time means we are
// running from icon or from command line with file name argument
{
      pFrameWnd->UpdateFundList(pDoc->GetDocList(),
            pDoc->GetCurrentFund());
      pFrameWnd->SetFundsVisible(TRUE);
}
```

DeleteContents()-funktion toteutus

Koska STUpload on SDI-sovellus, kaikki dokumenttiobjektiin tallennetut tiedot täytyy poistaa **DeleteContents()**-funktiolla.

► **CSTUploadDoc::DeleteContents()-funktion tekeminen**

1. Ylikuormita ClassWizardia käyttämällä **CSTUploadDoc**-luokan **DeleteContents()**-funktio.
2. Muokkaa funktion koodia. Korvaa //TODO-kommentti seuraavalla koodilla:

   (Tämä koodi on asennettu oheisrompulta tiedostoon CH6_07.cpp.)

```
m_DocList.RemoveAll();

CMainFrame * pWnd =
      dynamic_cast<CMainFrame *> (AfxGetMainWnd());

if(pWnd)
{
      pWnd->UpdateFundList(m_DocList);
      // No funds on file, so hide fund window
      pWnd->SetFundsVisible(FALSE);
      // And reset current fund value
      SetCurrentFund("");
}
```

Dokumentin muutoksista ilmoittaminen

Lopuksi täytyy vielä huolehtia siitä, että **CDocument::SetModifiedFlag()**-kutsu suoritetaan aina sovelluksen tietojen muuttuessa. STUpload-sovelluksen tietoja muutetaan kahdessa kohdassa:

- **CSTUploadDoc::OnDataImport()**-funktiossa sen jälkeen, kun **LoadData()**-funktio on palauttanut arvon TRUE, merkiksi siitä, että tietojen tuominen on onnistunut.
- **CFundDialog::OnSelchangeFundlist()**-funktiossa, kun käyttäjä vaihtaa valittua osaketta.

► **CSTUploadDoc::OnDataImport()-funktion muokkaaminen**

1. Etsi **OnDataImport()**-funktiosta rivi, jossa lukee:

```
LoadData(aFile);
```

2. Korvaa tämä rivi seuraavalla koodilla:

```
if(LoadData(aFile))
{
    SetModifiedFlag();
    UpdateAllViews(NULL);
}
```

► **CFundDialog::OnSelchangeFundlist()-funktion muokkaaminen**

Etsi **CFundDialog::OnSelchangeFundlist()**-funktio. Lisää funktion loppuun ennen viimeistä aaltosulkua seuraava rivi:

```
pDoc->SetModifiedFlag();
```

Serialisoinnin testaaminen

► **STUpload-sovelluksen kääntäminen ja testaus**

1. Käynnistä sovellus, lataa sitten **Data**-valikon **Import**-toiminnolla tiedosto ..\Chapter 6\Data\Test.dat oheisrompulta.
2. Valitse katseltava osake ja sulje sitten sovellus. Tallenna dokumentti kehotettaessa nimellä **MyFile.stu**.
3. Käynnistä sovellus uudelleen. Valitse **File**-valikosta **Open**, ja avaa MyFile.stu-tiedosto.
4. Valitse tutkittavaksi uusi tiedosto ja sulje sovellus tallentaen samalla tehdyt muutokset.
5. Etsi Resurssienhallinnan avulla MyFile.stu-tiedosto, ja käynnistä STUpload sovellus kaksoisnapauttamalla tiedoston kuvaketta.
6. Varmista, että tiedosto avautui oikein.

Kertaus

1. Kuinka avaat tiedoston tekstitilassa?

2. Millaisen poikkeuksen nostaa **CFile::Open()**-funktio?

3. Mitä vaiheita luokkaa serialisoitaessa täytyy suorittaa?

4. Mitä **SerializeElements()**-funktion perustoteutus tekee?

5. Mitä rekisteriavainta tulisi käyttää sovelluksen kaikille tietokonelaitteen käyttäjille yhteisten asetusten tallettamiseen?

6. Minne MFC:n profiilinkäsittelyluokat **WriteProfileString()** ja **WriteProfileInt()** tallentavat profiiliasetukset?