L U K U 3

Sovellusten ohjelmointi Microsoft Foundation Classes -luokkien avulla

Tässä luvussa

Luvussa 2 kerrottiin, kuinka Microsoft Foundation Classes (MFC) AppWizardia käytettään MFC-luokkiin perustuvan projektin luomiseen. Tässä luvussa tutustutaan yksityiskohtaisemmin MFC-luokkiin ja niiden rooliin Windows-ohjelmoinnissa. Opit kuinka MFC-sovelluskehyksessä toteutetaan Windows-sovelluksen perusosat ja tutustut sen tarjoamaan arkkitehtuurin, joka mahdollistaa sovelluksen tietojen käsittelyn, näyttämisen ja varastoinnin.

Ennen kuin aloitat

Ennen tämän luvun aloittamista sinun tulisi lukea luku 2, *Visual C++ -ohjelmointiympäristö* ja tehdä siihen liittyvät harjoitukset.

Oppitunti 1: Yleiskatsaus MFC:hen

MFC-kirjasto on kokoelma C++-luokkia ja globaaleja funktioita, jotka on suunniteltu nopeuttamaan Microsoft Windows-pohjaisten sovellusten kehitystyötä. MFC tarjoaa monia etuja kaiken tasoisille C++-ohjelmoijille, aloittelijoista kokeneisiin ammattilaisiin. Se yksinkertaistaa Windows-ohjelmointia, lyhentää tuotantoaikaa ja tekee koodista helpommin siirrettävää rajoittamatta kuitenkaan ohjelmoinnin vapautta ja joustavuutta. MFC tarjoaa helpon lähestymistavan vaikeasti ohjelmoitaviin tekniikoihin, kuten ActiveX- ja Internet-ohjelmointiin. MFC helpottaa käyttöliittymä komponenttien, kuten ominaisuusikkunoiden, tulostuksen esikatselun, pikavalikoiden ja kelluvien työkalupalkkien sekä työkaluvihjeiden ohjelmoimista.

Tässä luvussa opit kuinka MFC-kirjasto on järjestetty ja kuinka se kapseloi Microsoft Win32 API:n, joka on ohjelmoijan käyttöliittymänä Windows-käyttöjärjestelmän toimintoihin toimiva matalan tason ohjelma. Opit myös joitain tärkeimmistä ympäristöön liittyvistä asioista, joita MFC-kirjastoja käytettäessä tulee huomioida.

Tämän oppitunnin jälkeen:

- Osaat kertoa, kuinka MFC-kirjasto rakentuu ja kuinka se on yhteydessä Win32 API:in.
- Tiedät tärkeimmät asiat, jotka tulee huomioida tehtäessä ohjelmia eri Win32-ympäristöihin: Microsoft Windows NT:hen, Microsoft Windows 95:een ja Microsoft Windows 98:aan.
- Tunnet MFC-luokkien yleisen luokituksen ja niiden hierarkkisen rakenteen.
- Tiedä,t milloin käytetään MFC:n laajennettuja DLL:iä.

Oppitunnin arvioitu kesto: 40 minuuttia

MFC ja Win32 API

Termiä *Win32* käytetään kuvaamaan sovellusohjelmointiliittymää (Application Programming Interface, API), joka on yhteinen kaikille Microsoftin 32-bittisille ympäristöille. Tällaisia ympäristöjä ovat tällä hetkellä Windows 95, Windows 98, Windows NT ja Microsoft Windows CE. (Huomio. Tämä kirja ei käsittele Windows CE:n erikoispiirteitä.) Win32 API koostuu joukosta funktioita, rakenteita, sanomia, makroja ja rajapintoja, jotka muodostavat yhdenmukaisen ohjelmointirajapinnan, joka on riippumaton siitä, mihin Win32-ympäristöön sovellusta ollaan tekemässä. Taulukossa 3.1 on lueteltu joitain Win32 API:n tarjoamia palveluja.

Taulukkoa 3.1 Win32 API:n palvelut

Win32 API -palvelu	Kuvaus
Ikkunoiden hallinta (Window Management)	Sisältää toiminnot, joiden avulla käyttöliittymä voidaan luoda ja joilla sitä voidaan hallita.
Ikkunakontrollit (Window Controls)	Sisältää joukon yleisimpiä käyttöliittymäkontrolleja. Yleisten komponenttien käyttäminen auttaa pitämään sovelluksen käyttöliittymän yhdemukaisena ympäristön ja muiden sovellusten kanssa. Se myös vähentää huomattavasti ohjelmointityötä.
Komentotulkkipalvelu Features)	Palvelun avulla voidaan käsitellä järjestelmän objek-(Shell teja ja resursseja kuten tiedostoja, tallennusvälineitä, tulostimia ja verkkoresursseja.
Grafiikka-rajapinta (Graphics Device Interface)	Sisältää funktiot ja muut rakenteet, joita käytetään grafiikan tulostamiseen näytöille, tulostimille ja muille grafiikkalaitteille.
Järjestelmäpalvelut (System Services)	Tarjoaa pääsyn tietokoneen resursseihin alla olevan käyttöjärjestelmän toimintojen kautta.

Koska Win32 API koostuu C-funktioista, Visual C++:n käyttäjät voivat helposti linkittää header-tiedostot ja käyttää Win32 APIn funktioita koodissaan. Ennen kuin MFC oli saatavilla, Windows-sovellukset piti ohjelmoida käyttäen pelkkää Win32 API:a — mikä on hyvin aikaa vievää. Kokeneet Win32-ohjelmoijat käyttivät poikkeuksetta koodia uudelleen ja kehittivät kirjastoja no-peuttaakseen usein toistuvien perus-Windows-ohjelmointitehtävien suorittamista. MFC formalisoi tämän prosessin ja saattaa syntyneen uudelleen käytettävän koodin kaikkien C++-ohjelmoijien hyödynnettäväksi.

MFC kehitettiin nopeuttamaan ja yksinkertaistamaan Windows-sovellusten ohjel-mointia. Se sisältää joukon C++-luokkia, jotka kapseloivat Win32 API:n tärkeimmät toiminnot. C++-ohjelmoijana tiedät, kuinka hyvän luokkien suunnittelun avulla kätketään ohjelman monimutkaiset ominaisuudet selkeän ja yksinkertaisen rajapinnan taakse.

Esimerkiksi Win32:n ikkuna käsite on kapseloitu MFC-luokkaan **CWnd**. Toisin sanoen **CWnd**-niminen C++-luokka koteloi HWND-kahvan (HWND on Win32:ssa määritelty tietotyyppi, joka edustaa Win32-ikkunaa). Kapselointi tarkoittaa sitä, että **CWnd**-luokalla on HWND-tyyppinen jäsenmuuttuja ja luokan jäsenfunktiot kapseloivat ne Win32-funktioiden kutsut, jotka tarvitsevat HWND:n parametrikseen. Katsotaan esimerkiksi seuraavaa Win32-funktiota:

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

joka on kapseloitu MFC funktioon:

```
BOOL CWnd::ShowWindow(int nCmdShow);
```

MFC-luokkien funktioilla on yleensä sama nimi kuin Win32-funktioilla, jotka ne kapseloivat.

MFC kätkee joitakin Windows-ohjelmoinnin alemman tason toimintoja kapseloimalla, mutta kyseiset toiminnot ovat aina tarvittaessa käsiteltävissä myös suoraan. Kapselointi suojaa myös osoittimia, kuten ikkunoiden ja tiedostojen kahvoja, saamasta virheellisiä arvoja sekä auttaa estämään operaatioiden vääriä toimintoja virheellisten osoittimien käytön seurauksena.

Milloin käytetään MFC:tä ja milloin Win32 API:a

MFC:tä tulisi käyttää aina, jos ohjelmoidaan jotain muuta kuin aivan yksinkertaisia sovelluksia. Koska ohjelmointi on MFC:tä käyttäen yksinkertaisempaa ja koska sen avulla koodia voidaan helposti uudelleen käyttää, MFC:n käyttämisestä saatavat edut ovat paljon suuremmat kuin sen käytöstä aiheutuvat haitat nopeuden ja koon suhteen. Sovellukset, joiden käyttöliittymä perustuu komentokehotteen käyttöön tai joilla ei ole käyttöliittymää lainkaan, on todennäköisesti yhtä helppoa toteuttaa ilman MFC:tä. Jopa näissä sovelluksissa saatat huomata MFC:n käyttökelpoiseksi, koska se sisältää joukon apuluokkia kuten merkkijonoluokat ja yleiset kokoelmaluokat.

MFC kapseloi suurimman osan Win32 API:sta, muttei kaikkia osia. Vaikka voit suorittaa suurimman osan Windows-ohjelmointitehtävistä käyttämällä MFC-luokkia, täytyy sinun silti joskus kutsua Win32 API -funktioita suoraan. Tämä on tavallisesti tarpeen, kun haluat päästä käsiksi järjestelmän matalan tason toimintoihin. Esimerkiksi, jos sinun pitää tehdä sovellus, joka käsittelee verkon käyttäjätilejä, joudut käyttämään Win32 APIn verkkotoimintoja, koska MFC kirjastossa ei ole luokkia, joiden avulla voisit käsitellä Windows NT:n verkonhallintatoimintoja.

Win32-alustasta huomioitavia asioita

Kuten aiemmin mainittiin, Windows 95, Windows 98 ja Windows NT ovat kaikki 32-bittisiä käyttöjärjestelmiä, jotka käyttävät kaikki samaa Win32 API:a. Jokainen käyttöjärjestelmistä on suunniteltu eri tarkoitukseen ja siksi niillä on yhteisten, kaikille samojen Win32-ominaisuuksien lisäksi omat erikoisuutensa. Nämä erot tekevät käyttöjärjestelmäkohtaiset Win32 API -funktiot välttämättömiksi.

Saadaksesi lisätietoja siitä, mitkä funktiot ovat millekin käyttöjärjestelmälle tarkoitettuja, hae Visual C++ -ohjeesta lauseella "Differences in Win32 API implementations." Tällä haulla löydät hyödyllisen (englanninkielisen) artikkelin, jossa kerrotaan yksityiskohtaisesti erot, jotka tulee huomioida käytettäessä Win32 APIa Windows NT, Windows 95 ja Windows 98 -ympäristöissä.

Jos ohjelmaa suorittavan käyttöjärjestelmän selvittäminen on tarpeen, esimerkiksi, jos ohjelman täytyy pystyä päättämään, mikä käyttöjärjestelmäkohtaisista ohjelman haaroista suoritetaan, voidaan käyttää API-funktiota **GetVersionEx()** seuraavassa koodinäytteessä esitetyllä tavalla.

```
OSVERSIONINFO vinfo;
vinfo.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);

::GetVersionEx(&vinfo);

switch(vinfo.dwPlatformId)
{
case  VER_PLATFORM_WIN32_WINDOWS  :
      // Windows 95 / 98 specific code

case  VER_PLATFORM_WIN32_NT  :
      // Windows NT specific code
}
```

MFC:n abstraktion korkea taso säästää sinut monilta alustaa koskevilta pohdinnoilta. Suurin osa MFC-koodista, jonka kirjoitat, voidaan ajaa missä tahansa Win32-ympäristössä ongelmitta, mutta varmistaaksesi, että mitään ongelmia ei esiinny, sinun tulee olla tietoinen muutamista järjestelmäkohtaisista asioista.

Unicodeen liittyvät kysymykset

16-bittisissä järjestelmissä, kuten MS-DOSissa ja Microsoft Windows 3.1:ssa, ANSI-merkistö käyttää yhtä tavua kirjaimen esittämiseen. Koska yhtä tavua käyttäen voidaan esittää vain 256 erimerkkiä, ANSI-merkistö on riittämätön kielissä, joissa on suuri joukko kirjainmerkkejä, kuten kiinassa. Unicode ratkaisee tämän ongelman käyttämällä kahta tavua jokaisen merkin esittämiseen. 16-bittinen Unicode-merkistö voi käsittää 65 536 eri merkkiä ja näin siitä on tullut kansainvälinen merkistöstandardi.

Windows NT tukee sekä Unicode- että ANSI-merkkijonoja. Kaikki Windows NT:n sisäiset merkkijonot, mukaan lukien Windows NTFS -tiedostojärjestelmän (NTFS) nimet, ovat Unicode-merkkijonoja. Win32 API:ssa on käytettävissä valinnaisesti määriteltyjä tietotyyppejä ja funktioiden yleisiä versioita, joiden toteutus muuttuu sen mukaan, onko _UNICODE-symboli määritelty vai ei. Esimerkiksi, kun _UNICODE-symboli on määritelty, TCHAR-tietotyyppi on määritelty *wchar_t* (16-bittinen merkkityyppi) -tyyppiseksi; muutoin määrittely on *char*, eli 8-bittinen merkkityyppi.

Windows 95 ja Windows 98 tukevat vain ANSI-merkkijonoja — eivät Unicode-merkkijonoja. Varmistaaksesi, että sovelluksesi merkkijonotieto on siirrettävissä Win32-järjestelmästä toiseen, sinun tulisi aina käyttää yleisiä tietotyyppejä ja funktioita.

MFC:n merkkijonoluokka **CString** perustuu TCHAR-tietotyyppiin. Sen muodostimet, liittämis- ja vertailuoperaattorit tukevat kaikki Unicodea. Tämä tarkoittaa sitä, että voit käyttää **CString**-luokkaa läpinäkyvästi tietäen, että se toteuttaa Unicode- tai ANSI-merkkijonot riippuen käännösympäristöstä. Sinun tulee kuitenkin olla varovainen hyödyntäessäsi **CString**-luokan järjestelmä-

riippumattomuutta omassa koodissasi. Jotta koodisi olisi paremmin Unicode yhteensopivaa, sinun täytyy lisäksi huomioida seuraavat asiat:

- Tee vakiomerkkijonojen ehdollinen siirto Unicode-muotoon mahdolliseksi käyttämällä *_T*-makroa, eli kirjoita *"MyString"* sijasta *_T(MyString)*.
- Kun siirrät merkkijonoja funktioille, kiinnitä huomiota siihen, vaativatko funktion argumentit pituuden merkkeinä vai bitteinä. Ero on merkittävä, jos käytät Unicodea.
- Käytä C:n run-time-merkkijonojen käsittelyfunktioiden siirrettäviä versioita. Kirjoita esimerkiksi *"strlen"* sijaan *"_tcslen"* ja *"strcat"* sijaan *"tcscat"*.
- Käytä seuraavia tietotyyppejä merkeille ja merkkiosoittimille:
  - *TCHAR* tyypin *char* sijasta.
  - *LPTSTR* tyypin *char** sijasta
  - *LPCTSTR* tyypin *const char** sijasta. **CString**-luokassa on käytettävissä operaattori LPCTSTR, jolla voidaan tehdä muunnos tyyppien **CString** ja LPCTSTR välillä.

Tiedostojärjestelmään liittyvät kysymykset

NTFS on tehokas tiedostojärjestelmä, joka voidaan asentaa Windows NT:hen, Windows 95:n ja Windows 98:n käyttämän vanhemman File Allocation Table (FAT) -tiedostojärjestelmän sijasta. Sen lisäksi, että NTFS tarjoaa tuen unicodelle, se on myös vakaampi kuin FAT ja tarjoaa suoran tuen hakemisto- ja kansiokohtaiselle tietoturvalle.

Kun käytät MFC:tä, sinun ei juurikaan tarvitse huolehtia alla olevasta tiedostojärjestelmästä. Jos käytät MFC:n **Cfile**-luokkaa tiedostojen käsittelyyn, tiedostojen nimet muunnetaan automaattisesti sopivaan TCHAR muotoon. **CFile** osaa käsitellä myös tilanteen, jossa ohjelmalla ei ole riittävästi käyttöoikeuksia halutun tiedon saamiseen. On tärkeää muistaa, että tiedostojen suojaus on yritysten tietokoneilla tiukempi ja tarkempi kuin oman kehityskoneesi testikansioissa, joten sovellusten ei tulisi koskaan olettaa automaattisesti, että tiedosto on saatavilla. Käytä systeemimuuttujia polkunimissä (esimerkiksi *SystemRoot* viitattaessa hakemistoon, johon käyttöjärjestelmä on asennettu) aina kun mahdollista välttääksesi kovakoodattuja polkunimiä. Systeemin ympäristömuuttujia voi tutkia **_tgetenv()** nimisellä API funktiolla.

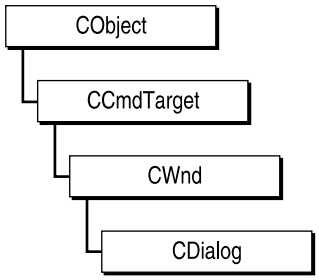
Näytön koordinaatit

Windows 95 ja Windows 98 -järjestelmissä kaikki näytön koordinaatit on rajoitettu 16-bittisiksi. Jos ohjelmoit Windows NT -koneella, muista, että rutiinit, jotka piirtävät näytölle koordinaattialueen -32768 ja 32767 ulkopuolelle, eivät toimi kunnolla Windows 95 ja Windows 98 -ympäristöissä.

MFC:n luokkahierarkia

MFC koostuu joukkosta C++-luokkia. Yksi tärkeimmistä tavoista, jolla koodia uusiokäytetään C++:ssa, on periyttäminen. C++-luokka voidaan johtaa kantaluokasta, jolloin se perii tämän ominaispiirteet. MFC-kirjaston, kuten monen muunkin C++-luokkakirjaston, sisältö järjestyy perintähierarkiaksi. Hierarkia koostuu suuresta joukosta tietyt toiminnot toteuttavia luokkia, jotka on periytetty pienestä joukosta kantaluokkia, jotka toteuttavat kaikille aliluokilleen yhteisen toiminnallisuuden.

Esimerkiksi kuvassa 3.1 näet **CDialog**-luokan perimisjärjestyksen. Tämä MFC-luokka edustaa Windowsin dialogia.



Kuva 3.1 **CDialog**-luokan perintäjärjestys

Hierarkian huipulla on **CObject**-luokka. **CObject** on kantaluokka suurelle joukolle MFC-luokkia. **CObject** tarjoaa peruspalvelut, kuten serialisointi (tietoyksiköiden avaaminen ja tallentaminen tiedostoihin), ajonaikaisen luokkainformaation, diagnostiikka- ja virheenjäljitystuen (valitations ja dumps), ja yhteensopivuuden kokoomaluokkien kanssa.

CCmdTarget-luokka on kantaluokka kaikille objekteille, jotka pystyvät käsittelemään Windowsin sanomia. Opit lisää tästä tämän luvun oppitunnilla 3, *MFC-sovelluskehys*.

CWnd-luokka, joka mainittiin kappaleen alussa, edustaa ikkunaa. Se, että **CDialog**-luokka on periytetty **CWnd**-luokasta, kuvaa hyvin ajatusta, että dialogi on ikkunan erikoistapaus.

MFC-objektihierarkiaan tutustuminen

Tässä harjoituksessa opit, kuinka käytät Visual C++ -ohjejärjestelmää MFC Hierarchy Chart -kartan tutkimiseen. Se esittää koko MFC-luokkajärjestelmän ja luokkien väliset perintäsuhteet.

► **MFC Hierarchy Chartin tutkiminen**

1. Käynnistä Visual C++. Valitse **Help**-valikosta **Index**. Visual Studion versio MSDN Librarysta avautuu, kohdistin on sijoitettu **Type in the keyword to find** -muokkausruutuun.
2. Kirjoita **Type in the keyword to find** -ruutuun **hierarchy chart**.
3. Kaksoisnapauta **keyword**-luettelosta kohtaa **hierarchy chart**. MFC:n Hierarchy Chart avautuu ruudun oikeaan reunaan. Muuta osan kokoa niin, että näet niin suuren osan kartasta kuin mahdollista.
4. Tutkaile Hierarcy Chartia. Huomaa, kuinka luokat on jaettu ryhmiin, jotka on nimetty lihavoiduilla tunnuksilla. Katso, mitkä luokista on periytetty **CObject-**luokasta, mitkä **CCmdTarget**-luokasta ja mitkä on periytetty **CWnd**-luokasta. Kannattaa ehkä pitää tämä kartta näkyvillä, kun luet tätä lukua.

MFC:n luokkien jaottelu

Kun opettelet MFC-luokkia, ne voivat olla helpommin hahmotettavissa taulukon 3.2 mukaisiin ryhmiin jaettuina.

Taulukko 3.2 MFC-luokkien jaottelu

MFC-luokkaryhmä	Kuvaus
Application architecture	Nämä luokat edustavat sovelluksen arkkitehtuurin peruselementtejä. Ryhmään kuuluu myös luokka **CwinApp**, joka edustaa ohjelmaa itseään.
User interface	Nämä luokat pitävät tunnusomaisesti sisällään ne Windows-pohjaisen sovelluksen osat, jotka näkyvät käyttäjälle. Näihin kuuluvat ikkunat, dialogit, valikot ja kontrollit. Käyttöliittymäluokat kapseloivat myös Windowsin piirtopinnan ja GDI:n piirto-objektit.
Collections	MFC sisältää joukon helppokäyttöisiä kokoomaluokkia, kuten **Arrays**, **Lists** ja **Maps**. Näistä on olemassa sekä template, että non-template -versiot.
General purpose	MFC:hen sisältyy joukko yleiskäyttöisiä luokkia, jotka eivät kapseloi Win32 API:n funktioita. Nämä luokat edustavat yksinkertaisia tietotyyppejä, kuten pisteitä, neliöitä ja monimutkaisempia tietotyyppejä, kuten merkkijonoja.
ActiveX	MFC:ssä on joukko luokkia, jotka yksinkertaistavat ActiveX-toimintojen lisäämistä sovellukseesi ja lyhentävät merkittävästi ohjelmointiin kuluvaa aikaa. ActiveX-luokat toimivat yhteistyössä muiden sovellusrunkoon kuuluvien luokkien kanssa avaten helpon tien ActiveX API:in.
Database	Tietojen haku tietokannasta on yleisimpiä ohjelmointitehtäviä Windows-ympäristössä. MFC sisältää luokkia, jotka mahdollistavat toimenpiteet Open Database Connectivity (ODBC) ja Data Access Object (ADO) -liittymien kautta.
Internet	Internetiä ja intranetiä hyödyntävien sovellusten tekemisestä on tulossa tärkeä ohjelmoinnin painopistealue. MFC:hen kuuluvat WinInet API ja Internet Server API (ISAPI), jotka sisältävät asiakas- ja palvelinosien luokat, molemmat erikseen.

Global functions	MFC:hen kuuluu joitakin funktioita, jotka eivät kuulu mihinkään kategoriaan. Näiden globaalien funktioiden nimet alkavat yleensä etuliitteellä afx ja ne tarjoavat ohjelmoijille yleiskäyttöisiä lisätoimintoja. Yleisesti käytetty luokka on esimerkiksi **AfxMessageBox().**

MFC:n DLL:t

AppWizard antaa mahdollisuuden valita, käytetäänkö MFC:n kirjastoja jaettujen DLL:ien kautta vai linkitetäänkö ne mukaan suoritettavaan sovellustiedostoon. Jos käytetään DLL:iä, tulee varmistaa, että sovelluksen käyttäjän tietokoneella ovat käytettävissä MFC:n kirjasto MFCxx.DLL (missä xx tarkoittaa käytetyn version numeroa) ja Visual C++:n standardikirjasto MSVCRT.DLL. Tämän varmistamiseksi DLL-tiedostot pakataan yhteen sovelluksen kanssa ja toimitetaan sen mukana.

Huomio Unicode-sovellusten tulee käyttää kirjaston versiota MFCxxU.DLL.

Käyttämällä jaettuja DLL:iä voidaan suoritettavien tiedostojen kokoa pienentää huomattavasti. Tämä on järkevää, jos samaan tietokoneeseen asennetaan suuri määrä MFC-pohjaisia sovelluksia. Näin sovellukset voivat käyttää samoja DLL:iä sen sijaan, että tuhlaisivat levytilaa useisiin kopioihin samoista DLL:stä.

MFC:n laajennetut DLL:t

Ohjelmoidessasi C++:lla teet usein mukautettuja luokkia, joita käytät myöhemmin uudelleen muissa sovelluksissa. MFC:n kirjastojen tapaan nämä luokat on usein pakattu DLL-tiedostoihin. DLL:ien avulla muiden sovellusten käyttöön jaettuja DLL:iä sanotaan *julkaistuiksi* (exported) — tämä tarkoittaa sitä, että niiden jäsenfunktiot ja jäsenmuuttujat tuodaan asiakassovellusten näkyville. Esimerkiksi MFC julkistaa DLL:n *CString*-luokan. Se tarkoittaa sitä, että sovellukset, jotka on linkitetty MFC DLL:iin voivat luoda ja käyttää **CString**-objekteja.

Voit valita **New Project** -toiminnon **MFC AppWizard (dll)**-dialogissa luodaksesi dll:iä, jotka julkistavat omia luokkiasi. AppWizard viittaa tämä tyyppisiin DLL:iin nimityksellä *regular DLL*.

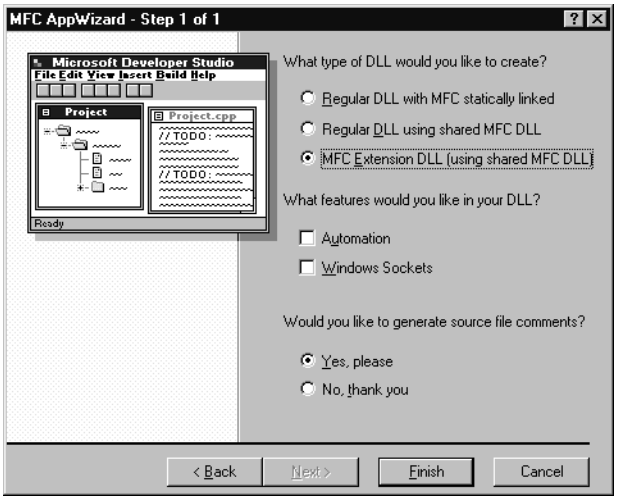
Regular DLL:ät voivat käyttää MFC:n luokkia toteutuksessaan. Mietitäänpä kuitenkin mitä tapahtuu, jos haluat julkaista luokan, joka on *periytetty* MFC:n luokasta. Kuvitellaan, että luot **CMyDialog**-luokan, joka on mukautettu dialogiluokka ja periytetty MFC:n **CDialog**-luokasta. Se sisältää joukon julkisia funktioita, joiden avulla asiakassovellukset voivat asettaa ja hakea tietoja kontrolleista. Kuinka voit varmistaa, että kantaluokka (**CDialog**) on kunnollisesti julkaistu ja että sen jäsenfunktiot ja muuttujat ovat asiakassovellusten saatavilla?

CMyDialog-luokan käyttö MFC-luokan tavoin, **CDialog**-kantaluokan jäsenfunktioiden kutsuminen ja viittaaminen **CmyDialog**-objektiin käyttämällä

CDialog-osoittimia, mahdollistetaan paketoimalla luokka erityiseen *laajennettuun* DLL:ään (MFC extension DLL). Laajennetut MFC DLL-tiedostot sisältävät olemassaolevista MFC-luokista periytettyjen luokkien uudelleen käytettävän koodin. Laajennetut MFC DLL -tiedostot antavat sinulle mahdollisuuden "laajentaa" MFC:tä.

Laajennetut DLL:ät on ohjelmoitu MFC:n jaettua DLL-versiota käyttäen. Vain MFC-ohjelmat (tai DLL:ät), jotka on tehty käyttäen MFC:n jaettua versiota, voivat käyttää laajennettuja DLL:iä. Sekä asiakassovelluksen että laajennetun DLL:n täytyy käyttää MFC DLL:n samaa versiota.

Laajennettu DLL tehdään valitsemalla **MFC AppWizard (dll)** -toiminto **New Project** -dialogissa ja valitsemalla MFC DLL AppWizardin ensimmäisellä sivulla (kuva 3.2) vaihtoehto **MFC Extension DLL**.



Kuva 3.2 Laajennetun MFC DLL:n luominen

Oppitunnin yhteenveto

Microsoft Foundation Class Library -kirjasto on kokoelma C++-luokkia, jotka on suunniteltu pääasiassa Microsoft Windows -pohjaisten sovellusten luomiseen.

MFC kapseloi kaikkein yleisimmin käytetyt Win32 API -funktiot. Win32 API on joukko käyttöjärjestelmän tarjoamia funktioita, rakenteita, sanomia, makroja ja rajapintoja, joita voit käyttää tehdessäsi sovelluksia mihin tahansa Microsoftin 32-bittiseen ympäristöön. MFC yksinkertaistaa Windows-ohjelmointia kätkemällä joitakin Win32 API:n monimutkaisempia ominaisuuksia. MFC:tä tulisi käyttää aina kuin mahdollista ajan ja vaivan säästämiseksi. Vaikka käytätkin MFC:tä, voit silti käyttää Win32 APIa, kun tarvitset tiettyjä matalan tason toimintoja, joita MFC ei tue.

Vaikka Win32 API on suunniteltu rajapinnaksi kaikkiin 32-bittisiin Windows käyttöjärjestelmiin, niiden välillä on kuitenkin muutamia eroja. Windows NT esimerkiksi käyttää sisäisesti Unicode merkkijonoja ja tukee NTFS:ää, mutta Windows 95 ja Windows 98 eivät. Vaikka MFC suojaakin ohjelmoijaa useimmilta ympäristöjen eroilta, sinun täytyy silti olla niistä tietoinen.

MFC koostuu joukosta hierarkkisia C++-luokkia, jotka käyttävät C++ -periyttämismekanismia tarjotakseen uudelleen käytettävän ja laajennettavan peruskoodin Win32-ohjelmoijille. MFC-luokkien ryhmät ovat:

- Application architecture classes (sovelluksen arkkitehtuuri).
- User interface classes (käyttöliittymä).
- Collection classes (kokoelmat).
- General purpose classes (yleiskäyttöiset luokat).
- ActiveX classes.
- Database classes (tietokantaluokat).
- Internet classes.
- Global functions.

Sovellukseen, jonka toteutuksessa on käytetty MFC-luokkia, voidaan MFC:n kirjastot linkittää joko kiinteästi suoritettavaan tiedostoon tai sitten kirjastot voidaan jakaa MFC DLL:inä muiden sovellusten kanssa. DLL:ien käyttäminen säästää levytilaa käyttäjän tietokoneessa, mutta tällöin täytyy varmistua siitä, että asiaan kuuluvat DLL:ät ja MSVCRT.DLL ovat kaikki asennetun sovelluksen saatavilla.

Voit luoda sekä tavallisia DLL:iä että MFC:n laajennettuja DLL:iä. MFC:n laajennetut DLL:ät sisältävät uudelleen käytettäviä luokkia, jotka on periytetty MFC-luokista. Laajennetut DLL:ät on tehty käyttäen MFC:n jaettuja DLL-versioita.

Oppitunti 2: Win32-sovellusarkkitehtuuri

Jotta voisit ymmärtää tavan, jolla MFC toteuttaa Windows-sovelluksen, sinun täytyy tuntea Win32-ympäristön arkkitehtuuri ja sovellukset, jotka siinä toimivat. Tällä oppitunnilla opit Win32-sovelluksen perusrakenteen ja -toiminnot.

Tämän oppitunnin jälkeen:

- Tunnet prosessien, säikeiden ja sovellusten väliset yhteydet.
- Tiedät, mikä on sanomien tehtävä Win32-ympäristössä.
- Tiedät, mitkä ovat välttämättömät työvaiheet luotaessa Win32-sovellusta.

Oppitunnin arvioitu kesto: 30 minuuttia

Windows-sovelluksen perusteet

Kohdeympäristön kunnollinen tunteminen on välttämätöntä, jotta voidaan tehdä tehokkaita ohjelmia. Tällä oppitunnilla kerrotaan Win32-käyttöjärjestelmien arkkitehtuurin elementeistä, joiden avulla pystyt tekemään hyvin käyttäytyviä sovelluksia, jotka hyödyntävät täysimittaisesti käyttöjärjestelmän palveluja.

Prosessit ja säikeet

Windows-ohjelma koostuu yhdestä tai useammasta *prosessista*. Prosessi (process) tarkoittaa yksinkertaistettuna suoritettavan sovelluksen ilmentymää. Prosessilla on käytössään osoiteavaruus sekä joukko resursseja ja yksi tai useampi säie, joka toimii prosessin viitekehyksessä.

Säie (thread) on perusyksikkö, jolle käyttöjärjestelmä jakaa prosessoriaikaa ja on pienin koodin osa, joka voidaan suorittaa ajastettuna. Säie toimii prosessin osoiteavaruudessa ja käyttää prosessille kuuluvia resursseja.

Prosessilla on aina vähintään yksi suoritettava säie, jota sanotaan *pääsäikeeksi* (primary thread). Voit luoda *toissijaisia* säikeitä huolehtimaan taustalla suoritettavista tehtävistä ja hyödyntää näin Windows 32-bittisten käyttöjärjestelmien moniajo-ominaisuuksia. Useamman kuin yhden säikeen käyttämistä sovelluksessa sanotaan *moniajoksi* (multithreading).

Sovelluksen käynnistyminen

Kun sovellus käynnistetään, käyttöjärjestelmä luo prosessin ja alkaa suorittaa sovelluksen pääsäiettä. Kun tämä säie lopetetaan, myös prosessi lopetetaan. Pääsäie ilmoitetaan käyttöjärjestelmälle funktionosoittimena käynnistyskoodissa. Kaikissa Windows-sovelluksissa on määriteltynä aloitusfunktio, joka on nimeltään **WinMain()**. WinMain-funktion osoite toimitetaan pääsäikeenä.

Tämän jälkeen sovellus luo ikkunan, joka toimii käyttöliittymänä. Ennen kuin ikkunoita voidaan esittää näytöllä, eri tyyppiset *ikkunaluokat* (window classes) täytyy rekisteröidä käyttöjärjestelmän kanssa. Ikkunaluokat ovat malleja, jotka sisältävät ikkunoiden luomiseen liittyvät yksityiskohdat. Rekisteröitäessä ikkunat yhdistetään *ikkunaproseduuriin* (window procedure), jonka avulla määrätään, mitä ikkunassa näkyy ja kuinka se reagoi käyttäjän syöttämään informaatioon määrittämällä sen reaktiot järjestelmän sanomiin.

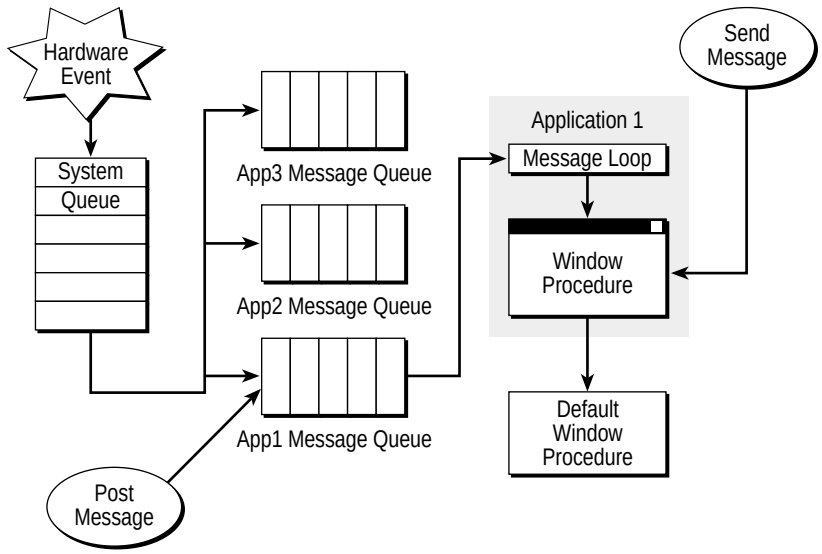
Windowsin sanomat

Ikkuna on käyttäjän ja sovelluksen välisen viestinnän tärkein väylä, sen sijaan sisäinen viestintä käyttöjärjestelmän, sovellusten ja sovelluksen komponenttien kanssa tapahtuu järjestelmän *sanomien* (messages) avulla. Esimerkiksi luotaessa sovelluksen ilmentymää käyttöjärjestelmä lähettää sovellukselle joukon sanomia, joihin sovellus vastaa saattamalla itsensä toimintakuntoon. Näppäimistön ja hiiren käyttäminen saa käyttöjärjestelmän luomaan sanomia, jotka sitten lähetetään asianomaiselle sovellukselle. Windows-pohjaisen käyttöjärjestelmän tärkeimpänä tehtävänä voidaan pitää sen vastaan ottamien sanomien käsittelemistä. Käsittely tarkoittaa sanomien välittämistä niille sovelluksille, joille ne on tarkoitettu ja odotettujen toimintojen suorittamista riippuen sanomien tyypistä ja parametreistä. Sovelluksen suunnittelijan tehtävä on päättää, mikä funktio minkäkin sanoman käsittelee ja huolehtii asianmukaisesta vastauksesta.

Sovelluksen sanomien käsittely

Jokainen suoritettavana oleva säie, joka luo ikkunan, on yhdistetty *sanomajonoon* (message queue). Sanomajono on tietorakenne, johon käyttöjärjestelmä varastoi ikkunalle toimitettavat sanomat. Kaikilla sovelluksilla on pääikkuna ja kaikilla pääikkunoilla on *sanomasilmukka* (message loop). Sanomasilmukka on ohjelmakoodin osa, joka vastaanottaa sanomat sanomajonosta ja lähettää sanomat asianmukaiselle proseduurille. Ikkunan proseduuri voi suorittaa sanoman seurauksena sovelluksesta riippuvan toiminnon tai siirtää sanoman oletuskäsittelijälle (default window procedure) — järjestelmän määrittelemälle funktiolle, joka suorittaa sanomaan liittyvät oletus toimenpiteet. Esimerkiksi sovelluksen saama sanoma, joka ilmoittaa, että käyttäjä on pienentänyt sovelluksen ikkunan, käsitellään samalla tavoin lähes jokaisessa sovelluksessa. Tässä tapauksessa oletuskäsittelijä on sopiva valinta.

Kuvassa 3.3 seuraavalla sivulla esitetään, kuinka järjestelmä asettaa sanomat jonoon ja kuinka sovellus ne käsittelee. Huomaa, kuinka sanomia voidaan luoda **PostMessage()** ja **SendMessage()** -funktioita käyttäen siinä missä laitteistossakin. Voit käyttää näitä Win32 API -funktioita tai ehkä mieluummin niiden MFC-vastineita **CWnd::PostMessage()** ja **CWnd::SendMessage()**, sanomien lähettämiseen sovelluksellesi tai sovelluksestasi. **PostMessage()** asettaa sanoman ikkunan viestijonoon ja palauttaa suorituksen ohjelmalle odottamatta, että ikkuna käsittelee viestin. **SendMessage()** lähettää sanoman, eikä palaa ennen kuin ikkuna on käsitellyt viestin.



Kuva 3.3 Windows NT:n sanomien prosessointi

Windows-sovelluksen elintärkeät osat

Windows-sovelluksen arkkitehtuurin käsittelyn voisi tiivistää seuraavaan listaan tehtävistä, jotka täytyy suorittaa Win32-sovellusta luotaessa:

- Luo **WinMain()**-funktio sovelluksen aloituskohdaksi.
- Rekisteröi jokainen ikkunaluokka ja julkista niihin liittyvät ikkunaproseduurit.
- Luo sovelluksen pääikkunan ilmentymä.
- Tee sanomasilmukka, joka huolehtii viestien toimittamisesta sopivalle ikkunaproseduurille.
- Tee ikkunaproseduurit, jotka käsittelevät sanomat.

Win32-sovelluksen luominen

Seuraavassa harjoituksessa nähdään kuinka Visual C++:lla tehdään yksinkertainen Windows-sovellus, joka suorittaa kaikki viisi tärkeää toimintoa. Harjoituksessa ei käytetä MFC:tä.

► **Win32-sovelluksen tekeminen**

1. Käynnistä Visual C++.
2. Valitse **File**-valikosta **New**.
3. Valitse vaihtoehto **Win32 Application**.
4. Kirjoita projektin nimeksi **MyWin32App**.
5. Napauta **OK**. Win32 AppWizard avautuu. Valitse **A typical "Hello World" application**, ja napauta **Finish**.
6. Luo projekti napauttamalla **New Project Information** -dialogissa **OK**.
7. Kun projekti on luotu, valitse **Workspace**-osan **FileView**-välilehti. Avaa näkyviin MyWin32App-tiedostot napauttamalla plusmerkkiä. Avaa samalla tavalla Source Files -kansio.
8. Aloita luodun lähdekoodin tutkiminen kaksoisnapauttamalla **MyWin32App.cpp**-tiedoston kuvaketta.
9. Käy koodi läpi ja etsi seuraavat kohdat:

- **WinMain()**-funktio, joka kutsuu alustusfunktioita ja luo pääsanomasilmukan.
- **MyRegisterClass()**-funktio, joka rekisteröi sovelluksen pääikkunan ikkunaluokan.
- Ikkunaproseduurifunktio **WndProc()**, joka on yhdistetty pääikkunaan **MyRegisterClass()**-funktiossa. Tämä ikkunaproseduuri käsittelee WM_COMMAND-sanoman (sanomat valikkokomennoilta, kontrolleilta ja pikanäppäimiltä), WM_PAINT-sanoman (lähetetään, kun järjestelmä pyytää maalaamaan sovelluksen ikkunan osan) ja WM_DESTROY-sanoman (lähetetään, kun ikkuna tuhotaan). Kaikki muut sanomat lähetetään oletusikkunaproseduurille.
- **InitInstance()**-funktio, joka luo ja tuo näkyviin pääikkunan ilmentymän.

Oppitunnin yhteenveto

Tällä oppitunnilla opit Win32-ympäristössä toimivan sovelluksen arkkitehtuurin perusteet ja perustyövaiheet, jotka tarvitaan luotaessa Windows-sovellusta Win32 API:n avulla.

Prosessi voidaan määritellä suoritettavan ohjelman ilmentymäksi. Jokaisella prosessilla on vähintään yksi säie, jota kutsutaan pääsäikeeksi. Kun sovellus käynnistyy, luodaan uusi prosessi ja sovelluksen **WinMain()**-funktion osoite lähetetään käyttöjärjestelmälle. **WinMain()**-funktion koodi suoritetaan sovelluksen pääsäikeessä.

Windows-sovellus odottaa laitteiston, järjestelmän tai muiden sovellusten ja komponenttien aiheuttamia tapahtumia. Sanomat käsitellään sovelluksen ikkuna-objekteihin yhdistetyissä proseduureissa. Ohjelmoija laatii nämä proseduurit, jotka toteuttavat ohjelmalta toivotut toiminnot.

Oppitunti 3: MFC sovelluskehys

Kun nyt tunnet Windows-sovelluksen arkkitehtuurin, olet valmis oppimaan, kuinka MFC toteuttaa perus-Windows-sovelluksen.

Sen lisäksi, että MFC kapseloi Win32 API:n, se määrittelee pienen joukon luokkia, jotka edustavat Windows-sovelluksen perustoimintoja ja luo yhteydet näihin toteuttaakseen Windows-sovelluksen tärkeimmät toiminnot. Nämä sovellusarkkitehtuuriin liittyvät luokat, yhdessä monien globaalien funktioiden kanssa, muodostavat *sovelluskehyksen* (application framework), jota voit käyttää sovelluksen perustana. Voit luoda sovelluskehyksestä periytettyjä luokkia MFC:n AppWizar -dia käyttämällä. Näiden luokkien päälle voit rakentaa sovelluksen, joka vastaa omia tarpeitasi.

Tämän oppitunnin jälkeen:

- Tiedät, kuinka MFC-sovelluskehys toteuttaa Windows-sovelluksen.
- Tiedät, mikä on sovellusluokan ja pääikkunaluokan rooli sovelluskehyksessä.
- Tiedät, kuinka AppWizardia käytetään kehykseen perustuvan yksinkertaisen sovelluksen luomiseen.
- Tiedät, kuinka sovelluskehys käsittelee Windowsin sanomia.

Oppitunnin arvioitu kesto: 40 minuuttia

MFC-sovelluksen arkkitehtuuri

MFC-sovelluskehys sisältää Windows-sovellusten perusarkkitehtuurin toteuttamisessa tarvittavat palvelut:

- Luokka, joka edustaa sovellusta.
- **WinMain()**-funktion toteutus.
- Luokka, joka edustaa sovelluksen pääikkunaa.

Sovellusluokka

MFC:n luokka **CWinApp** edustaa sovellusta kokonaisuudessaan. **CWinApp** on periytetty **CWinThread**-luokasta, joka edustaa säiettä MFC-sovelluksessa. **CWinApp** edustaa sovelluksen prosessin pääsäiettä ja kapseloi Windows-ohjelman alustamisen, käynnistämisen ja sulkemisen. Nämä toiminnot toteuttavat funktiot on kuvattu taulukossa 3.3.

Taulukko 3.3 CWin App:n jäsenfunktiot

CWinApp:n jäsenfunktio	Tarkoitus
InitInstance()	Alustaa jokaisen uuden sovelluksen ilmentymän, jota suoritetaan Windowsissa. Tuo sovelluksen pääikkunan näkyviin.
Run()	Sisältää sanomasilmukan toteutuksen.
OnIdle()	Sovelluskehys kutsuu tätä funktiota, kun muita Windows-sanomia ei käsitellä. Voit käyttää funktiota taustalla toimivien tehtävien suorittamiseen.
ExitInstance()	Kutsutaan aina, kun sovelluksen kopio suljetaan.

MFC-kehyksen päälle rakennetun sovelluksen täytyy toteuttaa yksi ja vain yksi **CWinApp**-luokasta periytetty luokka. Sovellukselle täytyy tehdä myös oma ylikuormitettu versio **InitInstance()**-jäsenfunktiosta. **WinMain()** kutsuu **InitInstance()-**funktiota suoraan ja se on oikea paikka sovelluksen omien alustustoimenpiteiden tekemiseen.

WinMain()-funktio

Sovelluskehys tuottaa sovelluksellesi valmiin **WinMain()**-funktion. **WinMain()** kutsuu useita globaaleja funktioita, jotka huolehtivat standardeista alustustoimista, kuten luokkien rekisteröinneistä. Tämän jälkeen se kutsuu sovellusobjektin **InitInstance()-**jäsenfunktiota, joka suorittaa sovelluksen alustustoimet. Seuraavaksi **WinMain()** käynnistää sovelluksen sanomasilmukan kutsumalla sovellusobjektin **Run()**-jäsenfunktiota. Sanomasilmukka hakee ja toimittaa sanomia kunnes vastaan ottaa WM_QUIT sanoman, jolloin se kutsuu sovellusobjektin **ExitInstance()**-funktiota ja lopettaa toimintansa. **WinMain()** kutsuu tämän jälkeen siivousrutiineja ja päättää sovelluksen suorittamisen.

Pääikkuna

Pääikkuna (main window) on Windows-sovelluksen elintärkeä osa, koska se on sovelluksen tärkein käyttöliittymä. Se voi olla yksinkertainen dialogi-ikkuna tai se voi olla *kehysikkuna*, joka pitää sisällään sovelluksen valikon, työkalurivit, ikkunan työalueen ja jonka kokoa voidaan muuttaa. Molemmissa tapauksissa sovelluskehykseen perustuvan sovelluksen tulee sisältää luokka — joko **CDialog** tai **CframeWnd**-luokasta periytetty — jota sovellus käyttää luodessaan pääikkunaobjektin. Sovellusobjektin **InitInstance()**-funktio avaa pääikkunan.

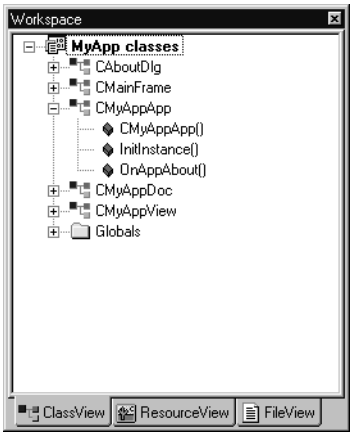
Palataan harjoituksen vuoksi luvussa 2 luomaasi MyApp-projektiin ja tutkitaan kehysluokkia, jotka on jo luotu puolestasi.

► **MyApp-sovelluksen sovelluskehysluokkien tutkiminen**

1. Avaa MyApp-työtila valitsemalla **Open Workspace** -komento **File**-valikosta. Jos työsi on tallennettu oletushakemistoon, työtilan nimi ilmestyy luetteloon.

Jos näin ei käy, hae paikka, johon tallensit työtilan. Avaa tiedosto **CMyApp.dsw** kaksoisnapauttamalla.

2. Avaa ClassView-ikkunassa listan ensimmäinen kohta, **MyApp classes** (jos se ei ole jo valmiiksi auki). Näin voit katsella tiedostoja, jotka AppWiz on luonut tätä projektia varten.
3. Avaa **CmyAppApp**-luokka, jolloin näet AppWizardin tästä luokasta ylikirjoittamat metodit. Työtilasi tulisi näyttää samalta kuin kuvassa 3.4.



Kuva 3.4 MyApp-projektin ClassViewikkuna

4. Kaksoisnapauta **CMyAppApp**-luokkaa, jolloin näet luokan määrittelyt. Huomaa, kuinka tämä luokka on periytetty MFC:n luokasta **CWinApp**.
5. Kaksoisnapauta kohtaa **InitInstance()**, jolloin sen lähdekoodi tulee näkyviin. Lue koodi läpi kommentteineen saadaksesi käsityksen siitä, mitä alustustoimenpiteitä tämä funktio tekee. Alustustoimet, jotka AppWizardin luoma versio **InitInstance()**-metodista suorittaa, perustuvat valintoihin, joita olet AppWizardia käyttäessäsi tehnyt.

Sovelluskehyksen sanomien käsittely

Oppitunnilla 2 opit, että yksi avaintehtäviä Windows-sovellusta ohjelmoitaessa on järjestelmän sanomien ohjaaminen niitä käsitteleville funktioille. Ilman MFC:tä tehdyissä Win32-sovelluksissa täytyy tehdä ikkuna-aliohjelma jokaista rekisteröityä ikkunaluokkaa kohden. Nämä ikkuna-aliohjelmat on usein toteutettu ohjaamalla switch-lausetta sanomien tunnistekentän perusteella. MFC-kehykseen perustuvissa sovelluksissa sovelluksen luokkien jäsenfunktiot huolehtivat sanomien käsittelemisestä. Ne voivat olla itse tehtyjä tai AppWizardin sovellukseen lisäämiä funktioita. Sanomien ohjaaminen oikeille käsittelijöille tapahtuu *sanomakarttojen* (message map) avulla.

Sanomakartta

Sanomakartta on luokan määrittelyssä esitelty taulukko, joka ohjaa järjestelmältä tulevat sanomat luokan jäsenfunktioille. Sanomakartta sisältää merkinnät, jotka yhdistävät sanomatunnisteet sanomia käsitteleviin funktioihin. Sanomakartta tuntee neljä eri sanomatyyppiä, jotka on kuvattu taulukossa 3.4.

Taulukko 3.4 MFC:n sanomatyyput

Sanomatyyppi	Kuvaus
Windows	Windows-sanomat ovat käyttöjärjestelmän tuottamia. Ne ilmoittavat sovellukselle ikkunan luomisesta, tulevasta ikkunan sulkemisesta, hiiren ja näppäimistön tapahtumista, järjestelmän värien muutoksista ja kaikesta muusta, mikä voi vaikuttaa sovelluksen toimintaan. Näiden sanomien tunnisteet alkavat yleensä liitteellä WM_. Windows-sanomat käsittelee tavallisesti se ikkuna, jolle Windows sanoman lähettää.
Command	Command-sanomat syntyvät käyttäjän käyttäessä käyttöliittymää — esimerkiksi, kun käyttäjä valitsee valikkotoiminnon, napauttaa työkalurivin painiketta tai painaa pikanäppäintä. Kun jokin näistä tapahtumista syntyy, lähetetään **WM_COMMAND**-sanoma tapahtuman yksilöivän tiedon kera sovellukselle. Kehys ohjaa Command-sanomat sovelluksen objekteille. Sanoman ohjaus on ominaisuus, jonka avulla sovellus pystyy käsittelemään sanoman luokassa, joka on todennäköisimmin yhteydessä sanomaan.
User interface update command	User interface update command -sanomat synnyttää sovellusrunko sovelluksen sisällä — eli ne ovat osa MFC:tä. Ne viestittävät sovellukselle, että sovelluksen käyttöliittymän osa, kuten valikkokomento tai työkalurivi, tulee päivittää. Esimerkiksi ennen kuin valikko avautuu, sovellusrunko lähettää sovellukselle asiaan kuuluvan päivitys komennon, joka antaa sovellukselle tilaisuuden muuttaa komentojen tilaa — tuleeko niiden olla käytettävissä, poissa käytöstä tai pitääkö niiden olla merkittynä.
Control notification	Kontrollit ja muut lapsi-ikkunat lähettävät Control notification -sanomia kantaikkunoilleen. Yleensä ne lähetetään WM_COMMAND-sanomina, joissa on kontrollin yksilöivä määre. Esimerkiksi edit-kontrolli lähettää isännälleen WM_COMMAND-sanoman, jossa on EN_CHANGE-määre, kun käyttäjä tekee toiminnon, joka on saattanut muuttaa muokkausruudun tekstiä.

Sanomakartan luominen

MFC:n avulla sanomakarttojen luominen on helppoa. Mikä tahansa luokka, joka on periytetty **CCmdTarget**-luokasta, kykenee tukemaan sanomakarttaa. Kaikki AppWizardilla luodut kyhysluokat sisältävät perustoiminnot sanomien käsittelylle. Voit käyttää ClassWizard-työkalua luodessasi uusia luokkia, jotka sisältävät sanomakarttoja ja ylläpitää sitä käyttäen olemassaolevia luokkia lisäämällä ja poistamalla sanomakarttoihin tehtyjä merkintöjä. Käytä ClassWizardia luokkiesi sanomakarttojen käsitelyyn aina, kun se on mahdollista säästääksesi aikaa ja varmistaaksesi, että sanomakartat ovat oikein toteutettuja.

Käsittelijöiden tekeminen ClassWizardilla

Tässä osassa opit, kuinka lisäät ClassWizardia käyttämällä käsittelijän Windows-sanomille ja command-sanomille. Myöhemmissä luvuissa opit, kuinka käyttöliittymän päivityskomentosanomien ja control notification -sanomien käsittely toteutetaan. Tutkimalla syntyvää koodia opit, kuinka sanomakartat toteutetaan MFC-sovelluksessa.

Seuraavassa harjoituksessa lisätään MyApp-sovellukseen käsittelijä, joka avaa viesti-ikkunan, kun käyttäjä napauttaa sovellusikkunan *työaluetta* (client area). Työalue on se ikkunan alue, jossa sovelluksen käsittelemät tiedot yleensä esitetään.

► **Windows-sanoman käsittelijän tekeminen**

1. Palaa MyApp-projektiin.
2. Avaa ClassWizard painamalla CTRL+W. Napauta **Message Maps** -välilehteä.
3. Valitse **Class Name** -luettelosta **CMyAppView**. Tällä osoitetaan se, että haluat käsitellä sanoman, joka lähetetään sovelluksen *näkymä*-luokalle. Näkymäluokka kapseloi pääikkunan työalueen.
4. Valitse **Object IDs** -luettelosta **CMyAppView**.
5. Valitse **Messages**-luettelosta **WM_LBUTTONDOWN**. Tämä sanoma lähetetään sovellukselle, kun käyttäjä napauttaa hiiren vasemmalla näppäimellä ikkunan työaluetta.
6. Napauta **Add Function**. ClassWizard luo tynkäfunktion, johon voit lisätä koodisi. Ylikuormitetut **CWnd**-kantaluokan virtuaalifunktiot käsittelevät kaikki Windows-sanomat. Ylikuormitetun virtuaalifunktion nimi alkaa tekstillä "On" ja sitä seuraa sanoman nimi ilman "WM_"-osaa ja vain jokaisen sanan alkukirjain isolla kirjoitettuna. Tässä tapauksessa luotavan kantafunktion nimi on **OnLButtonDown()**.
7. Napauta **Edit Code**. MyAppView.cpp-tiedosto avautuu ja kohdistin sijoitetaan **OnLButtonDown()**-funktion alkuun.
8. Korvaa // TODO -kommenttirivi seuraavalla koodilla:

```
AfxMessageBox(“You clicked?”);
```

so that your code reads as follows:

```
void CMyAppView::OnLButtonDown(UINT nFlags, CPoint point)
{
    AfxMessageBox(“You clicked?”);
    CView::OnLButtonDown(nFlags, point);
}
```

9. Käännä MyApp-projekti painamalla F7 ja käynnistä sovellus painamalla CTRL+F5. Kokeile, että WM_LBUTTONDOWN-sanoma käsitellään odotetulla tavalla napautamalla hiiren vasemmalla painikkeella sovelluksen työaluetta. Sulje MyApp-sovellus.

Seuraavassa harjoituksessa tehdään käsittelijä sanomalle, joka syntyy, kun käyttäjä valitsee **Edit**-valikon komennon **Paste** MyApp-sovelluksessa.

► **Command-sanoman käsittelijän lisääminen**

1. Avaa ClassWizard painamalla CTRL+W MyApp-projektissa. Napauta **Message Maps** -välilehteä.
2. Valitse **Class Name** -luettelosta **CMyAppApp**. Muista, että command-sanoman voi käsitellä mikä tahansa sovelluksen luokista. Koska käsittelijä on pieni, sijoitamme sen sovellusluokkaan. Yleensä luokan, joka on läheisimmin tekemisissä komennon kanssa, tulisi käsitellä komennon aiheuttama sanoma. Esimerkiksi **CMainFrame**-luokka olisi paras käsittelijä komennolle, joka muuttaa kantaikkunan attribuutteja esimerkkiksi tilarivin piilottamista tai näyttämistä ohjaavaa attribuuttia.
3. Valitse **Object IDs** -luettelosta **ID_EDIT_PASTE**. Tämä tunniste välitetään WM_COMMAND-sanoman parametrina, kun **Paste**-komento valitaan **Edit**-valikosta.
4. Valitse **Messages**-luettelosta **COMMAND**. Tällä määritellään se, että käsiteltävänä on command-sanoma eikä käyttöliittymän päivityskomento -sanoma.
5. Napauta **Add Function**. Dialogi avautuu ja ehdottaa käsittelijälle nimeä **OnEditPaste()**. Tämä johtuu siitä, että olet määrittelemässä uutta funktiota sen sijaan, että ylikuormittaisit olemassaolevan virtuaalifunktion. Hyväksy nimi napauttamalla **OK**.
6. Napauta **Edit Code**. MyAppView.cpp-tiedosto avautuu ja kohdistin sijoitetaan funktion alkuun.
7. Korvaa // TODO -kommentti seuraavalla koodilla:

```
AfxMessageBox(“MYAPP does not support the Paste command”);
```

so that your code reads as follows:

```
void CMyAppApp::OnEditPaste()
{
    AfxMessageBox(“MYAPP does not support the Paste command”);
}
```

8. Käännä ja käynnistä CMyApp-sovellus. Kokeile **Edit**-valikon **Paste**-komentoa nähdäksesi, että käsittelijä toimii odotetulla tavalla. Huomaa, että sovelluskehys poistaa käytöstä komennot, joille ei ole käsittelijää.

Sanomakartan koodi

Kun lisäät käsittelijöitä ClassWizardilla, se tekee puolestasi seuraavat tehtävät:

- Lisää käsittelijän esittelyn luokan header-tiedostoon.
- Luo tynkäfunktion luokan toteutustiedostoon.
- Lisää merkinnän käsittelijästä sanomakarttaan.

► **CMyAppApp::OnEditPaste()-funktion esittelyn tutkiminen**

1. Avaa MyAppApp.h-tiedosto tutkittavaksi kaksoisnapauttamalla ClassViewissä **CMyAppApp**.
2. Hae tiedostosta seuraava koodi:

```
//{{AFX_MSG(CMyAppApp)
afx_msg void OnAppAbout();
afx_msg void OnEditPaste();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

Huomaa **DECLARE_MESSAGE_MAP**-makro. Se on sanomakartan välttämätön osa, jonka AppWizard tai ClassWizard lisää. Opit seuraavissa luvuissa, kuinka luokkia lisätään ClassWizardilla.

Olet jo tutkinut ClassWizardin luomia tynkäfunktioita ja lisännyt niihin koodia.

► **CMyAppApp::OnEditPaste()-funktioon liittyvän sanomakarttamerkinnän tutkiminen**

1. Palaa CMyApp.cpp-tiedostoon. Jos se ei ole auki, voit avata sen kaksoisnapauttamalla **OnEditPaste()**-funktion kuvaketta ClassViewissä (**CMyAppApp**-luokan kuvakkeen alla).
2. Lähellä CMyApp.cpp-tiedoston alkua on sanomakartta, joka näyttää tältä:

```
BEGIN_MESSAGE_MAP(CMyAppApp, CWinApp)
    //{{AFX_MSG_MAP(CMyAppApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)
    //}}AFX_MSG_MAP
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Voit nähdä, että ClassWizard on lisännyt **ON_COMMAND**-makron sanomakart-taan. Tämän makron rakenne on melko suoraviivainen. Ensimmäinen parametri on komennon tunniste (ID). (Opit lisää komentojen tunnisteiden määrittämisestä seuraavassa luvussa.) Toinen parametri on käsittelijän nimi.

OnButtonDown()-funktio käsittelee WM_BUTTONDOWN-sanoman ja se on määritelty MyAppView.h-tiedostossa seuraavasti:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

Vastaava merkintä sanomakartassa MyAppView.cpp-tiedostossa on:

```
ON_WM_LBUTTONDOWN()
```

Koska CWnd-kantaluokan ylikuormitetut virtuaalifunktiot käsittelevät Windows-sanomat, ei ole syytä välittää käsittelijän nimeä parametrinä. MFC:n kehys ohjaa sanoman oikealle virtuaalifunktiolle. Windows-sanomissa on usein lisäinformaatiota, joka on liitetty niihin parametrien muodossa. Sovellusrunko huolehtii tämän lisäinformaation poimimisesta ja välittämisestä käsittelijöille. Jos katsot **CMyApp::OnLButtonDown**-funktion määrittelyä, huomaat, että se saa seuraavat määreet: kaksi parametriä, jotka on johdettu WM_LBUTTONDOWN-sanoman parametreistä, jotka osoittavat hiiren osoittimen nykyisen sijainnin näytöllä; ja lippuarvon, jonka arvo ilmoittaa, onko jokin "virtuaalinäppäimistä" (esimerkiksi CTRL tai SHIFT) myös painettuna.

Sanomakartan makrot

MFC:ssä on määritelty neljä sanomakarttamakroa, jotka vastaavat neljää sanomatyyppiä, joita sanomakartta voi käsitellä. Nämä makrot on lueteltu taulukossa 3.5.

Taulukko 3.5 Sanomakartan makrot

Sanomatyyppi	Makron muoto	Parametrit
Standard Windows message	ON_WM_*XXX* (jossa *XXX* on sanoman nimi)	Ei ole
Command message	ON_COMMAND	komentotunniste, käsitelijän nimi
Update command	ON_UPDATE_COMMAND_UI	komentotunniste, käsitelijän nimi
Control notification	ON_*XXX* (jossa *XXX* on kontrollin ilmoituksen sisältävän parametrin nimi)	kontrollitunniste, käsittelijän nimi

Olet ehkä huomannut, että ClassWizardin tuottama koodi sijoitetaan kommenttilohkoon, joka alkaa {{AFX_MSG -merkinnällä ja päättyy }}AFX_MSG -merkintään. Nämä merkinnät osoittavat alueen, jota ClassWizard on muokannut. Voit tehdä omia merkintöjä sanomakarttaan käsin, mutta ne tulisi lisätä näiden lohkojen ulkopuolelle. Lohkon sisälle tehty merkintä voi johtaa

ClassWizardin virheelliseen toimintaan. Käsittelijän esittelyä edeltävä AFX_MSG merkintä on myös ClassWizardin käyttämä. Kääntäjään niillä ei ole vaikutusta.

Merkintöjen poistaminen sanomakartasta

Jos käytät ClassWizardia sanomakarttamerkintöjen poistamiseen, se poistaa myös funktion esittelyn luokan header-tiedostosta. Se ei kuitenkaan poista käsittelijää, jossa saattaa olla kirjoittamaasi koodia.

► **CCMyAppApp::OnEditPaste()-funktion poistaminen**

1. Käynnistä ClassWizard CMyApp-projektista painamalla CTRL+W. Napauta **Message Maps** -välilehteä.
2. Valitse **Class Name** -luettelosta **CCMyAppApp**.
3. Valitse **Member functions** -luettelosta **OnEditPaste**.
4. Napauta **Delete Function**. Näytölle ilmestyy muistutus, että sinun täytyy poistaa funktion toteutusosa itse. Napauta **Yes**.
5. Sulje ClassWizard napauttamalla **OK**.
6. Etsi funktio MyApp.cpp-tiedostosta ja poista se.
7. Käännä ja käynnistä sovellus. Varmista, että funktio on poistettu oikein ja katso, kuinka muutos on vaikuttanut **Edit**-valikkoon.

Komentojen reititys

Muistanet, että sovellusrunko reitittää command-sanomat sovelluksen objekteille. Tämä *komentojen reitittäminen* (command routing) antaa sovelluksen käsitellä sanoman siinä luokassa, johon komento läheisimmin liittyy. MFC:n sovellusrunko ohjaa komennon tietyssä järjestyksessä kohdeobjekteille (määrittelyn teke-vät sovelluksessasi ne **CCmdTarget**-luokasta periytetyt luokat, joissa sanoma-kartta on toteutettu) tarkistaakseen, tarjoaako jokin niistä käsittelijän komennolle. Jokainen komennon kohdeolio tutkii sanomakarttansa tarkistaakseen voiko se käsitellä tulleen sanoman.

Voit joskus hyödyntää tätä komentojen reitityssarjaa. Voit esimerkiksi asettaa sovelluksen pikanäppäimen lähettämään sanoman I**D_SAVE_WINDOW_STATE**-tunnisteella. Tällainen pikanäppäin ilmoittaa, että käyttäjä haluaa tallentaa aktiivisen ikkunan asetukset. Jotkut sovelluksen ikkunaobjekteista voivat sisältää mukautetun version käsittelijästä **OnSaveWindowState()**. Muut ikkunat kuten väliaikaiset dialogit ja ohjeruudut eivät voi. Komennon reitityssarja tarkistaa ensin aktiivisen ikkunan sanomakartasta, toteuttaako se **OnSaveWindowState()**-funktion. Jos se ei näin tee, seuraavaksi tutkitaan kantaikkunan sanomakartta. Jos sekään ei sisällä käsittelijää, tarkistetaan sovellusobjektin sanomakartta.

Sanoman reitityssarja vaihtelee tilanteen mukaan. Lisätietoja reitityksestä saat, kun teet Visual C++:n ohjeessa haun "command routing".

Oppitunnin yhteenveto

MFC-sovelluskehys on joukko yhdessä toimivia luokkia, jotka yhdessä muutamien globaalien funktioiden kanssa, toteuttavat perus-Windows-sovelluksen. Käyttämällä AppWizardia voit helposti periyttää sovellusrungon luokista luokkia, joiden pohjalta voit kehittää omia erityistarpeitasi vastaavan ohjelman.

Sovellusrunkoon kuuluva **CWinApp**-luokka kapseloi sovelluksen kokonaisuudessaan. Se julkistaa useita funktioita, joita sovellusrungon **WinMain()**-funktio kutsuu käsitellessään Windows-pohjaisen sovelluksen alustusta, suoritusta ja lopettamista. **CFrameWnd**-luokka kapseloi sovelluksen pääikkunana toimivan kehysikkunan, joka isännöi sovelluksen valikkoja, työkalurivejä ja ikkunan työaluetta.

MFC:n sovellusrunkoon perustuvassa sovelluksessa sanomia käsittelevät sovelluksen luokkien funktiot. Sanomien ohjaaminen oikeille käsittelijöille hoidetaan sanomakartan avulla. Sanomakartta on MFC:n **CCmdTarget**-luokasta periytettyjen luokkien tukema ominaisuus. Sanomakartta on taulukko, joka luokan määrittelyssä yhdistää tietyt sanomatunnisteet luokan funktioihin. Sanomakartan merkintöjen lisäämiseen ja poistamiseen tulisi käyttää ClassWizardia.

Tällä tunnilla opit, kuinka ClassWizardia käytetään Windows-sanomien käsittelijöiden muodostamiseen. Käyttöjärjestelmä muodostaa Windows-sanoman informoidakseen sovellusta Windows-ympäristössä tapahtuvista muutoksista. Näihin muutoksiin sisältyvät hiiren ja näppäimistön tapahtumat ja command-sanomat. Command-sanomat syntyvät käyttäjän toimiessa vuorovaikutuksessa käyttöliittymän elementtien kuten valikoiden ja työkalupalkkien kanssa.

Sovellusrunko reitittää command-sanomia sovelluksen objektilta toiselle määrätyssä järjestyksessä kunnes sopiva käsittelijä löytyy. Tämä sanomien reititys antaa sovellukselle mahdollisuuden suorittaa tapahtuman käsittelyn luokassa, joka läheisimmin liittyy sanomaan.

Oppitunti 4: Dokumentti/näkymä-arkkitehtuuri

Sovelluksen tehtävä on suorittaa toimenpiteitä tietojoukolle. Kun tehdään sovelluksen loogista suunnittelua, on määriteltävä tapa, jolla sovelluksen tiedot jäsennetään ja kuinka ne esitetään käyttäjälle niin, että hän voi olla niiden kanssa vuorovaikutuksessa.

MFC:n asiakirja/näkymä-arkkitehtuuri on yksi yhtenäinen tapa koordinoida sovelluksen tietoja ja tapaa, jolla tietoja katsellaan. Asiakirja/näkymä-arkkitehtuurin toteuttavat luokat, jotka luodaan samalla, kun teet sovelluksen käyttäen AppWizardia. Tällä oppitunnilla kerrotaan asiakirja/näkymä-arkkitehtuurin ominaisuuksista ja siitä kuinka sovellusrunko sen toteuttaa.

Tämän oppitunnin jälkeen:

- Osaat kuvata dokumentti/näkymä-arkkitehtuurin ja ne edut, jotka se tarjoaa sovelluskehittäjille.
- Tiedät, kuinka sovellusrungon objektit ovat yhteistyössä keskenään toteutettaessa dokumentti/näkymä-arkkitehtuuria.
- Tiedät, kuinka dokumentti/näkymä-arkkitehtuuria voidaan hyödyntää esitettäessä sovelluksen tietoja tulostuslaitteella kuten näytöllä tai kirjoittimella.

Oppitunnin arvioitu kesto: 40 minuuttia

Dokumentit ja näkymät

MFC:n sovellusrungossa dokumentti on objekti, joka säilyttää sovelluksen tietoja. Näkymä on ruudulla oleva objekti, yleensä näytöllä olevan ikkunan työalueella, jonka avulla käyttäjä on vuorovaikutuksessa dokumentin sisältämän tiedon kanssa.

Tämä sovelluksen tietojen ja niiden visuaalisen esitystavan looginen erottaminen mahdollistaa sovelluksen tietojen liittämisen useisiin eri näkymiin. Esimerkiksi käsitellessäsi Microsoft Word -dokumenttia voit siirtyä normaalinäkymästä rakennenäkymään, asettelunäkymään tai jäsennysnäkymään. Kaikki nämä näkymät perustuvat samaan tietoon — tiedot vain esitetään eri tavoin. Kaikki yhdessä näky-mässä tehdyt muutokset heijastuvat suoraan muihin näkymiin.

Dokumentti/näkymä-arkitehtuurin edut

Dokumentti/näkymä-arkkitehtuuri auttaa sinua esittämään sovelluksen tietoja tietokoneen näytöllä niin, että käyttäjä voi olla niiden kanssa vuorovaikutuksessa. Tämä toiminnallisuus voidaan luoda kokonaan nollapisteestä ylikuormittamalla näkymän piirtofunktiot, hiiren ja näppäimistön toiminnot ja valikkokomentojen käsittely. Vaihtoehtoisesti voit käyttää valmiiksi määriteltyä näkymää,

joka perustuu kontrolliin. Tällaisia vaihtoehtoja käyttäen voit luoda Windowsin Resurssienhallinnan tapaisen sovelluksen, jossa käytetään puurakennetta ja luette-lonäkymää jaetussa ikkunassa. Rich-text edit -kontrolliin perustuvaa Edit Viewiä voidaan käyttää tekstieditorin pohjana.

Dokumentti/näkymä-arkkitehtuuri yksinkertaistaa myös tulostamista ja tulostuksen esikatselua, koska se käyttää samaa piirtologiikkaa (näkymän piirtofunktioon sisältyvää) sovelluksen tietojen esittämiseen näkymässä, tulostuksen esikatselussa ja tulosteissa.

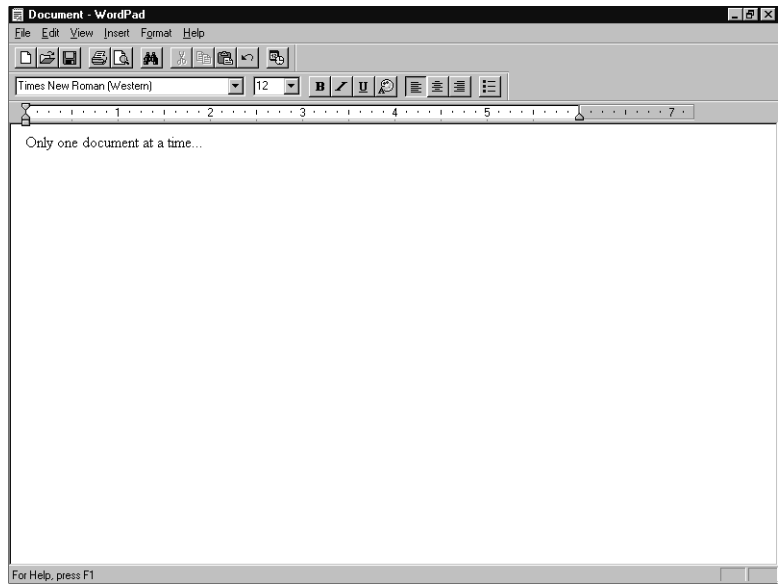
Dokumentti/näkymä-arkkitehtuuri sisältää suurelta osin logiikan, jota tarvitaan tietojen tallentamiseen levylle ja tietojen hakemiseen takaisin muistiin. Tämä tiedon tallennustapa sallii tietojen tallentamisen ja lataamisen sovelluksen omassa objektimuodossa tai missä tahansa MFC:n **CObjec**t-luokasta periytetyssä objektimuodossa. Sovelluksen tietojen serialisointia on käsitelty luvussa 6, *Asetusten tallentaminen ja serialisointi*.

Dokumentti/näkymä-arkkitehtuuri on tärkeä, koska sovellukset, jotka käyttävät dokumentteja ja näkymiä saavat suurimman hyödyn sovelluskehyksestä. Vaikka suurin osa Visual C++:n ja MFC:n toimintojen käsittelystä perustuukin oletukselle, että dokumentti/näkymä-arkkitehtuuria käytetään, voit tehdä MFC-sovelluksia käyttämättä dokumentti/näkymä-arkkitehtuuria. Dokumentti/näkymä-arkkitehtuurin potentiaalisiin etuihin kuuluvat myös mahdollisuus merkittäviin suorituskyvyn parannuksiin ja sovelluksen koon pienenemiseen.

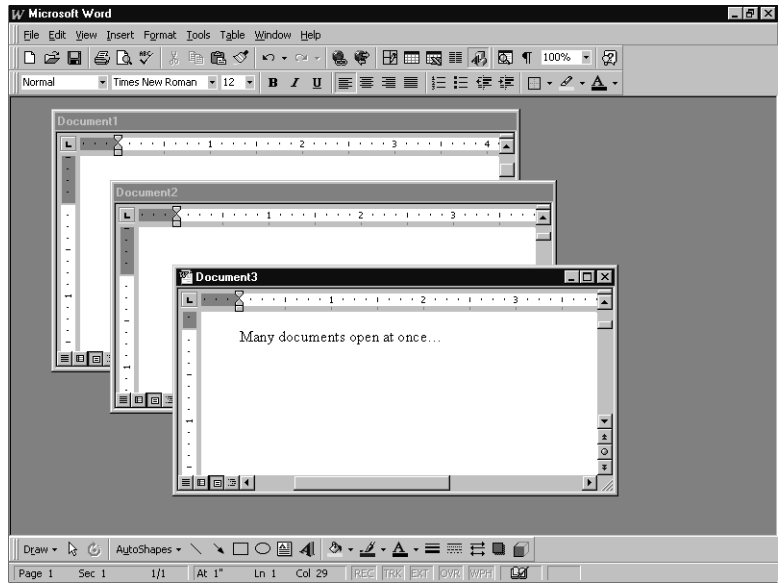
Joissain tapauksissa dokumentti/näkymä-arkkitehtuuri ei sovellu sovelluksellesi. Esimerkiksi sovellus, joka pakkaa tekstitiedostoja, tarvitsee todennäköisesti vain dialogin, jossa kysytään tiedoston nimi ja näytetään prosessin eteneminen. Kanta-ikkunaa ja näkymää ei tarvita, joten dokumentti/näkymä-arkkitehtuurin käyttäminen toisi tässä tapauksessa vain vähän, jos lainkaan, etua. Tässä tapauksessa voisit käyttää AppWizardin dialogipohjaista sovellusrunkoa.

SDI ja MDI -sovellukset

Kuten opit luvussa 2 MFC AppWizard voi yksinkertaisen dialogipohjaisen sovel-luksen lisäksi luoda rungon kahdelle erityyppiselle dokumentti/näkymä-arkkiteh-tuuriin perustuvalle sovellustyypille: single document interface (SDI) ja multiple document interface (MDI). SDI-sovellus sallii vain yhden dokumenttikehyksen näyttämisen kerralla. Windowsin mukana toimitettavat Paint ja Wordpad -ohjelmat ovat esimerkiksi SDI-ohjelmia. MDI sallii useiden dokumentti-ikkunoiden olla avoinna samanaikaisesti samassa sovelluksessa. MDI-sovelluksessa käyttäjä voi avata useita MDI-lapsi-ikkunoita kantaikkunaan. Nämä lapsi-ikkunat ovat kaikki kehysikkunoita, joista jokainen sisältää erillisen dokumentin. Microsoft Word ja Microsoft Excel ovat esimerkkejä MDI-sovelluksista. Näet esimerkin SDI- ja MDI-sovelluksesta kuvissa 3.5 ja 3.6.



Kuva 3.5 SDI-sovellus



Kuva 3.6 MDI-sovellus

Dokumentti/näkymä-arkkitehtuurin objektit

AppWizard luo MFC Dokumentti/näkymä-sovelluksen perusrungon puolestasi. Kuten aina, voit muokata ja laajentaa tätä runkoa niin, että sovellus vastaa omia erityisvaatimuksiasi.

Dokumenttien ja näkymien rungon muodostavat luokat periytetään MFC-kantaluokista **CDocument** ja **CView**. **CWinApp**, **CFrameWnd** ja **CDocTemplate** luokat toimivat yhteistyössä **CDocument** ja **CView** luokkien kanssa, jotta voitaisiin olla varmoja siitä, että kaikki sovelluksen osaset sopivat yhteen.

Taulukossa 3.6 on luettelo dokumentti/näkymäpohjaisen sovelluksen objekteista ja niiden MFC-kantaluokista sekä kuvaus jokaisen objektin tärkeimmistä tehtävistä.

Taulukko 3.6 Dokumentti/näkymä-sovelluksen objektit ja niihen liittyvät MFC-kantaluokat

Objekti	Kantaluokka ja tehtävä
Document	Johdettu luokasta **CDocument**. Sisältää sovelluksen datan.
View	Johdettu luokasta **CView**. Käyttäjän ikkunana tietoon näkymäluokka määrittää tavan, jolla käyttäjä näkee sovelluksesi datan ja kuinka hän sitä käyttää.
Frame window	Johdettu luokasta **CFrameWnd**. Näkymät esitetään dokumenttien kantaikkunoiden sisällä. SDI-sovelluksessa dokumentin kantaikkuna on myös sovelluksen kantaikkuna.
Document template	Johdettu luokasta **CDocTemplate**. Dokumenttimalli ohjaa dokumenttien, näkymien ja dokumenttien kantaikkunoiden luomista. Erityinen dokumenttimalliluokka luo ja käsittelee kaikki avoimet, tietyn tyyppiset dokumentit.
Application	Johdettu luokasta **CWinApp**. Kontrolloi kaikkia tämän tauluko objekteja ja määrittelee sovelluksen toimintaa, kuten alustustoimet ja siivoustoimet.

Dokumentti/näkymä-sovelluksen objektit vastaavat yhteistoiminnallisesti käyttäjän toimiin, komentojen ja muiden sanomien yhteen sitomina.

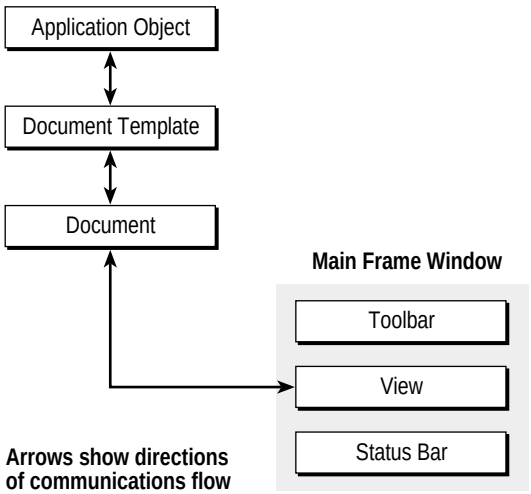
Dokumenttimalliobjektit

Sovellusobjekti luo ja ylläpitää dokumenttimalleja. Yksi **InitInstance()**-funktion tärkeimmistä tehtävistä on yhden tai useamman sopivan tyyppisen dokumenttimallin luominen. Seuraavassa koodissa näet, kuinka tämä tehdää SDI-sovelluksessa:

```
// From CMyAppApp::InitInstance()
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CMyAppDoc),
    RUNTIME_CLASS(CMainFrame),     // main SDI frame window
    RUNTIME_CLASS(CMyAppView));
AddDocTemplate(pDocTemplate);
```

Kun dokumentin malliobjekti luodaan, se liittää dokumenttiluokan yhteen useiden resurssien (kuten valikkojen ja kuvakkeiden), kantaikkunan ja näkymän kanssa. Malli lisätään sovellukseen **CWinApp::AddDocTemplate()**-funktiota käyttäen.

SDI-sovelluksessa käyttäjä tutkii ja käsittelee dokumenttia pääkantaikkunan sisältämän **CFrameWnd**-luokasta periytetyn näkymän kautta. Kuvassa 3.7 näet dokumentti/näkymä-objektien väliset suhteet suoritettavassa SDI-sovelluksessa.



Kuva 3.7 SDI-sovelluksen objektit

MDI-sovellukset käyttävät **CMultiDocTemplate**-objketia, joka pitää luetteloa monista saman tyyppisistä, avoinna olevista dokumenteista. MDI-sovellus käyttää **CMDIFrameWnd**-luokasta periytettyä luokkaa sovelluksen pääikkunan toteu-tuksessa. Näkymät ovat sovelluksen lapsi-ikkunoissa, jotka toteutetaan **CMDIChildWnd**-luokan avulla. MDI-lapsi-ikkuna muistuttaa paljon tavallista kantaikkunaa, paitsi että se sijaitsee MDI-kantaikkunan sisällä. MDI-lapsi-ikku-nalla ei ole omaa valikkoa vaan se jakaa MDI kantaikkunan valikon. Sovellus-runko muuttaa MDI-kantavalikon vastaamaan aktiivista MDI-lapsi-ikkunaa.

Sovelluksen **CMultiDocTemplate**-objektin rakenne on kuvattu seuraavassa koodikatkelmassa.

```
// From CMyMDIAppApp::InitInstance()
      CMultiDocTemplate* pDocTemplate;
      pDocTemplate = new CMultiDocTemplate(
            IDR_MYMDIATYPE,
            RUNTIME_CLASS(CMyMDIAppDoc),
            RUNTIME_CLASS(CChildFrame), // custom MDI child frame
```

```
        RUNTIME_CLASS(CMyMDIAppView));
    AddDocTemplate(pDocTemplate);
```

Dokumenttiobjektit

MFC-dokumentti/näkymä-sovelluksessa dokumenttiobjekti toteutetaan käyttämällä **CDocument**-luokasta periytettyä luokkaa. Tämä luokka lataa, varastoi ja hallitsee sovelluksen tietoja. Se sisältää myös tiedon käsittelyssä tarvittavat funktiot. Jotta säilytettäisiin läheinen yhteys dokumenttien ja näkymien välillä, jokainen dokumenttiobjekti pitää listaa siihen liitetyistä näkymistä. Tätä listaa käsitellään funktioiden **CDocument::GetFirstViewPosition()** ja **CDocument::GetNextView()** avulla.

CDocument-luokka sisältää myös **UpdateAllViews()**-funktion, joka käy läpi kaikki dokumenttiin liittyvät näkymät ja lähettää ilmoituksen jokaiselle, jonka täytyy päivittää ulkoasunsa (kutsumalla **CView::OnUpdate()**-jäsenfunktiota).

Sovelluksesi näkymät tulisi päivittää aina, kun sovelluksen tietoihin tehdään muutoksia, jotka vaikuttavat tapaan, jolla tiedot näytöllä esitetään.

Näkymäobjektit

Näkymäobjekti vastaa fyysisesti sovelluksen työaluetta. Loogisesti se vastaa dokumenttiluokkaan varastoitua tietoa ja vastaanottaa käyttäjän näppäimistöllä ja hiirellä syöttämät tiedot. Kun dokumenttiobjektiin voidaan liittää useita näkymiä, voidaan näkymä liittää vain yhteen dokumenttiin. Sovelluksesi näkymät periytetään **CView**-luokasta, jollet erikseen määrää, että AppWizardin tulee käyt-tää jotain sen erikoistuneista aliluokista. Näitä aliluokkia ovat **CScrollView**, joka on vieritysfunktioilla varustettu näkymät ja **CListView** ja **CTreeView**, joka mah-dollistaa tietojen esittämisen luettelo- ja puukontrolleja käyttämällä.

CView-luokalla on **GetDocument()**-jäsenfunktio, joka toimittaa osoittimen siihen liitettyyn dokumenttiobjektiin.

Piirto-, tulostus- ja esikatseluarkkitehtuuri

Tämän oppitunnin viimeinen osa käsittelee lähemmin **CView**-luokkaa ja sen käyttämistä sovelluksen tietojen visuaaliseen esittämiseen näyttölaitteella kuten näytöllä tai tulostimella.

Piirtopinnat ja GDI

Windowsissa on sovelluksen ja näyttölaitteen välillä yksi abstraktiokerros, josta käytetään nimitystä Graphic Device Interface (GDI). GDI toimii sovelluksille standardi ohjelmointirajapintana, jonka avulla on mahdollista kirjoittaa koodia, joka toimii kaikilla GDI-yhteensopivilla laitteilla.

GDI käsittelee *piirtopinnaksi* (device context) kutsuttua tietorakennetta, joka sisältää tiedon laitteen nykyisistä piirtomäärittelyistä. Määrittelyihin voivat kuulua väripaletti, kirjasinlaji, viivojen piirtämiseen käytetyn kynän paksuus ja alueiden täyttämiseen käytettävän pensselin tyyli. Windows API:in kuuluu monia GDI-funktiota viivojen ja muotojen piirtämistä, alueiden täyttämistä ja tekstin tulostamista varten. Nämä funktiot muodostavat tulosteensa piirtopinnalle — ne ottavat piirtopinnan kahvan parametrinä.

MFC:ssä piirtopinta, samoin kuin useat yleiset GDI-operaatiot, on kapseloitu CDC-luokkaan, joka sisältää piirtopinnan luomiseen ja alustamiseen tarvittavat funktiot, sekä piirtofunktiot, joita voit käyttää sovelluksesi tulosteiden muodostamiseen. CDC:stä on periytetty joukko erityistapauksiin tarkoitettuja luokkia, mukaan lukien taulukossa 3.7 esitetyt.

Taulukko 3.7 CDC:stä periytetyt luokat

Periytetty luokka	Kuvaus
CPaintDC	Kapseloi piirtopinnan, joka on valmistettu työalueen epäkelvon alueen päivittämistä varten vastauksena WM_PAINT-sanomaan. Konstruktori kutsuu **CWnd::BeginPaint()**-funktiota, joka luo piirtopinnan ja valmistelee asiakasalueen graafista tulostusta varten. Tuhoaja kutsuu **CWnd::EndPaint()**-funktiota siivousoperaatioiden suorittamiseksi.
CClientDC	Kapseloi piirtopinnan, joka edustaa vain ikkunan työaluetta.
CWindowDC	Kapseloi piirtopinnan, joka edustaa koko ikkunaa sen kehys mukaan luettuna.
CMetafileDC	Kapseloi Windows metatiedoston piirtämisen. Windows-metatiedosto on kokoelma rakenteita, jotka varastoivat kuvan laitteistoriippumattomaan muotoon.

Näkymään piirtäminen

Sovelluksesi graafisen tulostuksen hoitaa näkymäolion **OnDraw()**-jäsenfunktio. AppWizardin luoma **CView**-luokasta periytetty luokka sisältää **OnDraw()**-funktion tyngän, johon sinun tulee lisätä koodi, joka hoitaa sovelluksesi tietojen näyttämisen. **OnDraw()**-tynkäfunktio näyttää seuraavalta:

```
void CMyAppView::OnDraw(CDC* pDC)
{
    CMyAppDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
}
```

Huomaa, että osoitin CDC-objektiin annetaan **OnDraw()**-funktiolle. Tätä osoitinta käytetään piirtopinnan piirtofunktioiden kutsumiseen sovelluksen tulostetta piirrettäessä. Huomaa myös, että kanta hakee osoittimen myös sovelluksesi dokumenttiolioon. Tätä osoitinta käytetään sovelluksesi tietojen hakemiseen.

Seuraava harjoitus esittelee **OnDraw()**-funktion hyvin yksinkertaisen toteutuksen.

► **OnDraw()-funktion toteutus**

1. Palaa edellisellä oppitunnilla luotuun MyApp-projektiin.
2. Napauta ClassViewissä hiiren kakkospainikkeella **CMyAppDoc**-luokan kuvaketta.
3. Valitse **Add Member Variable** -komento avautuvasta pikavalikotsa.
4. Kirjoita **Add Member Variable** -dialogissa **Variable Type** -ruutuun **CString** ja **m_string Variable Name** -ruutuun.
5. Jätä **Access type** -kohtaan arvo Public ja luo muuttuja napauttamalla **OK**.
6. Avaa **CMyAppDoc** -luokan kuvake ja varmista, että m_string-jäsenmuuttuja on lisätty.
7. Etsi **CMyAppDoc** -muodostimen kuvake, joka jäsenfunktiona sijaitsee **CMyAppDoc**-luokan kuvakkeen alapuolella. Huomaa avainkuvake, joka kertoo kyseessä olevan suojatun jäsenmuuttujan. Aloita muodostimen koon muokkaaminen kaksoisnapauttamalla **CMyAppDoc** -kuvaketta.
8. Lisää seuraava koodirivi muodostimen runkoon:

```
m_string = “Hello World!”;
```

9. Avaa **CMyApp**-luokan kuvake. Siirry muokkaamaan **OnDraw()**-funktion koodia kaksoisnapauttamalla sen kuvaketta.
10. Lisää seuraava koodi **OnDraw()**-funktioon //TODO-kommentin paikalle:

```
CSize TextSize = pDC->GetTextExtent(pDoc->m_string);
CRect rect(10, 10, 10+TextSize.cx, 10+TextSize.cy);
rect.InflateRect(4, 4);

pDC->Rectangle(&rect);
pDC->TextOut(10, 10, pDoc->m_string);
```

11. Käännä ja käynnistä ohjelma. Tarkista, että teksti "Hello World!" näkyy sovelluksen näkymän vasemmassa yläkulmassa olevan nelikulmion sisällä.

Opit lisää sovelluksen tulosteiden näyttämisestä luvussa 5, *Sovelluksen toimintojen toteuttaminen*. Huomioi tässä vaiheessa seuraavat asiat:

- Piirtopinnan osoitinta käytetään kutsuttaessa MFC:n CDC-luokan GDI-jäsenfunktioita sovelluksen tulosteiden näyttämistä varten.
- Dokumenttiosoitinta käytetään sovelluksen tietojen hakemiseen.
- Piirtokoodi on suunniteltu niin, että sovelluksen tietojen muuttuessa ne esitetään aina yhdenmukaisessa muodossa — nelikulmio reunustaa aina siististi tekstiä. Kokeile liittää pidempi merkkijono **CMyAppDoc:: m_string**-jäseneen **CMyAppDoc**-konstruktorissa. Käännä sovellus uudelleen ja suorita se todetaksesi, että tulostus toimii oikein.

OnDraw()-funktion kutsuminen sovelluskehyksestä

Kun sovelluksen tiedot muuttuvat niin, että se vaikuttaa tietojen visuaaliseen esitystapaan, näkymä täytyy piirtää udelleen vastaamaan muutoksia. Tyypillisesti näin käy, kun käyttäjä tekee muutoksen dokumenttiin näkymän kautta. Jokaisen koodinosan, joka muuttaa dokumentin tietoja, tulisi kutsua dokumentin jäsenfunktiota **UpdateAllViews()**, joka ilmoittaa kaikille saman dokumentin näkymille, että niiden tulee päivittää itsensä.

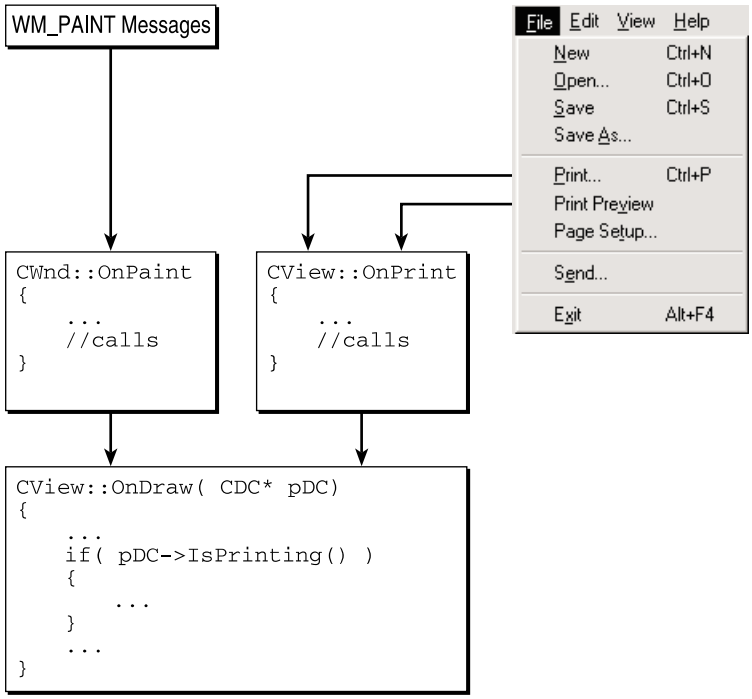
UpdateAllViews() kutsuu jokaisen näkymän **OnUpdate()**-jäsenfunktiota. **OnUpdate()**-funktion perustoteutus merkitsee näkymän koko työalueen epäkelvoksi. Voit muuttaa **OnUpdate()**-funktiota niin, että se merkitsee epäkelvoksi vain muuttuneen alueen.

Kun näkymästä tulee epäkelpo, Windows lähettää WM_PAINT-sanoman. Näkymän oletus **OnPaint()**-käsittelijä vastaa sanomaan luomalla **CPaintDC**-luokan piirtopintaolion ja lähettää sen näkymän **OnDraw()**-jäsenfunktiolle. Olet jo nähnyt, kuinka **OnDraw()**-funktio käyttää tätä piirtopintaoliota tietojesi esittämiseen.

Tulostus ja esikatselu

Windows-ohjelmoinnissa tulostaminen kirjoittimelle on hyvin saman tyyppinen toimenpide kuin tulostamien näytölle. Tämä johtuu siitä, että GDI on laitteistoriippumaton. Voit käyttää samoja GDI-funktioita sekä näytölle että kirjoittimelle tulostettaessa yksinkertaisesti käyttämällä sopivaa piirtopintaa. Jos CDC-olio, jonka **OnDraw()** vastaanottaa, edustaa kirjoitinta, **OnDraw()**-funktion tulostus ohjautuu kirjoittimelle.

Tämä selittää, kuinka MFC-sovellus voi suorittaa yksinkertaisen tulostustyön ilman, että sinun tarvitsee tehdä lisätyötä. Sovellusrunko huolehtii **Print**-dialogin esittämisestä ja tulostinta vastaavan piirtopinnan luomisesta. Kun käyttäjä valitsee **File**-valikosta **Print**-komennon, näkymä lähettää tämän piirtopinnan **OnDraw()**-funktiolle, joka piirtää dokumentin tulostimelle.



Kuva 3.8 **OnDraw()**-funktion kutsumien sovelluskehyksestä

Kirjoittimen ja näytön käytössä on kuitenkin olemassa merkittäviä eroja. Kun tulostat, sinun täytyy jakaa dokumentti erillisiin sivuihin ja näytettävä ne yksi kerrallaan sen sijaan, että näyttäisit dokumentista sen osan, joka sattuu olemaan ikkunassa näkyvillä. Luonnollisesti sinun tulee olla selvillä paperin koosta (onko kysymyksessä A4, letter vai kirjekuori). Voit haluta tulosteen eri suuntiin kuten vaakaan tai pystyyn. **CView**-luokkassa on joukko tulostamiseen liittyviä funktioita, joiden avulla voit toteuttaa tulostusrutiinien logiikan.

Oppitunnin yhteenveto

Sovelluskehys toteuttaa MFC:n dokumentti/näkymä arkkitehtuurin tarjotakseen yhden yhdenmukaisen tavan sovelluksen tietojen ja näiden tietojen esittämisen koordinointiin. Dokumentti on objekti, joka toimii säiliönä sovelluksen tiedoille. Näkymä on ikkunaobjekti, yleensä näytöllä olevan ikkunan työalueeseen liittyvä, jonka avulla käyttäjä on vuorovaikutuksessa dokumentin varastoiman tiedon kanssa.

Dokumentti/näkymä-arkkitehtuuri tarjoaa sovelluskehittäjille monia etuja. Se yksinkertaistaa sovelluksen käyttäjän kanssa vuorovaikutuksessa olevien tietojen esittämistä ruudulla. Tulostaminen ja esikatselu yksinkertaistuvat myös, koska samaa piirtologiikkaa käytetään näytöllä olevissa ikkunoissa, esikatseluikkunassa ja tulostimella. Dokumentti/näkymä arkkitehtuuri suorittaa suuren osan tiedostojen *serialisoinnista* — tiedostojen tallentamisessa levylle ja takaisin muistiin lataamisessa.

MFC:n AppWizardin luomat luokat toimivat yhteistyössä toteuttaessaan dokumentti/näkymä-arkkitehtuurin. Sovellusobjekti sisältää vähintään yhden dokumenttimallin, joka liittää dokumenttiluokan (periytetty MFC-luokasta **CDocument**) joukkoon resursseja, joihin kuuluvat kantaikkunaluokka ja näkymäluokka (periytetty luokasta **CView**).

Dokumentti/näkymä sovellukset voivat käyttää joko yksidokumentista näkymää (SDI) tai monidokumenttista näkymää (MDI). SDI-sovellus sallii vain yhden dokumentin avaamisen kerrallaan sovelluksen pääikkunaan. MDI-sovellus sallii useiden dokumenttien avaamisen yhden sovelluksen ilmentymän lapsi-ikkunoihin.

Tulostuslaitteille, kuten näytöille ja kirjoittimille, piirtämistä käsitellään GDI:n avulla. GDI edustaa standardia ohjelmointirajapintaa, joka mahdollistaa laitteistoriippumattoman graafisen tulostuskoodin kirjoittamisen. GDI käsittelee piirtopinnaksi kutsuttua tietorakennetta, joka sisältää laitteen nykyiset piirtämiseen liittyvät määreet. GDI-piirtofunktiot muodostavat tulostuksensa piirtopinnalle. MFC kapseloi piirtopinnan ja siihen liittyvät GDI-piirtofunktiot **CDC**-luokkaan.

Sovelluksen piirtologiikka toteutetaan näkymän **OnDraw()**-funktion avulla. Sovelluskehys kutsuu **OnDraw()**-funktiota ja välittää sille CDC-olion, joka edustaa nykyistä piirtopintaa. Sinun on lisättävä **OnDraw()**-funktioon koodi, joka suorittaa sovelluksesi dokumenttiolion sisältämien tietojen visuaalisen esittämisen. Samaa piirtokoodia käytetään kuvan muodostamiseen sovelluksen näkymissä, tulostuksen esikatselussa ja tulostimella.

Laboratorio 3: Sovelluksen tietojen esittäminen

Tässä harjoituksessa opit, kuinka käytät MFC AppWizardin luomia perusluokkia dokumentti/näkymä-arkkitehtuurin toteuttamiseen. Käyttämällä laboratoriossa 2 luomaasi STUpload-projektia opit kuinka:

- Käytät dokumentti-luokkaa sovelluksen tietojen varastoimiseen.
- Käytät view-luokan **OnDraw()**-funktiota sovelluksen tietojen esittämiseen näytöllä.

Kuten muistat laboratoriosta 1, StockWatch Data Services vaatii, että tiedot sovellukseen haetaan ASCII-tiedostosta. Tekstitiedoston jokainen rivi sisältää kolme arvoa: osakkeen nimi, päivämäärä ja päätöskurssi. Ensimmäinen asia, joka sinun tulisi tehdä, on luoda tietorakenne, joka on sopiva tiedolle, jota aioit mallintaa. Voit toteuttaa tämän rakenteen osana sovelluksen dokumenttiluokkaa.

Datatiedostoa voidaan ajatella kolmen tietorivin kokoelmana. Siksipä sopivin tapa edetä on seuraava:

1. Tehdään luokka, joka edustaa tietoriviä.
2. Tehdään luokka, joka on tietoriviluokkien kokoelma.

Säästääksemme aikaasi olemme tehneet tietoriviluokan ja tietoriviluokan kokoelman kapseloivan luokan puolestasi. Tietoriviluokan **CStockData** sisältää seuraavat jäsenmuuttujat:

```
double m_dblPrice;
COleDateTime m_date;
CString m_strFund;
```

COleDateTime-luokka on MFC-luokka, joka kapseloi päivämäärän ja ajan. **CString**-luokka on MFC:n vaihtelevan pituinen merkkijonoluokka. Tietoja molemmista luokista saat Visual C++:n ohjeesta.

CStockData-luokan määrittely ja toteutus ovat oheisrompulla tiedostoissa StockData.h ja StockData.cpp \Chapter 3\Code-kansiossa. Tiedostoissa määritellään myös muodostimet, apufunktiot ja luokan vertailuoperaattori, sekä **GetAsString()**-funktio, jonka avulla objektin tiedot voidaan esittää merkkijonoina.

Olemme määritelleet **CStockDataList**-luokan, joka on määritelty tiedostossa \Chapter 3\Code\StockDataList.h, seuraavasti:

```
class CStockDataList : public CList<CStockData, CStockData &>
```

CStockDataList-luokka on periytetty MFC-malliin pohjautuvasta **CList**-kokoomaluokasta. Malli tarvitsee kaksi parametriä: ensimmäinen määrittelee listaan sijoitettavan objektin tyypin ja toinen määrittelee listaan tallennettuun objektiin viitattaessa käytettävän tyypin.

MFC sisältää joukon malleihin perustuvia luokkia, joita käytetään tietojen varastointiin, esimerkiksi **CArray** ja **CMap**. Valitsimme **CList**-luokan, koska se mahdollistaa kohteen lisäämisen listaan mistä tahansa sijainnista. Saat lisätietoja MFC:n kokoomaluokista suorittamalla haun Visual C++ -ohjeessa hakumääreellä "MFC collection classes".

Voidaksesi käyttää **CStockData** ja **CStockDataList**-luokkia, sinun täytyy lisätä niiden lähdekooditiedostot projektiisi.

► **Lähdekooditiedostojen tuominen projektiin**

1. Kopioi tiedostot StockData.h ja StockData.cpp kansiosta \Chapter 3\Code STUpload-kansion työhakemistoon Windowsin Resurssienhallinta ohjelmalla.
2. Avaa laboratoriossa 2 luomasi STUpload-projekti.
3. Siirry Workspace-ikkunassa FileViewiin.
4. Napauta kakkospainikkeella **STUpload Files** -kuvaketta. Valitse pikavalikosta **Add Files to Project**.
5. Siirry **STUpload** hakemistoon (todennäköisesti olet siellä jo valmiiksi). Valitse StockData.h, StockData.cpp, StockSataList.h ja StockDataList.cpp tiedostot ja napauta **OK**. Tiedostot lisätään projektiisi.
6. Siirry Workspace-ikkunassa ClassViewiin. STUpload luokkapuu sisältää nyt **StockData** ja **StockDataList** -luokat. Avaa luokkien kuvakkeet ja tarkastele niiden sisältämiä jäsenmuuttujia ja funktioita.

Nyt sinulla on luokka, jota voidaan käyttää sovelluksen tietojen esittämiseen. **CStockDataList**-luokka edustaa kaikkia tietueita, jotka haetaan StockWatch Data Services yhtiön ASCII tiedostoista. Seuraava tehtävä on luoda sovelluksesi tietoluokan ilmentymä dokumenttiluokkaasi niin, että dokumenttiluokka sisältää sovelluksesi tiedot. Voit tehdä tämän luomalla **CStockDataList** tyyppisen muuttujan **CSTUploadDoc**-luokkaan.

► **m_DocList muuttujan lisääminen CSTUploadDoc luokkaan**

1. Napauta ClassViewissä hiiren kakkospainikkeella **CSTUploadDoc** luokan kuvaketta.
2. Valitse pikavalikosta toiminto **Add Member Variable**.
3. Kirjoita **Add Member Variable** dialogissa **CStockDataList** muokkausruutuun **Variable Type**.
4. Kirjoita **m_DocList** muokkausruutuun **Variable Name**.
5. Aseta suojaukseksi **Protected**. Lisää muuttuja napauttamalla **OK**.
6. Avaa CSTUploadDoc.h-tiedosto kaksoisnapauttamalla **CSTUploadDoc** luokan kuvaketta ClassViewissä ja siirry luokan määrittelyn alkuun.
7. Varmista, että ClassView on lisännyt rivin:

   ```
   #include "StockDataList.h"
   ```

   CSTUploadDoc.h tiedoston alkuun.

Koska lisäsit m_DocList-muuttujan Protected-tyyppisenä jäsenmuuttujana, sinun täytyy lisätä funktio, jonka avulla muuttujan sisältö voidaan lukea.

► **Apufunktion lisääminen suojattua m_DocList-jäsenmuuttujaa varten**

Lisää seuraava rivi **CSTUploadDoc**-luokan määrittelyn **public**-osaan:

```
const CStockDataList & GetDocList() { return m_DocList; }
```

Tässä laboratoriossa ei lisätä funktioita, jotka hakevat tietueita tekstitiedostoista. Nyt kuitenkin lisätään muutamia **CStockData**-tietoja **STUploadDoc:: m_DocList** luetteloon, jotta voidaan nähdä miten sovelluksen tiedot esitetään näytöllä. Nyt lisätään väliaikaisesti muutamia vale-**CStockData**-tietueita **CSTUploadDoc**-luokan muodostimeen. Muodostimen valmis versio on oheisrompulla tiedostossa \Chapter 3\Code\Ch3_01.cpp.

► **Valetietueiden lisääminen**

1. Etsi hakemisto \Chapter 3\Code oheisrompulta käyttämällä Windowsin Resurssienhallintaa.

2. Avaa Ch3_01.cpp-tiedosto Visual C++:aan kaksoisnapauttamalla. Uusi **CSTUploadDoc**-muodostin näyttää seuraavalta:

```
CSTUploadDoc::CSTUploadDoc()
{
    m_DocList.AddTail(CStockData(_T("ARSC"),
        COleDateTime(1999, 4, 1, 0, 0, 0),
        22.33));
    m_DocList.AddTail(CStockData(_T("ARSC"),
        COleDateTime(1999, 4, 2, 0, 0, 0),
        23.44));
    m_DocList.AddTail(CStockData(_T("ARSC"),
        COleDateTime(1999, 4, 3, 0, 0, 0),
        24.55));
    m_DocList.AddTail(CStockData(_T("ARSC"),
        COleDateTime(1999, 4, 4, 0, 0, 0),
        25.66));
    m_DocList.AddTail(CStockData(_T("ARSC"),
        COleDateTime(1999, 4, 5, 0, 0, 0),
        26.77));
}
```

   Koodi lisää viisi **CStockData**-objektia **CSTUploadDoc::m_DocList**-muuttujaan. Maalaa funktio kokonaan ja kopioi koodi leikepöydälle painamalla CTRL+C.

3. Sulje Ch3_01.cpp-tiedosto.
4. Avaa **STUploadDoc**-muodostimen koodi kaksoisnapauttamalla ClassViewissä sen kuvaketta. Nykyisen tyhjän muodostimen pitäisi näyttää seuraavalta:

```
CSTUploadDoc::CSTUploadDoc()
{
    // TODO: add one-time construction code here
}
```

5. Valitse ja poista koko tyhjä muodostin. Liitä tilalle uusi muodostin painamalla CTRL+V.

Havainnollistamista varten toteutetaan seuraavaksi yksinkertainen versio **CSTUploadView::OnDraw()** funktiosta. Teet paljon hienostuneemman version piirtofunktiosta luvussa 5, *Sovelluksen toimintojen toteuttaminen*.

► **CSTUploadView::OnDraw()-funktion toteutus**

1. Etsi hakemisto \Chapter 3\Code oheisrompulta Windowsin Resurssienhallintaa käyttämällä.
2. Avaa Ch3_02.cpp-tiedosto muokattavaksi Visual C++:aan kaksoisnapauttamalla. Uusi **OnDraw()**-funktio näyttää seuraavalta:

```
void CSTUploadView::OnDraw(CDC* pDC)
{
    CSTUploadDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // Save the current state of the device context
    int nDC = pDC->SaveDC();

    // Create font for axis labels
    CFont aFont;

    if(aFont.CreateFont(16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        FF_MODERN, 0))
        pDC->SelectObject(&aFont);

    else
    {
        AfxMessageBox("Unable to create font");
        return;
    }

    const CStockDataList & pData = pDoc->GetDocList();

    int yPos = 10;
    int nTextHeight = pDC->GetTextExtent("A").cy;

    POSITION pos = pData.GetHeadPosition();

    while(pos)
    {
        CStockData sd = pData.GetNext(pos);

        pDC->TextOut(10, yPos, sd.GetAsString());
        yPos += nTextHeight;
    }

    // Restore the original device context
    pDC->RestoreDC(nDC);
}
```

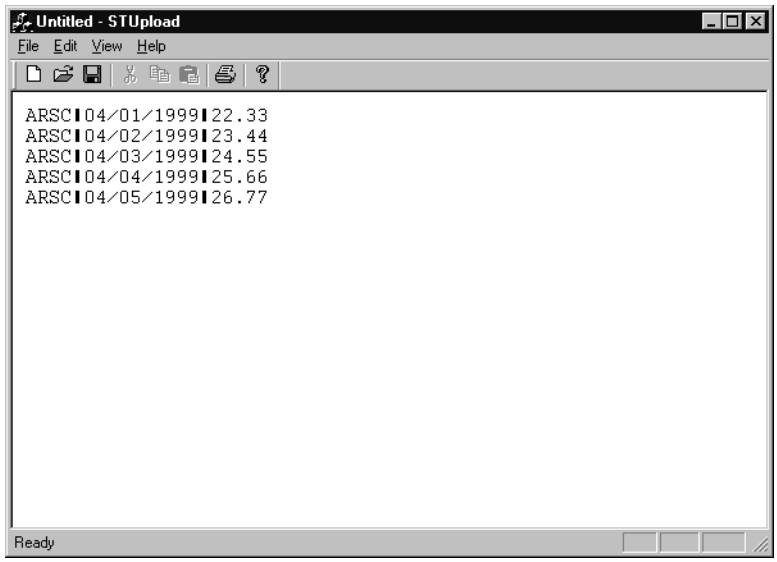
Koodissa käytetään **CList**-luokan funktioita **CSTUploadDoc:: m_DocList** listan jäsenten läpikäymiseen ja tuotetaan **CStockData::GetAsString()** funktion avulla merkkijonon tiedoista, jolloin ne voidaan esittää käyttämällä **CDC::TextOut()**-funktiota. Huomaa, kuinka CDC-funktioita **SaveDC()** ja **RestoreDC()** käytetään laiterajapinnan (device context) tallentamiseen ja palauttamiseen funktion alussa ja lopussa. Tätä käytäntöä kannattaa käyttää aina, kun funktio käsittelee laiterajapintaa. Tämä funktio käsittelee piirtopintaa valitsemalla uuden fontin **CDC::SelectObject()**-funktion avulla.

3. Valitse maalaamalla koko funktio ja kopioi se leikepöydälle painamalla CTRL+C.
4. Sulje Ch3_02.cpp-tiedosto.
5. Siirry ClassViewissä **CSTUploadView::OnDraw()**-funktioon kaksoisnapauttamalla funktion kuvaketta. Nykyisen funktion tulisi näyttää seuraavalta:

```
void CSTUploadView::OnDraw(CDC* pDC)
{
    CSTUploadDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
}
```

6. Valitse ja poista koko **OnDraw()**-funktio. Liitä tilalle uusi versio painamalla CTRL+V.

Voit nyt kääntää ja käynnistää STUpload-sovelluksen. Viiden osakkeen tietojen tulisi tulla näkyviin kuten kuvassa 3.9.



Kuva 3.9 STUpload laboratorion 3 lopussa

Kertaus

1. Kun ohjelmoit sovellusta MFC:tä käyttäen, missä tilanteissa saatat joutua kutsumaan Win32 APIa suoraan?

2. Mitä palveluja **CObject**-luokka tarjoaa?

3. Miksi käyttäisit MFC:n laajennettuja DLL:iä?

4. Minkä MFC-luokan jäsenfunktio toteuttaa sovelluksen sanomasilmukan? Minkä funktio on kyseessä?

5. Minkä tyyppisissä luokissa voidaan sanomakartta toteuttaa?

6. Nimeä neljä Windows-sanomien luokkaa, joita sanomakartat pystyvät käsittelemään?

7. Mikä on dokumenttimalliobjektin tehtävä?

8. Mikä MFC-luokka kätkee GDI:n piirtofunktiot?

9. Kuinka varmistat sen, että sovelluksesi pääikkunassa on aina ajantasainen kuva sovelluksen tiedoista?