L U K U 1 3

Virheiden käsittely, virheiden poisto, ja testaus

Tässä luvussa

Tässä luvussa opiskellaan verheiden käsittelyä ja poistamista sekä testausta. Oikaiseminen näissä elintärkeissä, mutta ei niin hohdokkaissa, viimeisissä vaiheissa on yleensä virheliike; huonosti testatusta sovelluksesta voi koitua huomattavia kustannuksia myyntitulojen menetyksinä, hukattuna tuottoina ja suhteiden huonontumisena käyttäjien ja tuottajien välillä.

Ennen kuin aloitat

Ennen tämän luvun aloittamista sinun tulisi lukea luvut 2-12 ja suorittaa niihin liittyvät harjoituksest. Lisäksi sinun täytyy asentaa Microsoft Visual C++ kehitystyökalut luvun 1 oppitunnilla 2 kerrotulla tavalla.

Oppitunti 1: Virheiden käsittely

Kokeneet ohjelmoijat pitävät *virheiden käsittelyä (*error handling) ohjelmointipro-sessin luonnollisena osana; asian mukainen virheen käsittely on vakaan sovelluksen elintärkeä osa.Tällä oppitunnilla tutustutaan muutamiin yleisimpiin virheen käsittelymenetelmiin. Tällä oppitunnilla puhutaan myös siitä, kuinka tehdä koodia, joka ei vain tunnista virheitä ja ongelmia niiden ilmetessä, vaan osaa reagoida niihin oikein (esimerkiksi näyttämällä viesti-ikkunan käyttäjälle). On paljon helpompaa huomioida virheenkäsittely ohjelmointivaiheessa kuin yrittää lisätä sitä koodiin jälkeen päin.

Tämän oppitunnin jälkeen:

- Osaat kirjoittaa koodia, joka käsittelee virheet tehokkaasti niiden ilmetessä ohjelmaa suoritettaessa.
- Tiedät, mitä ovat poikkeukset ja osaat käyttää poikkeuksenkäsittelytekniikoita.
- Osaat pitää ajantasaista rekisteriä sovelluksen toiminnasta **TRACE**-makroa käyttäen.

Oppitunnin arvioitu kesto: 30 minuuttia

Ohjelman virheiden ennakointi

Useimmat kokeneet ohjelmoijat suhtautuvat terveen epäilevästi siihen, että koodi toimii oletetulla tavalla. Ohjelmassa saattaa olla loogisia virheitä, jotka saavat so-velluksesi toimimaan odottamattomilla tavoilla. Jopa ohjelma, joka on *loogisesti* virheetön voi toimia ennalta arvaamattomasti tietyissä ympäristöissä. Ohjelmoi-dessa kannattaa kehittää tapa jokaisen oletuksen kyseenalaistamisesta. Koeta en-nakoida asiat, jotka voivat mennä pieleen ja lisää koodia, joka hoitaa ongelmat tyylikkäästi. Nämä lisätyt koodirivit voivat olla käytössä harvoin, jos koskaan, mutta ne ovat olemassa, mikäli niitä tarvitaan ongelmien korjaamisessa.

Useat ohjelmoijat epäonnistuvat ympäristön rajoitusten ennakoinnissa. Ajatellaanpa seuraavaa tyypillistä tilanneta: funktio varaa muistista puskurin ja täyttää sen sitten tekstillä. Käyttämällä **new**-operaattoria, funktio saa osoittimen muistilohkoon ja kirjoittaa sinne heti tämän jälkeen. Kehitysvaiheessa funktio toimii virheittä, koska ohjelmoijan tietokoneessa on paljon vapaata muistia käytettävissä. Ohjelman julkistamisen jälkeen ohjelmoija alkaa saada käyttäjiltä kiukkuisia puheluita heidän menetettyään päivän työt ohjelman kaaduttua, koska käyttäjien koneissa ei ollut riittävästi muistia tehtävän suorittamiseen.

Kokeneet ohjelmoijat toimivat varovaisemmin varatessaan muistia. He käyttävät koodia, joka tarkistaa **new**-operaattorin palauttaman osoittimen ja käyttävät

osoitinta vain, jos osoittimella on nollasta poikkeava arvo. Jos **new** palauttaa **NULL** osoittimen — mikä tarkoittaa varauksen epäonnistumista — ohjelman tulisi osata reagoida oikein. Lopputuloksen tulisi muistuttaa seuraavaa koodia:

```
int  *ptr = new int[BLOCK_SIZE];
if (ptr)
{
      // Memory block successfully allocated, so use it
      .
      .
      .
      delete[] ptr;
}
else
{
      // Allocation failed — take appropriate steps
}
```

Potentiaalisten ongelmien ennakoiminen ja sopivien varmistusfunktioiden tekeminen johtavat sovelluksen vakaampaan toimintaan ja suurempaan käyttövarmuuteen hankalissa olosuhteissa, mikä puolestaan laskee virheiden esiintymistiheyttä. Ohjelman vakautta sanotaan *vikasietoisuudeksi*, joka on yhteydessä niiden virheiden määrään, jotka ohjelma pystyy käsittelemään ilman kaatumista. Voit ajatella tätä ennakoivaa lähestymistapaa välittömänä virheen käsittelynä (inline error handling), jossa mahdolliset virheet käsitellään heti koodin sisällä.

Virheiden käsittely paluuarvoja tutkimalla voi tehdä koodista vaikeasti luettavaa, koska ohjelman kulku katkeaa toistuvasti. Tämä virheiden tarkistusmenetelmä voi johtaa pitkiin peräkkäisiin **IF-ELSE** lohkoihin, joissa **IF**-osa sisältää suoritettavaksi tarkoitetun koodin ja **ELSE**-osa sisältää virheiden käsittelyn. Seuraavassa esitetään pseudokoodilla, kuinka tällainen ketjutettu testaus siirtyy kohti ruudun oikeaa reunaa. Jos ketjutetuissa testeissä on pitkiä koodirivejä, koodin toimintaa voi olla vaikea seurata:

```
if (condition1 == TRUE)
{
      // Compute condition2
      if (condition2 == TRUE)
      {
            // Compute condition3
            if (condition3 == TRUE)
            {
                  // Other nested conditions
                  .
                  .
                  .
            }
            else
```

```
            {
                // Failure for condition3
            }
        }
        else
        {
            // Failure for condition2
        }
}
else
{
    // Failure for condition1
}
```

Microsoft Windows ja Visual C++ sisältävät myös toisen tavan välittömään virheiden käsittelyyn: *poikkeuskäsittelyn* (exception handling). Poikkeuskäsittely antaa ohjelmoijalle mahdollisuuden jakaa funktio loogisesti kahteen osaan: yhteen tavalliseen ja toiseen, joka huolehtii virheistä. Virheille altis koodi tavallisessa osassa ei tarkista paluuarvoja ja etenee kuin virheitä ei olisi olemassakaan. Toinen osa nappaa virheet niiden ilmetessä.

Poikkeukset

Poikkeus (exception) on mikä tahansa tilanne, jota käyttöjärjestelmä pitää virheenä. Kun sovellus nostaa poikkeuksen, käyttöjärjestelmä yrittää tiedottaa asiasta virheen aiheuttaneelle sovellukselle kutsumalla sen poikkeuskäsittelijöitä (olettaen, että sellainen on olemassa). Jos sovelluksella ei ole poikkeuskäsittelijää, käyttöjärjestelmä ratkaisee ongelman itse, usein sulkemalla sovelluksen äkisti ja näyttämällä käyttäjälle lyhyen virheilmoituksen tyyliin "This program has performed an illegal operation and will be shut down."

Poikkeuskäsittelyssä on kaksi tasoa:

- **Structured exception handling (SEH)** käsittelee yksinomaan käyttöjärjestelmän virheitä.
- **C++ exception handling** liittyy virheiden käsittelyyn Visual C++-sovelluksissa.

Structured Exception Handling

Vaikka tämä luku käsitteleekin pääasiassa C++:n poikeuskäsittelyä, Structured Exception Handling (SEH) ansaitsee tulla mainituksi kahdesta syystä. Ensinnäkin, vaikka kaksi poikkeusten käsittely menetelmää eroaa selvästi toisistaan, ne usein sekoitetaan toisiinsa. Toiseksi, vaikka C-kieli ei voi hyödyntää C++:n poikkeuskäsittelyä, se voi käyttää SEH:ia.

Kaikki poikkeuskäsittely perustuu SEH-mekanismiin. SEH on sovellustasolle päättyvän kommunikaatioketjun alku. Sovellukseen SEH sisällytetään **__try**,

__except ja **__finally** -avainsanoilla. Yhtä **__try**-lohkoa kohden täytyy olla joko yksi **__except**-lohko tai yksi **__finally**-lohko, mutta ei molempia. Syntaksi on seuraava:

```
__try
{
    // Normal code goes here
}
__except (filter)
{
    // Errors that occur in the try block are trapped here
}
```

Huomaa, että SEH-avainsanoja edeltää kaksi alleviivaa yhden sijasta. **__except**-lohkossa oleva koodi suoritetaan vain, jos **__try** lohkossa olevassa koodissa nousee poikkeus. Jos **__try**-lohko suoritetaan onnistuneesti loppuun, suoritus jatkuu ensimmäisestä **__except**-lohkon jälkeisestä komennosta, joten **__except**-lohko ohitetaan täysin.

__except-lohko määrittää *filter*-parametrin avulla, kykeneekö se käsittelemään poikkeuksen. **filter**-parametrin tulee sisältää yksi taulukossa 13.1 olevista arvoista 13.1.

Taulukko 13.1 Filter-parametrin arvot

Arvo	Merkitys
EXCEPTION_CONTINUE_SEARCH	**__except**-lohko taannuttaa poikkeuksen ja siirtää ohjauksen järjestyksessä seuraavalle edeltäjälleen.
EXCEPTION_CONTINUE_EXECUTION	**__except**-lohko poistaa poikkeuksen ilman toimenpiteitä, palauttaen toiminnan komentoon, joka aiheutti poikkeuksen.
EXCEPTION_EXECUTE_HANDLER	**__except**-lohkon runko suoritetaan.

EXCEPTION_CONTINUE_EXECUTION:n tapainen filteri näyttää ehkä oudolta, jättäähän se **__except**-lohkon suorittamatta. Miksi käyttää poikkeuskäsittelijää, jos sitä ei koskaan suoriteta, vaan itseasiassa pyytään vain tekemään uudelleen poikkeuksen aiheuttaneen komennon? **__except**-lohko yleensä kutsuu apufunktiota, joka antaa *filter*-parametrin paluuarvon, kuten seuraava koodi esittää:

```
__except (GetFilter())
{
    // Body of __except block
}
.
.
.
long GetFilter()
{
```

```
    long lFilter;
    // Determine the filter appropriate for the error
    return lFilter;
}
```

Apufunktion täytyy analysoida nykytilanne ja sen on pyrittävä korjaamaan poikkeuksen aiheuttanut ongelma. Jos se onnistuu, apufunktio palauttaa arvon EXCEPTION_CONTINUE_EXECUTION, aiheuttaen näin **__except**-lohkon sivuuttamisen ja paluun takaisin **__try**-lohkossa olevaan alkuperäiseen komentoon. Tällaista tekniikkaa täytyy kuitenkin käyttää varoen, sillä jos apufunktio ei pystykään todellisuudessa poistamaan ongelmaa, joutuu sovellus ikuiseen silmukkaan, jossa **__try**-lohkossa oleva komento suoritetaan toistuvasti ja se aiheuttaa uudelleen ja uudelleen poikkeuksen, johon ei koskaan saada toimivaa ratkaisua.

Vaikka **__finally**-avainsanan onkin nimellisesti SEH:n osa, sillä on hyvin vähän tekemistä käyttöjärjestelmän kanssa. Ennemminkin se määrittää joukon komentoja, joiden suorituksen kääntäjä takaa **__try**-lohkon suorittamisen päätyessä. Jopa silloin kun **__try**-lohko sisältää **RETURN** tai **GOTO** -komentoja, jotka ulottuvat **__finally**-lohkon ulkopuolelle, kääntäjä varmistaa, että **__finally**-lohko suoritetaan ennen hyppyä tai paluuta. **__finally**-avainsana ei käsittele parametrejä.

Poikkeusten käsittely C++:ssa

C++:n poikkeuskäsittely on komentoketjussa korkeammalla kuin SEH. Siinä missä SEH on järjestelmän palvelu, C++:n poikkeuskäsittely on koodia, jonka sinä kirjoitat toteuttaaksesi kyseisen järjestelmäpalvelun. C++:n poikkeuskäsittely on hienostuneempi ja sisältää useampia vaihtoehtoja kuin SEH. SEH:n matalantason **__try** ja **__except** -avainsanojen käyttöä Visual C++ -sovelluksissa ei suositella.

Samaan tapaan kuin C-sovellukset käyttävät SEH:tä, Visual C++ -sovellukset sisältävät poikkeuskäsittelykoodin, joka suoritetaan virheen tapahduttua. Tämä koodi joko korjaa ongelman ja yrittää virheen aiheuttaneen komennon suorittamista uudelleen, sivuuttaa ongelman täysin tai lähettää ilmoituksen eteen-päin mahdollisten käsittelijöiden ketjussa. C++ sisältää tätä tarkoitusta varten avainsanat **try**, **catch** ja **throw**. Toisin kuin SEH:n vastaavat, nämä avainsanat eivät ala kaksinkertaisella alleviivalla.

Ajatellaan esimerkkinä C++:n tapahtumakäsittelystä taas tilannetta, jossa ohjelma yrittää kirjoittaa muistialueelle **new**-operaattorin epäonnistuttua muistin varaamisessa. Aikaisemmin näimme, kuinka sovellus voi estää virheen syntymisen tarkastamalla palautetun osoittimen arvon. Poikkeuskäsittely tarjoaa jossain määrin hienostuneemman ratkaisun, jossa koodi jaetaan **try** ja **catch** -lohkoihin ketjutettujen **IF-ELSE**-lohkojen sijasta, seuraavassa koodissa esitetyllä tavalla:

```
try
{
      int  *iptr = new int[BLOCK_SIZE];
      .
      .             // If we reach this point, allocation succeeded
      .
      delete[] iptr;
}
catch(CMemoryException* e)
{
      // Allocation failed, so address the problem
      e->Delete();
}
```

Jos muistin varaaminen **new**-komennolla epäonnistuu, se laukaisee poikkeuksen, joka käynnistää **catch**-lohkon suorittamisen. Tässä esimerkissä **catch**-lohko ottaa parametrinä osoittimen MFC:n **CMemoryException**-objektiin, joka sisältää informaatiota **new**-operaattorin aiheuttamasta muisti-loppu-tilanteesta. C++-ohjelmat, jotka eivät käytä MFC:tä, voivat käyttää tähän tarkoitukseen omaa luokkaansa, tai jopa käyttää osoitinta standarditietotyyppiin, kuten merkkijonoon. Lohkon parametriluettelona voi olla myös kolme pistettä (…), mikä kertoo kääntäjälle, että **catch-**lohko käsittelee kaiken tyyppisiä poikkeuksia, eikä rajoitu muistiin liittyviin poikkeuksiin.

Jos ongelmaa ei **catch**-lohkossa pystytä korjaamaan, kokeillaan poikkeuksen aiheuttaneen komennon suorittamista uudelleen **throw**-komennon avulla. Huomaa, että tämä vaihtoehto on joustavampi kuin SEH:n EXCEPTION_CONTINUE_EXECUTION-suodattimen käyttäminen, koska se mahdollistaa **catch**-lohkon suorittamisen. Jos **catch**-lohko ei uudelleen käynnistä (tai *rethrow*) poikkeusta, ohjelman suoritus jatkuu **catch**-lohkoa seuraavasta komennosta.

MFC:n poikkeusmakrot

MFC:n ensimmäiset versiot sisälsivät makroja, jotka korvasivat C++:n poikkeuskäsittelykomennot. Nimiltään ne olivat **TRY**, **CATCH**, ja **THROW**. Useista eri syistä makrojen suosio on vähentynyt; versiosta MFC 3.0 lähtien nämä makrot ovat olleet vain alkuperäisten C++-avainsanojen **try**, **catch** ja **throw** vaihtoehtoisia nimityksiä. Joten vaikka MFC edelleen tunnistaa isoilla kirjaimilla kirjoitetut makronimet niiden käyttöä ei suositella, sillä niistä ei saada minkäänlaista etua.

Hyvälaatuiset poikkeukset

Jotkut poikkeuksista ovat hyvälaatuisia, eivätkä näin ollen esiinny virheinä sovelluksissa. Hyvälaatuinen poikkeus voi esiintyä esimerkiksi tilanteessa, jossa ohjelma koettaa käyttää pinostaan sitomatonta muistialuetta. Käyttöjärjestelmä hoitaa tilanteen läpinäkyvästi pyydystämällä väärän osoitteen ja sitomalla toisen

muistialueen pinoon ja antaen toiminnan jatkua tämän jälkeen sallittua muistialuetta käyttäen. Sovellus ei havaitse syntynyttä poikkeusta. Ainoa ulkoinen viite tällaisesta poikkeuksesta on hetkellinen viive, joka aiheutuu käyttöjärjestelmän toiminnasta.

Virheiden kirjaaminen

Tämän oppitunnin kahdessa ensimmäisessä osassa kerroimme, kuinka virheitä voidaan käsitellä välittömästi niiden ilmetessä. Koska ohjelmoija ei voi ennakoida ja käsitellä kaikkia virheitä, on tärkeää pitää kirjaa odottamattomista virheistä niiden ilmetessä. Tässä osassa kuvataan kolmas menetelmä, jossa suoritettava ohjelma yksinkertaisesti kirjaa syntyneet virheet virhekirjanpitoon (error log) pysähtymättä käsittelemään niitä. Ohjelmoijat voivat näin hakea kirjanpidosta sovelluksen kaatumisen aiheuttaneen virheen ja korjata koodia varmistaakseen, että virhe ei ilmene uudelleen.

MFC-ohjelmissa tämä “offline”-virheidenkäsittely toteutetaan käyttämällä **TRACE**-makroa ja sen muunnelmia. Monet C-ohjelmoijat toteuttavat vastaavan toiminnon käyttämällä soveltaen **printf()**-komentoa koodissaan, saaden näin jatkuvia komentteja sovelluksen suorituksesta lyhyinä viesteinä seuraavaan tapaan:

```
Entering Function1
Allocating memory block
Block successfully allocated
.
.
.
Leaving Function1
```

TRACE-makro kirjaa virheet näyttämällä viestit **AfxDump**:ssa määritellyssä paikassa. Oletuksena käytetään Visual C++:n Output-ikkunan **Debug**-välilehteä. **TRACE** toimii vain projektien debug-käännöksissä (kuvattu tämän luvun oppitunnilla 3). Julkaisukäännöksissä makro ei tee mitään. Koska **TRACE** ei laajenna koodia, se ei lisää julkaisuversioiden kokoa.

TRACE käyttää samoja merkkijonon muotoilukomentoja kuin **printf()**, joten voit esittää muuttujia **TRACE**-rivillä kuten seuraavassa koodissa on esitetty:

```
int iFileSize = 10;
char sz[] = “kilobytes”;
// Display the string “File size is 10 kilobytes”
TRACE(“File size is %d %s\n”, iFileSize, sz);
```

Muotoilun jälkeen tulevan merkkijonon pituus ei loppuun tulevan **NULL**-merkin kanssa saa ylittää 512 merkkiä. MFC sisältää myös makrot **TRACE0**, **TRACE1**, **TRACE2** ja **TRACE3**, jotka ottavat määrätyn määrän muuttujia (0, 1, 2 tai 3).

Ainoa näistä **TRACE** variaatioista saatava hyöty on se, että ne johtavat hieman tiiviimpään koodiin.

Oppitunnin yhteenveto

Tällä oppitunnilla olet oppinut muutamia tapoja, joilla voit ennakoida ongelmia ja kuinka sovellus voi käsitellä virheitä niiden esiintyessä suorituksen aikana.

C++:n poikkeuskäsittely on rakennettu käyttöjärjestelmän rakenteellisen poikkeuskäsittelymekanismin päälle. Visual C++ toteuttaa C++-poikkeuskäsittelyn **try**, **catch** ja **throw** -avainsanojen avulla. Näitä avainsanoja vastaavia MFC-makroja ei enää käytetä C++-ohjelmoinnissa.

Paluuarvojen tarkistaminen on loistava ohjelmointitapa, mutta se voidaan usein korvata poikkeuskäsittelyllä. **TRACE** -komennon avulla voit seurata ohjelman loogista etenemistä ja varmistat, että se toimii niin kuin on ajateltu.

Virheen käsitelyn perusfilosofia voidaan tiivistää kolmeen sananaa: “Älä tee olettamuksia!” Virheillä on tapana ilmetä silloin, kun niitä vähiten odotetaan. Asianmukainen virheenkäsittely antaa sovelluksellesi mahdollisuuden käsitellä odottamattomat tilanteet tyylikkäästi ilman, että sovellus sulkeutuu epänormaalisti — tai vielä pahempaa: jatkaa toimintaansa epävakaassa tilassa.

Oppitunti 2: COM virheet

COM-sovellukset ja komponentit voivat käyttää oppitunnilla 1 käsiteltyjä virheen käsittelytekniikoita, mutta niiden tulee myös hallita COM:n suosimat virheinformaation siirtotavat, joita nimitetään HRESULT-koodeiksi.

Tällä oppitunnilla HRESULT-koodit käsitellään yksityiskohtaisesti ja korostetaan muutamia COM-virheiden käsittelyssä varottavia sudenkuoppia. Pidä mielessä, että COM:n yhteydessä onnistumista ja epäonnistumista ei voi mustavalkoisesti erottaa. COM hyväksyy ajatuksen, jonka mukaan onnistuminen tai epaonnistuminen voi olla myös osittaista.

Opit myös, kuinka COM-komponentti voi ilmoittaa virheestä asiakkaalleen hyödyntämällä **Error-**tapahtumaa, joka on yksi monista COM:ssa määritellyistä varastotapahtumista.

Tämän oppitunnin jälkeen:

- Tiedät, mitä ovat ja kuinka luodaan HRESULT-koodeja COM-ohjelmoinnissa.
- Osaat kompensoida sovellusta, joka ei vastaanota COM-palvelun virheilmoituksia asianmukaisesti.
- Tiedät, milloin käytetään COM:n **Error**-tapahtumaa.

Oppitunnin arvioitu kesto: 15 minuuttia

HRESULT-koodit

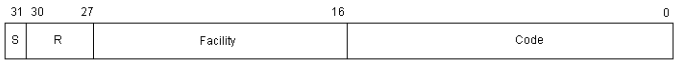
COM:ssa käyttöliittymämetodi ilmaisee onnistumisen tai epäonnistumisen palauttamalla HRESULT-koodin. Huolimatta “H”-etuliitteestään HRESULT-koodi ei ole kahva, vaan pelkkä 32-bittinen arvo, joka ilmaisee onnistumisen tai epäonnistumisen tilan.

COM:ssa on valmiiksi määritelty useita HRESULT-koodeja kuten onnistumista tarkoittava **S_OK**, epäonnistumista merkitsevä **S_FALSE** ja joukko muita virhekoodeja kuten **E_INVALIDARG** ja **E_NOTIMPL**. “E”-etuliite merkitsee virhettä. “S”-etuliite osoittaa, että ensimmäiset kaksi koodia ovat tilakoodeja (status code, SCODE). 32-bittisessä Windowsissa HRESULT ja SCODE tarkoittavat samaa asiaa.

COM antaa sinulle mahdollisuuden määritellä sovelluksissasi uusia HRESULT-arvoja seuraavassa jaksossa kerrotulla tavalla. Sekaannusten välttämiseksi tulisi kuitenkin käyttää COM:ssa ennalta määriteltyjä arvoja aina kun se on mahdollista. Esimerkiksi koodi **E_OUTOFMEMORY**, joka on jo määritelty COM-kirjastossa, ilmaisee yksiselitteisesti yleistä virhetilannetta. Luettelo ennalta määrätyistä HRESULT-arvoista ja niiden merkityksistä on tiedostossa WinError.h otsikon “OLE Error Codes.” alla.

HRESULT-koodin rakenne

HRESULT-koodi rakentuu neljästä bittikentästä kuvassa 13.2 kuvatulla tavalla. Kuvan alla olevassa taulukossa 13.2, kuvataan kentät vasemmalta oikealle.



Kuva 13.1 HRESULT-arvon bittikentät

Taulukko 13.2 Bittikenttien kuvaukset

Tunniste	Bitit	Kuvaus
S	31	Vakavuus. Arvo 0 osoittaa onnistumisen ja arvo 1 ilmaisee virhettä. Koska vakavuusbitti on myös etumerkkibitti, arvo 1 tekee HRESULT koodin arvosta negatiivisen luvun.
R	27-30	Laitteisto. Tulisi olla nolla.
Facility	16-26	Toiminteen koodi, joka ilmoittaa yleisen luokan, johon virhe kuuluu.
Code	0-15	16-bittinen WORD-arvo, joka ilmaisee tilan.

Laitteistokoodit ovat osa COM-määrittelyä. Useimmat laitteisiin liittyvät koodit (esimerkiksi FACILITY_NULL, FACILITY_RPC) on määritelty COM:ssa ja niillä on yleismaailmallinen merkitys. COM-määrittelyssä ohjelmoijan omien virhekoodien, joiden arvot määritellään niissä jäsenfunktioissa, joka ne palauttaa, sijoituspaikaksi on määrätty FACILITY_ ITF-kategoria.

Kun määrität oman HRESULT-koodin, käytä FACILITY_ITF-kategoriaa seuraavaan tapaan:

```
#define MY_SUCCESS_CODE ((0 << 31) | (FACILITY_ITF << 16) | 0x200)
#define E_MY_ERROR_CODE ((1 << 31) | (FACILITY_ITF << 16) | 0x201)
```

WinError.h-tiedostossa FACILITY_ITF määritellään 4:ksi, joten ensimmäinen esimerkki luo positiivisen arvon 0x00040200. Toinen esimerkki tuottaa negatiivisen arvon 0x80040201. WinError.h sisältää myös **MAKE_HRESULT**-makron, joka yksinkertaistaa HRESULT-määrittelyä lennossa:

```
// Set HRESULT value 0x80040202
HRESULT hr = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, 0x202);
```

Kannattaa myös muistaa, että HRESULT-koodien yhteydessä COM olettaa, että arvo nolla edustaa onnistumista, ei epäonnistumista. Esimerkiksi vakion **S_OK** arvo on nolla. Kuten taulukossa 13.1 sanotaan, COM olettaa myös minkä tahansa positiivisen arvon edustavan onnistumista jollain tasolla.

Jotta ohjelmoijan ei tarvitsisi muistaa näitä sääntöjä, COM sisältää kaksi makroa nimiltään **SUCCEEDED** ja **FAILED**, joita usein käytetään HRESULT-koodien testaamiseen. Makrojen käyttö ei ole välttämätöntä, mutta ne tekevät koodista luettavampaa, kuten seuraava koodi osoittaa:

```
HRESULT hr;

hr = pUnk->QueryInterface(IID_SomeObject, (PVOID*) &pObj);
if (SUCCEEDED(hr))
{
    // Use the pObj value
    .
    .
    .
    pObj->Release();
}
```

WinError.h-tiedoston makromäärittelyistä nähdään, että **SUCCEEDED** palauttaa arvon **TRUE** kaikille lausekkeille, joiden arvo on vähintään nolla ja vastaavasti **FAILED** palauttaa arvon **TRUE** kaikille negatiivisille lausekkeille, kuten seuraavassa nähdään:

```
#define SUCCEEDED(Status) ((HRESULT)(Status) >= 0)
#define FAILED(Status) ((HRESULT)(Status) < 0)
```

Kun käsittelet HRESULT-koodia, joka ei ole itse tekemäsi, huolehdi siitä, että koodisi kääntää sen. Voit yleensä luottaa siihen, että kaikilla vakioilla, jotka alkavat etuliitteellä “E”, on negatiivinen arvo; muiden arvoista ei ole mitään takuita. Jos olet epävarma, tarkista vakion numeerinen arvo sopivasta header-tiedostosta.

_com_error-tukiluokka

Visual C++ sisältää **_com_error**-tukiluokan, joka kapseloi HRESULT-koodin ja auttaa piilottamaan taustalla olevan COM-virhekoodin. **_com_error**-luokka sisältää useita jäsenfunktioita, jotka hakevat tietoja kapseloiduista virheistä. Tärkeimmät funktiot on lueteltu seuraavalla sivulla taulukossa 13.3.

Taulukko 13.3 Tärkeimmät _com_error-jäsenfunktiot

Jäsenfunktio	Paluuarvo
Error	HRESULT-koodi, josta **_com_error**-objekti on muodostettu.
ErrorInfo	Osoitin liitettyyn **IErrorInfo**-objektiin (kuvataan myöhemmin), tai NULL, jos **IErrorInfo**:a ei ole. Kutsujan täytyy kutsua **IErrorInfo**-objektin **Release()**-funktiota, kun työskentely objektin kanssa lopetetaan.
Wcode	HRESULT miinus 0x80040200 jos HRESULT käyttää FACILITY_ITF:ä. Muuten funktio palauttaa nollan.
ErrorMessage	**TCHAR**-osoitin järjestelmän sanomaan, joka kuvaa virheen. Jos järjestelmäsanomaa ei ole saatavilla, palautetaan merkkijono "Unknown error" ja HRESULT-arvo heksadesimaalimuodossa.

Jos kapseloituun HRESULT-arvoon liittyy **IErrorInfo**-objekti, **_com_error** voi noutaa virheeseen liittyvät tiedot objektista. **HelpFile()** ja **HelpContext()** -jäsenfunktiot esimerkiksi voivat tutkia **IErrorInfo**-objektia ja palauttaa tiedon virheeseen liittyvästä ohjeaiheesta, jonka kutsuja voi sitten näyttää käyttäjälle tutun Windowsin ohjejärjestelmän kautta.

Koska **_com_error** edustaa poikkeuskäsittelyä, se usein toimii **catch**-lohkoon välitettynä objektina. Seuraavassa on yksinkertainen ohjelma, joka havainnollistaa, kuinka **_com_error**:in avulla napataan virheitä, käyttäen **E_OUTOFMEMORY**-koodia esimerkkinä:

```
#include <comdef.h>
#include <stdio.h>

void main()
{
    try
    {
        _com_error  e(E_OUTOFMEMORY);    // Construct the object
        throw(e);                        // Force an exception
    }

    catch(_com_error& e)
    {
        printf("Error   = %08lx\n", e.Error());
        printf("WCode   = %04x\n",  e.WCode());
        printf("Meaning = %s\n\n",  e.ErrorMessage());
    }
}
```

Ohjelma näyttää seuraavat viestit suorituksensa aikana:

```
Error   = 8007000e
WCode   = 0000
Meaning = Not enough storage is available to complete this operation.
```

COM-asiakkaat ja HRESULT-koodit

Kaiken tämän jälkeen saattaa tuntua pettymykseltä, että useat asiakassovellukset eivät voi edes vastaanottaa palvelimen metodien palauttamia HRESULT-koodeja. Siten huolellisesti muodostetut kuvaavat koodit menevät usein hukkaan. Syyt löytyvät **IDispatch**-rajapintaan luonnostaan kuuluvista rajoitteista.

Luvussa 11 kuvatut MFC-asiakassovellukset esimerkiksi eivät vastaanota HRESULT-koodeja niihin upotetuilta COM-komponenteilta. Tämä johtuu siitä, että Visual C++:n luoma kääreluokka käyttää **IDispatch**-rajapintaa sen sijaan, että kutsuisi metodeja suoraan vtablen kautta. Tämän mutkittelevan palvelinyhteyden aikana **IDispatch** hukkaa metodin paluuarvon, joka ei koskaan päädy asiakkaalle. **IDispatch**:in käyttävä MFC-asiakas voi puuttua virheisiin vain silloin, kun virhe tapahtuu COM-palvelimessa, sillä metodin palauttama negatiivinen HRESULT-arvo nostaa poikkeuksen. Jos palvelimeen yhteydessä oleva koodi sijoitetaan **try**-**catch** lohkoon, asiakas voi reagoida yleisesti virheisiin, vaikka ei voikaan tarkkaan kertoa, mikä virhe syntyi.

Microsoft Visual Basic -asiakas ei toimi sen paremmin. Kuten MFC-asiakaskaan, Visual Basicilla kirjoitettu ohjelma ei pysty vastaanottamaan HRESULT-paluuarvoja. Visual Basic -asiakas voi vain reagoida negatiivisiin virhekoodeihin, yleensä **On Error**-komennolla.

Ennen tämän osuuden loppua tulee mainita vielä muutamia C++-asiakkaita, jotka vastaanottavat HRESULT-paluuarvot asianmukaisesti. Käytettiinpä MFC:tä tai ei, C++:lla kirjoitetun sovelluksen ei tarvitse tukeutua **IDispatch** rajapintaan ja ne voivat saada palvelimen paluuarvot Visual C++:n luoman kääreluokan ohi. Tämä tekniikka ei kuitenkaan kuulu tähän kurssiin.

Error-tapahtuma

Kun COM-komponentin metodi palauttaa HRESULT-koodin asiakas-sovellukselle, sanotaan ilmoituksen olevan *synkroninen.* Synkroninen ilmoitus on osa asiakkaan ja palvelimen välistä normaalia tietovirtaa: asiakas kutsuu palvelinta, palvelin huomaa virheen, palvelin palauttaa HRESULT-koodin, jossa virhe kuvataan ja asiakas reagoi virheeseen. Nämä vaiheet etenevät järjestyksessä; asiakas saa tiedon virheestä vasta komponentin lopetettua toimintansa ja annetua paluuarvon.

Joissain tapauksissa komponentti saattaa törmätä virheeseen, josta asiakasta pitäisi välittömästi informoida. Tällaisessa tapauksessa komponentti voi laukaista tapahtuman ja jatkaa toimintaansa. Toisessa tapauksessa komponentilla saattaa olla työsäikeitä, jotka toimivat asiakkaan kanssa samanaikaisesti. Jos työsäie havaitsee virheen, sen ainoa keino ilmoittaa asiasta asiakkaalle on tapahtuman laukaiseminen. COM määrittelee **Error**-nimisen tapahtuman, joka mahdollistaa tällaisen asynkronisen tiedottamisen.

Error-tapahtumalla on ennalta määrätty lähetetunniste **DISPID_ERROREVENT**. MFC:tä käyttävä ja **COleControl**:sta periytetty ActiveX-kontrolli voi käyttää hyväkseen **FireError()**-jäsenfunktiota.

Oppitunnin yhteenveto

Tällä oppitunnilla kerrottiin, kuinka käsitellään COM:n virhekoodeja, erityisesti kiinnitettiin huomiota HRESULT-koodien luomiseen ja käyttöön palvelimissa kuten ActiveX-kontrolleissa. HRESULT-koodit eivät ole kahvoja, vaan yksinkertaisia 32-bittisiä arvoja, jotka koostuvat neljästä bittikentästä. Kentät kuvaavat virheen vakavuuden, toiminnekategorian ja sen koodien kategoriassa. Negatiivinen HRESULT-koodi osoittaa virhettä ja aiheuttaa poikkeuksen MFC ja Visual Basic -asiakkaissa. HRESULT- ja SCODE-tyyppimäärittelyt ovat samat 32-bittisissä Windows-ohjelmissa.

_com_error-tukiluokka helpottaa COM:n käsittelyä kapseloimalla HRESULT-arvot. **_com_error**-objekti välitetään usein poikkeusparametrina **catch**-lohkolle.

Vaikka COM-palvelimien tekijöiden velvollisuus onkin varmistaa, että heidän tuotteensa palauttavat tarkat ja kuvaavat virhekoodit, useimmat asiakasohjelmat eivät pysty vastaanottamaan noita koodeja. Tähän tulee varmasti tulevaisuudessa muutos.

Error-tapahtuma antaa COM-komponenteille keinon ilmoittaa havaitsemistaan virheistä välittömästi asiakkaalle. **Error**-tapahtumaa tulisi käyttää asynkronisissa ohjelmointitilanteissa, joissa on joko mahdotonta tai epäkäytännöllistä käyttää HRESULT-paluuarvoa virheistä ilmoittamiseen.

Oppitunti 3: Johdatus virheiden poistoon

Tämä oppitunti sisältää Visual C++:n integroidun debuggerin yleisen esittelyn. Laajan debuggaus-aiheen käsittely on parasta aloittaa yleiskatsauksella sen sijaan, että heti uppouduttaisiin debuggerin yksityiskohtiin. Kun on saatu varma käsitys debuggerin tarkoituksesta samoin kuin sen vahvuuksista ja heikkouksista, olet valmis aloittamaan debuggerin käytön. Käytöstä on kerrottu seuraavalla oppitunnilla.

Tämän oppitunnin jälkeen:

- Tunnet debuggerin toiminnan yleiset periaatteet.
- Tunnet Visual C++:n debuggerin erityisominaisuudet.
- Osaat käyttää MFC-frameworkin virheiden poistamista helpottavia makroja.

Oppitunnin arvioitu kesto: 20 minuuttia

Mikä on debuggeri?

Debuggaus on yleisnimitys kaikelle sovelluksen virheiden etsimiseen ja korjaamiseen liittyvälle työlle. Debuggauksen hallitseminen on onnistuneen ohjelmistotyön edellytys. Debuggeri on sovellus, joka helpottaa virheiden korjausprosessia. Debuggeri ajaa ohjelmaa tiukassa tarkkailussa ja se voi pysäyttää toiminnot koska vain antaakseen sinulle mahdollisuuden tarkistaa ohjelman suoritusaikaisen tilan.

Visual C++ -debuggeri kuuluu Visual Studio -ympäristöön ja sillä on omat valikko- ja työkalurivikomentonsa. Ennen kuin voit käyttää debuggeria, sinun täytyy tehdä projektistasi debug-käännös. Et voi käyttää debuggeria ennen kuin koodi on saatu käännettyä suoritettavaksi tiedostoksi ilman syntaksivirheitä.

► **Debug-käännöksen tekeminen**

1. Napauta hiiren oikealla painikkeella työkalurivialuetta ja ota esille täysi **Build**-työkalurivi (**Build**-minirivi ei käy). Varmista, että **Build**-työkalurivillä Win32 Debug-käännösasetukset on tehty kuten kuvassa 13.2.

Kuva 13.2 Win32 Debug-käännöksen asetukset

2. Napauta samalla työkalurivillä olevaa **Build**-painiketta.

Virheiden poisto-operaatio koostuu useista vaiheista, tyypillisesti aloitetaan tekstieditorista. Debuggauksen aluksi nimeä kaatuvan ohjelman lähdekoodista

osa, jossa arvelet virheen syntyvän ja merkitse sitten tämän osan ensimmäinen komento. Käynnistä debuggeri, joka suorittaa ohjelmaa, kunnes vuoroon tulee merkintä, jonka asetit kyseisen osan alkuun. Kun debuggeri keskeyttää ohjelman suorituksen, voit suorittaa komentoja yksi kerrallaan. Tämä toiminto antaa mahdollisuuden tarkistaa sovelluksen muuttujien arvot sen ollessa pysähdyksissä ja voit varmistaa, että sovelluksen toiminta etenee odotetulla tavalla.

Debug- ja julkaisuversio

Tuotteen debugversio on versio, jonka kanssa työskentelet kehitystyön ja testauksen aikana, kun pyrit tekemään ohjelmasta virheetöntä. Debugversio sisältää symbolista informaatiota, jonka kääntäjä sijoittaa objektitiedostoon. Lukemalla alkuperäistä lähdetiedostoa ja projektin symbolista informaatiota debuggeri voi yhdistää lähdetiedoston jokaisen rivin vastaaviin binaarikomentoihin suoritettavassa tiedostossa. Debuggeri suorittaa käännettyä tiedostoa, mutta näyttää etenemisen lähdekoodissa.

Huomio Kääntäjän optimoima koodi saattaa johtaa debuggerin odottamattomaan toimintaan. Jos odottamattomia virheitä esiintyy seurattaessa ohjelmaa debuggerissa, tarkista, että projektisi debug-asetuksissa on kääntäjän optimointi asetukset otettu pois käytöstä. Optimoinnin asetukset ovat **Project Settings** -dialogin **C/C++**-sivulla.

Tuotteen julkaisuversio on tarkemmin käännetty versio, joka jaellaan asiakkaille. Julkaisuversion käännetty tiedosto sisältää vain suoritettavat komennot kääntäjän optimoimassa muodossa, ilman symbolista informaatiota. Voit käynnistää julkaisuversion debuggerissa, mutta jos teet niin, saat kuitenkin ilmoituksen, että tie-dostossa ei ole symbolisia tietoja. Vastaavasti voit ajaa sovelluksen debug-version ilman debuggeria. Näillä suoritustavoilla on käytännön merkitystä Visual C++:n ominaisuudelle, joka tunnetaan nimellä just-in-time debugging. Jos ohjelma kohtaa poikkeuksen, jota se ei käsittele, järjestelmän SEH-mekanismi ohjaa suorituksen Visual C++:lle, joka puolestaan käynnistää debuggerin. Debuggeri näyttää komennon, joka aiheutti virheen, sekä muuttujien arvot sellaisina kuin ne olivat tapahtumahetkellä.

MFC:n debuggausmakrot

MFC sisältää useita makroja, jotka helpottavat debuggausta ja vaikuttavat vain debug-käännöksiin. Olemme jo tutkineet **TRACE**-makroa ja sen muunnelmia, ja olemme nähneet, kuinka **TRACE**-makroa käytetään sovelluksen toimintojen ja ti-lojen kirjaamiseen. Tässä osassa käsitellään muutamia muita MFC-makroja, joihin kuuluvat **ASSERT** ja sen muunnelmat **VERIFY** ja **DEBUG_NEW**.

ASSERT-makro on miellyttävä tapa testata tehtyjä oletuksia sovelluksen kehitysvaiheessa ilman pysyvän virheentarkistuskoodin lisäämistä. Voit esimerkiksi

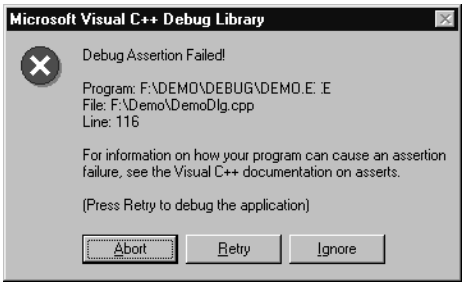
haluta varmistaa, että osoittimen arvo on nollasta poikkeava ennen kuin sen käyttämistä. Yksi tapa tämän tekemiseen on erityisen **if**-testin käyttö, kuten seuraavassa koodissa havainnollistetaan. Funktio saa parametrinä **CButton**-osoittimen:

```
void SomeFunction(CButton* pbutton)
{
    if (pbutton)
    {
        // Use the pbutton pointer
        ...
    }
}
```

Vaikka tällaiset tarkistukset kuuluvatkin hyviin ohjelmointikäytäntöihin, joiden avulla vältetään mahdollisia ongelmia, niitä ei välttämättä haluta enää sovelluksen julkaisuversioon. Satojen tällaisten tarkastusten lisääminen eripuolille koodia voi lisätä tarpeetonta koodia lopputuotteeseen — koodia, joka lisää ohjelman kokoa ja hidastaa sen toimintaa. Tämä tarpeeton koodi voidaan välttää käyttämällä **ASSERT**-makroa, seuraavassa koodissa kuvatulla tavalla:

```
void SomeFunction(CButton* pbutton)
{
    ASSERT(pbutton);  // Make sure that pbutton is non-zero
    // Use the pbutton pointer
    ...
}
```

Jos **ASSERT**:lle välitetty lauseke tuottaa nollan, ohjelma pysähtyy antaen järjestelmäviestin, jossa kerrotaan ongelmasta. Kuten kuvassa 13.3 nähdään, viesti antaa sinun valita vaihtoehdoista: sovelluksen sulkeminen, ongelman sivuuttaminen ja lisätietojen hankkiminen sovellusta debuggaamalla.



Kuva 13.3 ASSERT:n vikailmoitus

ASSERT-makro tuottaa koodia vain sovelluksen debug-versioon, eikä jätä jälkiä julkaisuversioon. Tästä syystä voit lisätä satoja **ASSERT**-lausekkeita koodiisi ilman, että lopputuotteen koko kasvaa. Jos haluat sisällyttää tarkistuksen sekä

debug että julkaisuversioon, käytä **VERIFY** makroa, joka toimii riippumatta käännöksen asetuksista.

ASSERTia voidaan käyttää kaikkien Boolean-muotoisen tuloksen antavien lauseiden kuten **ASSERT**(x < y) ja **ASSERT**(i > 10) kanssa, mutta sen muunnokset ovat erikoistuneet pidemmälle. Samantyylinen **ASSERT_VALID**-makro esimerkiksi, toimii vain MFC **CObject**-luokasta periytettyjen objektien osoittimien kanssa. Tämä makro tekee aivan samat toimet kuin **ASSERT**, mutta se tarkistaa lisäksi objektin kutsumalla lisäksi objektin **AssertValid**-jäsenfunktiota.

ASSERT_KINDOF-makro toimii vain **CObject**-luokasta periytettyjen objektien osoittimien yhteydessä. **ASSERT_KINDOF** tutkii, että osoitin edustaa määrätyn luokan objektia tai tästä luokasta periytetyn luokan objektia. Tämä rivi esimerkiksi varmistaa, että **pDoc** osoittaa objektiin, joka on periytetty **CDocument**:sta:

```
ASSERT_KINDOF(CMyDocument, pDoc)
```

DEBUG_NEW-makro auttaa löytämään muistivuodot, joita syntyy, kun **new**-lausetta ei ole yhdistetty oikein vastaavaan **delete**-lauseeseen. Makroa on helppo käyttää ja se vaatii vain yhden rivin lisäämisen lähdekoodiin:

```
#define new DEBUG_NEW
```

Kun tämä määrittely on asetettu, kääntäjä kääntää koodin jokaisen **new**-komennon muotoon **DEBUG_NEW** ja MFC huolehtii lopusta. Debug-käännöksissä **DEBUG_NEW** pitää lukua tiedostoista ja riveistä, joissa **new**-komentoja esiintyy. Ohjelmassa voidaan tämän jälkeen kutsua **CMemoryState:: DumpAllObjectsSince()**-jäsenfunktiota, joka näyttää luettelon kaikista **new**-varauksista Visual C++:n Output-ikkunassa, yhdessä rivinumeron ja tiedostonimen kanssa. Tällainen tieto helpottaa vuotojen osoittamista. Julkaisukäännöksessä **DEBUG_NEW** palautuu muotoon **new**.

Oppitunnin yhteenveto

Tällä oppitunnilla esiteltiin debuggauksen tärkeää osuutta, jossa haetaan ja korjataan sovelluksen loogisia virheitä. Debuggerin toiminta käytiin läpi yleisellä tasolla ja kuvattiin projektin debug- ja julkaisuversioiden ero. Lisäksi tutustuttiin muutamiin MFC:n tarjoamiin makroihin, joita voidaan hyödyntää debuggauksessa.

ASSERT-makro on tehokas keino virheiden havaitsemiseen, kun virheen seurannasta ei haluta sovelluksen pysyvää osaa. Sen muunnoksia **ASSERT_VALID** ja **ASSERT_KINDOF**-makroja käytetään MFC:n **CObject**-luokasta periytettyjen objektien osoittimien tarkistamiseen. **ASSERT**-makro toimii sovelluksen debug-käännöksessä. **VERIFY**-makroa voidaan käyttää sekä debug- että julkaisuversiossa. **DEBUG_NEW**-makron avulla löydetään tilanteet, joissa muistia varataan käyttöön vapauttamatta sitä myöhemmin.

Oppitunti 4: Integroidun debuggerin käyttö

Varustettuna oppitunnin 3 antamilla perustiedoilla olet nyt valmis tutustumaan Visual C++:n debuggeriin. Kuten tulet näkemään, debuggeri helpottaa virheen korjausprosessia huomattavasti ja sen avulla pystyt löytämään lähes minkä tahansa bugin, johon Windows-ohjelmoinnissa saattaa törmätä.

Tämän oppitunnin jälkeen:

- Osaat käyttää Visual C++:n integroitua debuggeria ja ymmärrät debuggerin ikkunoissa näkyvän lisäinformaation.
- Osaat tarkkailla ajettavaa ohjelmaa ja tiedät kuinka asetetaan keskeytyskohtia, jotka keskeyttävät ohjelman.
- Tiedät, kuinka sovellusta voidaan ajaa komennoittain.
- Osaat käyttää debuggerin **Edit and Continue** -toimintoa, jonka avulla voidaan virheitä korjata lennossa debuggauksen aikana.
- Osaat etsiä virheitä COM-pohjaisista sovelluksista ja komponenteista.
- Osaat tutkia ActiveX-kontrolleja Test Container -apuohjelmalla.

Oppitunnin arvioitu kesto: 40 minuuttia

Keskeytyskohdat

Debuggeri ei pysäytä debugattavaa ohjelmaa. Pikemminkin ohjelma itse pysäyttää itsensä tavatessaan tekstieditorissa asetetun merkin. Merkkiä kutsutaan *keskeytyskohdaksi* (breakpoint). Kun ohjelmaa ajetaan, debuggeri nukkuu. Ohjaus palautuu sille ohjelman törmättyä keskeytyskohtaan.

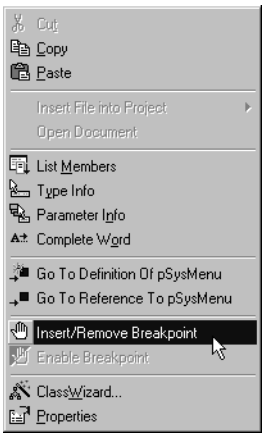
Debuggeri tunnistaa kahdenlaisia keskeytyskohtia, sijaintiin perustuvia ja sovelluksen tietoihin perustuvia. *Sijaintiin perustuva keskeytyskohta* (location breakpoint) on johonkin tiettyyn ohjelman komentoon liitetty merkintä, samaan tapaan kuin kirjanmerkki Visual C++ -tekstieditorissa. *Arvoon perustuva keskeytyskohta* (data breakpoint) seuraa koodin sijasta tietoja. Arvoon perustuvaa keskeytyskohtaa käytetään, kun halutaan keskeyttää sovelluksen suoritus, kun muuttujan arvo vaihtuu. Arvoon perustuva keskeytys on käytännöllinen tutkittaessa muuttujaa, joka saa suorituksen aikana väärän arvon, mutta ei tarkkaan tiedetä milloin se tapahtuu. Arvoon perustuva keskeytyskohta keskeyttää sovelluksen suorittamisen muuttujan arvon vaihtuessa tai sen saadessa tietyn arvon (kun esimerkiksi osoitin kiinnitetään uudelleen tai kun muuttuja *x* ylittää arvon 500).

Keskeytyskohtien asettaminen

Debuggaus on parasta aloittaa tekstieditorilla. Paikkaan perustuvat keskeytyskohdat ovat ehdottomasti eniten käytettyjä keskeytyskohtia ja niiden asettaminen Visual C++ -debuggerissa on helppoa. Tarvitset vain yleiskäsityksen siitä, missä sovelluksesi toimii väärin.

► **Sijaintiin perustuvan keskeytyskohdan asettaminen**

1. Avaa ohjelman lähdekooditiedostot ja etsi paikka, jossa haluat keskeyttää sovelluksen toiminnan.
2. Sijoita kohdistin riville napauttamalla hiirellä ja aseta keskeytyskohta painamalla F9. Editori merkitsee rivin sijoittamalla pienen punaisen pallon rivin kohdalle vasempaan marginaaliin.
3. Voit poistaa keskeytyskohdan painamalla uudelleen F9.
4. Voit asettaa ja poistaa keskeytyskohdan myös napauttamalla riviä hiiren oikealla painikkeella. Tee valinta avautuvasta pikavalikosta (kuvassa 13.4), napauttamalla **Insert/Remove Breakpoint** -komentoa.

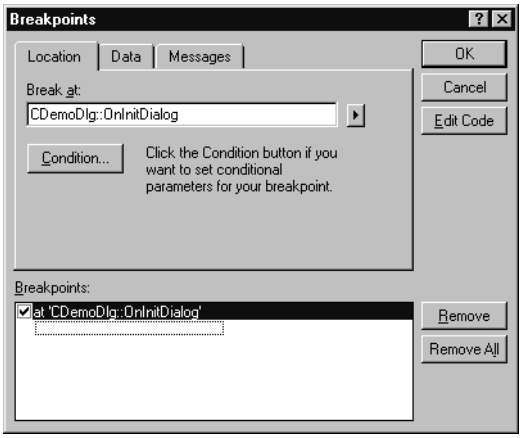


Kuva 13.4 Insert/Remove Breakpoint -komento

Voit asettaa keskeytyskohdan myös **Breakpoints**-dialogissa, vaikka se ei olekaan yhtä käytännöllistä. Tätä dialogia täytyy käyttää asetettaessa arvoon perustuvaa keskeytyskohtaa ja kahta muuta keskeytysversioita: ehtoon perustuvaa (conditional breakpoints) ja sanomiin perustuvaa (message breakpoints). Ehtoon perustuva keskeytys laukeaa heti, kun muuttujalle asetetaan tietty arvo. Sanomaan perustuva keskeytys taas laukeaa ikkunan vastaanotettua määrätyn sanoman.

Breakpoints Dialogi

Avaa kuvassa 13.5 näkyvä **Breakpoints** dialogi, painamalla CTRL+B tai napauttamalla **Breakpoints**-komentoa **Edit**-valikossa. Dialogin kolme välilehteä antavat sinulle mahdollisuuden asettaa paikkaan, arvoon tai sanomaan perustuvia keskeytyskohtia.



Kuva 13.5 Breakpoints-dialogi

Location-välilehti

Breakpoints dialogi tarjoaa useita mahdollisuuksia usein hyödyllisten, sijaintiin perustuvien keskeytyskohtien asettamiseen. Voit esimerkiksi kirjoittaa funktion nimen **Break At** -muokkausruutuun ja asettaa näin keskeytyskohdan funktion ensimmäiselle riville, tai kirjoittaa sen tunnuksen nimen, jonka kohdalle keskeytyskohta halutaan asettaa. Teksti **Break At** -kontrollissa erottelee isot ja pienet kirjaimet toisistaan, joten sen täytyy vastata täysin tunnusta. C++-funktion nimen täytyy sisältää luokan nimen lisäksi sitä soveltavan operaattorin nimi. Joten merkintä **OnInitDialog()** ei ole riittävä, mutta **CDemoDlg:: OnInitDialog()** on.

Data-välilehti

Breakpoints-dialogi on ainoa tapa, jolla arvoon perustuva keskeytyskohta voidaan asettaa. Kirjoita **Breakpoints**-dialogin **Data-**välilehdelle sen muuttujan tai lausekkeen nimi, jota haluat debuggerin seuraavan. Kirjoita lausekkeet normaaleina C/C++-ehtolauseina, kuten seuraavassa:

```
1 == 100 or nCount > 25
```

Debuggeri osaa seurata osoittimella määrättyä muuttujajoukkoa, kuten taulukkoa tai struktuuria.

► **Arvoon perustuvan keskeytyskohdan asettaminen taulukolle tai struktuurille**

1. Napauta **Breakpoints**-dialogin **Data**-välilehteä.
2. Taulukkoa asetettaessa krijoita nimeksi taulukon nimi ja sen perään **[0]** — esimerkiksi **iArray[0]**.
3. **Enter the number of elements in an array or structure** -muokkausruutuun asetetaan tarkkailtavien elementtien määrä, jolloin päästään ensimmäisen arvon lisäksi kiinni myös taulukon muihin arvoihin. Huomaa, että tässä käytetään nimenomaan elementtien määrää, ei tavujen määrää.
4. Halutessasi asettaa tietoon perustuvan keskeytyskohdan struktuuriin, aseta nimeksi osoitinmuuttuja, esimerkiksi **pStruct** ja sijoita sen eteen tähtimerkki.

Vastaavasti voit viitata merkkien muodostamaan jonoon, johon muuttuja pString osoittaa kirjoittamalla **Enter The Expression** -muokkausruutuun ***pString**. Kirjoita **Enter the number of elements** -muokkausruutuun tarkkailtavien tavujen määrä. Arvo **pString** ilman tähtimerkkiä asettaa keskeytyskohdan laukeamaan vain, jos pString siirretään osoittamaan muualle. Tässä tapauksessa debuggeri seuraa pString-muuttujaa itseään, ei sen osoittaman merkkijonon arvoa.

Ohjelmasi suoritus saattaa hidastua huomattavasti, jos asetat yli neljä arvoon perustuvaa keskeytyskohtaa tai jos jokin seurattavista muuttujista sijaitsee pinossa.

Ehdolliset keskeytyskohdat

Debuggeri reagoi ehdolliseen keskeytyskohtaan vain, jos määrätty ehto on arvoltaan TRUE, kun saavutaan merkittyyn komentoon. Joka kerran, kun ehdolliseksi keskeytyskohdaksi merkitty komento suoritetaan, debuggeri tarkistaa lausekkeen arvon ja keskeyttää ohjelman etenemisen vain, jos lausekkeen arvo ei ole nolla.

► **Ehdollisen keskeytyskohdan asettaminen**

1. Napauta **Breakpoints**-dialogin **Location**-välilehteä.
2. Määritä lähdekoodista komento, johon keskeytyskohta asetetaan.
3. Avaa **Breakpoint Condition** -dialogi napauttamalla **Condition**-painiketta (kuvassa 13.5).
4. Kirjoita **Enter the expression to be evaluated** -muokkausruutuun ehto C/C++-ehtolauseena.

Sanomiin perustuvat keskeytyskohdat

Sanomiin perustuvat keskeytyskohdat liittyvät ikkunaproseduuriin. Suoritus pysähtyy, kun ikkuna ottaa vastaan tietyn sanoman, kuten WM_SIZE tai WM_COMMAND. Sanomiin perustuvat keskeytyskohdat eivät välttämättä ole hyödyllisiä MFC:tä hyödyntävissä C++-ohjelmissa, koska proseduurit ovat usein MFC frameworkin sisällä. Saat tietyn sanoman kiinni MFC-sovelluksissa asettamalla sijaintiin perustuvan keskeytyskohdan funktioon, joka toimii sanoman käsittelijänä. Funktio on nimetty luokan sanomakartassa.

Debuggerin käynnistäminen

Kun olet tehnyt debug-käännöksen ja olet määrännyt missä ja millaisissa tilanteissa sovellus pysäytetään, voit käynnistää ohjelman.

► **Debuggerin käynnistys**

1. Napauta **Build**-valikosta **Start Debug**. Saat neljä vaihtoehtoa: **Go**, **Step Into**, **Run to Cursor** ja **Attach to Process**.
2. Kun olet asettanut vähintään yhden keskeytyskohdan lähdekoodiin, napauta **Go**. Debuggeri käynnistää ohjelman suorituksen, joka jatkuu, kunnes tavataan paikkaan perustuva keskeytyskohta tai tietoon perustuvan keskeytyskohdan määritys täyttyy.
3. Käynnistä ohjelman suoritus napauttamalla **Step Into**, jolloin suoritus pysähtyy ensimmäiseen komentoon.
4. Saat ohjelman suorituksen pysähtymään riville, jossa osoitin on, valitsemalla komennon **Run to Cursor.** Jos yhtään lähdetiedostoa ei ole auki, **Run to Cursor** -komento ei ole käytettävissä. Muuten se antaa mahdollisuuden hypätä nopeasti ohjelmaan ilman, että keskeytyskohtia on asetettu.
5. Käynnistä debuggeri **Attach to Process** -komennolla, kun haluat liittää sen ohjelmaan, joka on juuri suoritettavana.

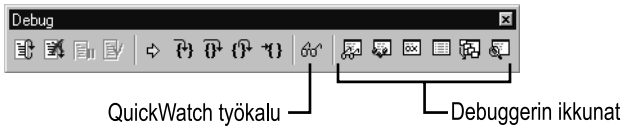
Debuggerissa on pikanäppäimet kolmelle ensimmäiselle **Start Debug:in** alikomennolle, jolloin **Build**-valikon avaaminen ei aina ole tarpeen. Pikanäppäimet ovat F5 komennolle **Go**, F11 komennolle **Step Into** ja CTRL+F10 komennolle **Run to Cursor**.

Debuggerin ikkunat

Kun debugattava ohjelma pysähtyy keskeytyskohtaan, debuggeri päivittää ikkunoihinsa sovelluksen nykyisen tilan mukaiset tiedot. Tärkein ikkunoista on source-ikkuna, joka näyttää lähdekoodin kohdasta, jossa suoritus pysähtyi. Pieni keltainen nuoli, *komento-osoitin* (instruction pointer), ilmestyy marginaaliin osoittamaan keskeytettyä komentoa. Komento-osoitin osoittaa komennon, jota ei

ole vielä suoritettu, mutta joka suoritetaan ensimmäisena ohjelman suorituksen jatkuessa.

Kun suoritus palautuu debuggerille, **Debug**-työkalurivi ilmestyy näytölle. Kuvassa 13.6 osoitetut kuusi painiketta toimivat kaksiasentoisesti niin, että ne vuoroin tuovat näyttöön ja vuoroin piilottavat telakoitavat ikkunat, jotka näyttävät tietoja ohjelman nykyisestä tilasta. Taulukossa 13.4 on kerrottu kunkin ikkunan antaman tiedon tyyppi.



Kuva 13.6 Debugger-ikkunan painikkeet

Taulukko 13.4 Debuggerin Windows-painikkeilla saatavat tiedot

Ikkuna	Näytettävä tieto
Watch	Debuggerin seuraamien muuttujien ja lauseiden nykyiset arvot
Variables	Keskeytyskohdassa tai sen läheisyydessä käsiteltyjen muuttujien nykyiset arvot
Registers	CPU:n rekisterien nykyiset arvot
Memory	Määrätyn muistiosoitteen sisältö
Call Stack	Luettelo kutsutuista funktioista, joiden suoritus ei ole vielä päättynyt
Disassembly	Assembly-kielinen käännös käännetystä koodista, joka täydentää näytöllä olevaa source-ikkunaa

Variables-ikkuna ja Watch-ikkuna

Variables-ikkunassa esitetään keskeytyskohdassa relevanttien muuttujien tietoja. Muuttujat, joita viimeksi suoritettu komento on käsitellyt ja yleensä myös parin edellisen komennon käsittelemät muuttujat, näkyvät Variables-ikkunassa. Voit muuttaa muuttujan arvoa kaksoisnapauttamalla sitä Variables-ikkunassa ja kirjoittamalla uuden arvon.

Watch-ikkunassa näytetään määrättyjen muuttujien arvot riippumatta siitä, onko niihin viitattu ohjelmassa vai ei. Uusien muuttujien lisääminen Watch-ikkunaan tehdään kaksoisnapauttamalla ikkunan new-entry-ruutua ja kirjoittamalla uuden muuttujan nimi. Kuvassa 13.6 näkyvä QuickWatch antaa mahdolllisuuden muuttujan arvon tutkimiseen ilman sen lisäämistä Watch-ikkunaan. Jos muuttujan nimi näkyy ruudulla, debuggerissa on käytössä helpompikin tapa sen nykyisen arvon selvittämiseksi. Siirrät vain hiiren osoittimen muuttujan päälle ja näkyviin ilmestyy työkaluvihje, joka kertoo muuttujan arvon.

Memory-ikkuna ja Registers-ikkuna

Memory-ikkuna näyttää annetun muistiosoiteen sisällön. Memory-ikkuna on hyödyllinen, jos halutaan tutkia puskureita, jotka eivät välttämättä näy Variables-ikkunassa. Määritä puskurin osoite etsimällä se Variables- tai Watch-ikkunasta ja kirjoita tai liitä osoite Memory-ikkunaan ja paina ENTER.

Registers-ikkunasta nähdään prosessorin rekisterien tila sillä hetkellä, kun ohjelman suoritus pysäytettiin. Registers-ikkunaa käytetään yleensä vain Disassembly-ikkunan ollessa näytössä, jolloin nähdään samalla koodi assembly-kielisessä muodossa.

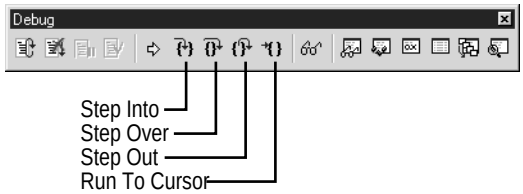
Call Stack -ikkuna

Call Stack -ikkuna näyttää reitin, jota kautta sovellus on päätynyt tutkittavaan kohtaan. Se vastaa kysymykseen, “Kuinka minä tänne jouduin?”

Kutsupino (call stack) on luettelo toisiinsa liittyvistä funktioista, joita on kutsuttu, mutta joita ei vielä ole suoritettu loppuun. Lista alkaa nykyisestä funktiosta, jossa keskeytyskohta oli ja jatkuu käänteisessä järjestyksessä kohti vanhinta kantafunktiota. MFC-ohjelmissa suoritus etenee usein monien frameworkiin kätkeytyneiden funktioiden kautta, joten niiden kutsupinot saattavat olla pitkiä.

Suoritus komennoittain

Debug-työkalurivillä on joukko painikkeita (kuvassa 13.7), joiden avulla voit käydä pysäytettyä ohjelmaa läpi vaiheittain. Tunnistat nämä työkalut nuolista ja niissä olevista mutkikkaista haaroista. Järjestyksessä lueteltuina painikkeet käynnistävät **Step Into**, **Step Over**, **Step Out** ja **Run To Cursor** -komennot. Olemme jo käsitelleet **Run To Cursor** -komentoa; tässä osassa käydään läpi **Step Into**, **Step Over** ja **Step Out** komennot.



Kuva 13.7 **Debug**-työkalurivin asekelluspainikkeet

Step Into ja **Step Over** -komennot (tai niitä vastaavat pikanäppäimet F11 ja F10) antavat mahdollisuuden suorittaa ohjelmaa komennoittain. Kun valitset joko **Step Into** tai **Step Over**, debuggeri antaa ohjelman suorituksen jatkua,

mutta vain yhden komennon verran. Kun komento on suoritettu, debuggeri pysäyttää suorituksen uudelleen. Jos Disassembly-näkymä on käytössä, **Step** -työkalut suorittavat yhden assembly komennon korkean tason C/C++-komennon sijasta. Jos suoritettava komento kutsuu funktiota, **Step Over** pysähtyy kutsua seuraavaan komentoon ja **Step Into** puolestaan ensimmäiseen komentoon kutsutussa funktiossa. Jos komento ei kutsu funktiota, **Step Into** ja **Step Over** toimivat molemmat samalla tavoin.

Step Out -komentoa voidaan käyttää funktiosta poistumiseen. Komennon seurauksena funktio suoritetaan loppuun ja suoritus pysähtyy funktion kutsua seuraavaan komentoon. Toisin sanoen funktion kutsun yhteydessä **Step Into** ja **Step Out** -komennot yhdessä toimivat kuten **Step Over**.

Edit and Continue

Visual C++:n **Edit and Continue** -ominaisuuden avulla voit korjata pysyvästi monia ongelmia suoraan debuggerin source-ikkunassa ilman, että debuggeri pitäisi sulkea ja ohjelma kääntää uudelleen. Kun jatkat muutetun ohjelman suorittamista, Visual C++ kääntää ensin muuttuneet osat ja korvaa tarvittavat osat oikeilla versioilla.

Edit and Continue -toiminnossa on tiettyjä rajoituksia; se ei tunnista muutoksia, jotka ovat mahdottomia, epäkäytännöllisiä tai eivät ole turvallisia debuggauksen aikana kuten:

- Muutokset poikkeuskäsittelijöihin.
- Funktioiden poistaminen kokonaan.
- Muutokset luokkien ja funktioiden määrittelyihin.
- Muutokset staattisiin funktioihin.
- Muutokset resurssien tietoihin projektin resurssi (.rc) -tiedostoissa.

Jos yrität käynnistää ohjelman uudelleen **Edit and Continue** -toiminnon kautta tällaisen muutoksen jälkeen, debuggeri näyttää tilarivillä virheilmoituksen, jossa kuvataan ongelma. Voit jatkaa debuggausta käyttäen alkuperäistä koodia tai sulkea debuggerin ja kääntää muutetun koodin normaalisti.

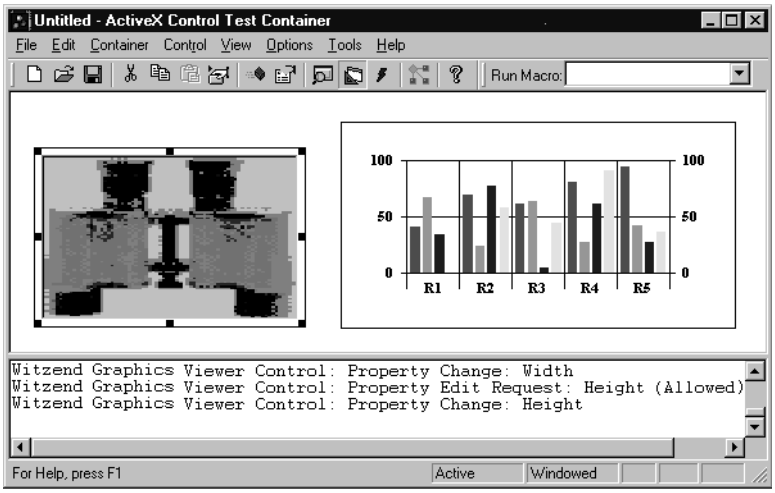
COM-komponenttien debuggaus

Visual C++ -debuggeri pystyy käsittelemään ongelmitta in-process COM-komponentteja kuten ActiveX-kontrolleja, vaikka nämä komponentit vaativatkin suoritusta varten säilösovelluksen. Vaikka säilö olisi tehty erillisenä projektina, debuggaus voidaan aloittaa myös komponenttiprojektista. Debuggeri osaa ylittää projektien välisen rajan näkymättömästi ohjelman suorituksen siirtyessä asiakkaalta palvelimelle ja takaisin.

Aloita kontrolliprojektin debuggaus määrittelemällä säilösovellus, johon haluat komponentin upottaa. Kirjoita **Project Settings** -dialogin **Debug**-välilehdellä säilön polku ja tiedostonimi **Executable For Debug Session** -nimiseen muokkausruutuun. Voit myös etsiä säilön napauttamalla ruudun vieressä olevaa nuolta. Tämä toiminto näyttää pienen valikon, jonka yhtenä vaihtoehtona on **ActiveX Control Test Container**.

ActiveX Control Test Container

Kuvassa 13.8 nähtävä Test Container apuohjelma on yleiskäyttöinen säilöohjelma ActiveX-kontrolleille. Visual C++ sisältää Test Containerin, joten voit debugata ja testata ActiveX-kontrolleja myös ilman niitä varten erikseen tehtyjä säilösovelluksia.



Kuva 13.8 ActiveX Control Test Container

Ensin täytyy asettaa keskeytyskohdat projektin lähdekoodiin. Jos Test Container on valittu säilöksi, debuggeri käynnistyy automaattisesti, kuten kaikkien muidenkin säilöjen kanssa. Kun keskeytyskohta laukeaa, suoritus palaa Visual C++ -debuggeriin, joka on valmiina sovelluksen vaiheittaiseen läpikäymiseen.

Voit käyttää Test Containeria myös ilman debuggeria, kuten seuraavassa harjoituksessa nähdään.

ActiveX-kontrollien ajaminen Test Containerissa

Tässä harjoituksessa käynnistetään Test Container apuohjelma manuaalisesti, käynnistämättä debuggeria. Opit, kuinka avaat ActiveX-kontrollin Test Containeriin ja ohjaat sitä sen metodien ja ominaisuuksien kautta.

► **Test Containerin käynnistäminen**

1. Napauta **Tools**-valikosta **ActiveX Control Test Container**.
2. Kun Test Container avautuu, napauta työkaluriviltä **New Control**, joka näyttää luettelon rekisteröidyistä ActiveX-kontrolleista.
3. Jos olet kääntänyt ActiveX-kontrollin ja se on oikein rekisteröity, etsi se listalta. Muussa tapauksessa aktivoi kaksoisnapauttamalla mikä tahansa listalla oleva kontrolli, esimerkiksi Internet Explorerin mukana tulevaa **Calendar**-kontrollia.
4. Kun kontrolli avautuu Test Container ikkunaan, avaa **Invoke Methods** -dialogi napauttamalla **Invoke Methods** -työkalua.
5. Avaa luettelo kontrollin julkistamista metodeista napauttamalla **Method Name** -ruutua. Jos olet aktivoinut **Calendar**-kontrollin, valitse luettelosta **BackColor (PropPut)**. **PropPut**-merkinnällä varustetut metodit asettavat kontrollin ominaisuuden ja **PropGet**-metodit lukevat ominaisuuden arvon.
6. Kirjoita ominaisuuden uusi arvo **Parameter Value** -ruudusta. Aseta väri kirjoittamalla RGB-arvoa vastaava kymmenjärjestelmän luku, kuten **255**, joka vastaa kirkaan punaista väriä.
7. Kutsu metodia uudella arvolla napauttamalla **Invoke**-painiketta. Jos ominaisuus vaikuttaa kontrollin ulkoasuun (kuten **BackColor**-ominaisuus), vastaavan muutoksen pitäisi tulla näkyviin.
8. Poista valittu kontrolli painamalla DELETE ja sulje Test Container.

Prosessin ulkopuolisen COM-palvelimen debuggaus

Monet prosessin ulkopuolisista COM-palvelimista voivat toimia itsenäisinä ohjelmina, eivätkä näin vaadi teknisesti säilöä suoritustaan varten. Koska palvelimen metodit ja tapahtumat käynnistyvät kuitenkin vain asiakkaan niitä kutsuessa, täytyy **Project Settings** -dialogin **Debug-**välilehdellä määritellä polku palvelimen sijasta asiakkaaseen. Aseta haluamasi keskeytyskohdat serverin koodiin ja käynnistä palvelin, joka puolestaan käynnistää automaattisesti asiakasohjelman. Kun sekä palvelin että asiakas ovat aktiivisia, siirry asiakkaaseen ja suorita toiminnot, jotka tarvitaan palvelimen tutkittavan funktion käynnistämiseen. Prosessin sisäisenä palvelimena Visual C++ debuggeri aktivoituu, kun suoritus laukaisee keskeytyskohdan. Voit tämän jälkeen tutkia palvelimen koodia.

Oppitunnin yhteenveto

Tällä oppitunnilla esiteltiin Visual C++ -debuggeri, joka on arvokas työkalu haettaessa ohjelmassa olevien virheiden syitä. Opit, kuinka asetetaan paikkaan ja tietoon perustuvia keskeytyskohtia **Breakpoints**-dialogin avulla, kuinka debuggeri käynnistetään ja kuinka ohjelmaa voidaa suorittaa komennoittain, sen suorituksen pysähdyttä keskeytyskohtaan. Oppitunnilla esiteltiin myös debuggerin **Edit and Continue** -ominaisuus, jonka avulla voit korjata lähdekoodia poistumatta debuggerista uudelleenkääntämistä varten.

Tyypillisesti debuggaus aloitetaan tekstieditorissa. Ennen debuggerin käynnistämistä asetetaan keskeytyskohdat paikkoihin, joihin ohjelman suorittaminen halutaan pysäyttää. Debuggeri puolestaan käynnistää ohjelmasi automaattisesti ja pysäyttää sen tavatessaan keskeytyskohdan. Debuggerissa on kuusi telakoituvaa ikkunaa:

- **Watch** näyttää valitun tiedon arvon.
- **Variables** näyttää nykyiseen komentoon liittyviä tietoja.
- **Call Stack** sisältää luettelon yhteenliittyvistä funktiokutsuista.
- **Registers** näyttää CPU:n rekistereiden nykyiset tilat.
- **Memory** näyttää minkä tahansa käytettävissä olevan muistialueen sisällön.
- **Disassembly** kääntää lähdekoodin vastaaviksi assembly-komennoiksi.

In-process ActiveX-kontrollit ovat DLL:iä ja siksi ne voidaan debugata samoin kuin normaalit DLL:iä. Visual C++ sisältää myös Test Container -apuohjelman, joka toimii yleisenä säilöohjelmana ajettaessa ActiveX-kontrolleja. Test Container antaa mahdollisuuden testata ActiveX-kontrolleja ilman erillistä sille tehtyä säilöohjelmaa.

Oppitunti 5: Dependency Walker

Visual C++ sisältää debuggerin lisäksi muitakin apuohjelmia, joden avulla sovelluksia ja DLL-moduleita voidaan tutkia. Tällä oppitunnilla opetellaan käyttämään Dependency Walker -apuohjelmaa, joka tunnetaan myös tiedostonimellä Depends.exe.

Tämän oppitunnin jälkeen:

- Tiedät miksi Windows-ohjelmien suorittaminen riippuu muista moduleista.
- Osaat tutkia ohjelman riippuvuuksia Dependency Walker -apuohjelmalla.

Oppitunnin arvioitu kesto: 15 minuuttia

Mitä riippuvuus tarkoittaa?

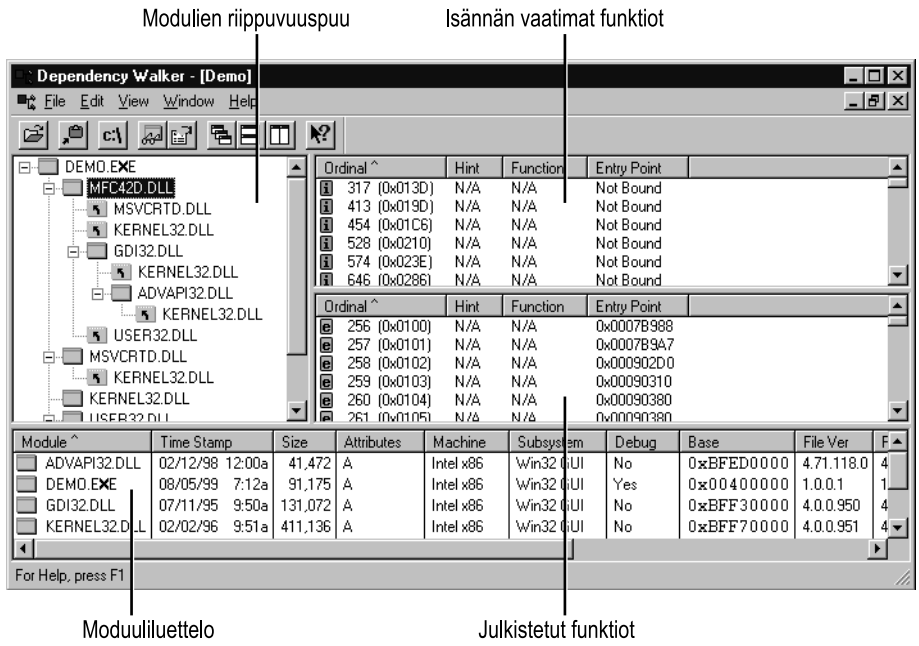
Windows-ohjelmat eivät ole niin itseriittoisia kuin ensi katsomalta näyttää. Jopa yksinkertainen "Hello, world"-ohjelma vaatii toimiakseen useiden järjestelmän tarjoamien DLL:ien kuten Kernel32 ja GDI32 olemassa oloa. Koska perussovellus MyApp (jota on käytetty tämän kirjan esimerkeissä) on linkitetty MFC:hen, ohjelma on riippuvainen muista tiedostomoduleista, kuten MFC42.dll. MFC puolestaan ei voi toimia ilman muita tiedostoja, kuten C run-time-kirjastoa MSVCRT.dll. Näiden vaadittujen modulien jonon täytyy olla kokonaisuudessaan järjestelmän saatavilla, että ohjelma voidaan suorittaa. Yhdessä ne muodostavat ohjelman *riippuvuudet* (dependencies).

Jokainen suoritettava Windows-moduli, olipa kyseessä sovellus tai DLL, sisältää luettelon riippuvuuksistaan. Luettelo on tiedoston otsikko-osassa. Kun käyttöjärjestelmä lataa moduulin, se lukee riippuvuuksien luettelon ja lataa kaikki tarvittavat moduulit. Alkuperäinen ohjelma käynnistyy vain, jos kaikki riippuvuudet saadaan ladattua.

Riippuvuusinformaatio

Alkujaan sovelluksen riippuvuudet ovat käyttäjälle näkymättömissä. Riipuvuus tulee tietoon yleensä vain, jos jokin tarvittavista tiedostoista puuttuu. Tällaisessa tapauksessa Windows näyttää virheilmoituksen, jossa kerrotaan, ettei ohjelmaa voida käynnistää, koska tarvittavaa tiedostoa ei löydy.

Dependency Walker -apuohjelma lukee riippuvuusluettelon ohjelman alusta ja näyttää jokaisen riippuvuuden tiedot. Lopputulos on kuvattu seuraavalla sivulla kuvassa 13.9, jossa nähdään tyypillisen Demo.exe-ohjelman riippuvuudet.



Kuva 13.9 Dependency Walker -apuohjelma

Dependency Walker näyttää seuraavat tiedot:

- Kaikkien ohjelman tai DLL:n suorittamiseen vaadittavien tiedostojen nimet ja sijainnit.
- Jokaisen riippuvuusmoduulin perusosoite.
- Jokaisen riippuvuusmoduulin versiotiedot.
- Sisältääkö riippuvuusmoduuli debug-tiedot.

► **Dependency Walkerin käynnistäminen**

1. Valitse **Käynnistä**-valikosta **Ohjelmat**.
2. Napauta **Microsoft Visual Studio 6.0**, napauta **Microsoft Visual Studio 6.0 Tools**, ja napauta sitten **Depends**. Voit myös lisätä Dependency Walkerin käynnistävän komennon Visual C++:n **Tools**-valikkoon.
3. Kun apuohjelma avautuu, napauta **Open** valikosta **File** ja etsi sovellus tai DLL, jota haluat tutkia.

Dependency Walker on MDI-sovellus, joten voit avata useita näkymiä, joista jokainen näyttää eri tiedoston riippuvuudet. Kuten kuva 13.9 paljastaa, Dependency Walker näyttää neljä toisistaan erotettua ruutua. Taulukossa 13.5 kuvataan kuvassa 13.9 nähtävät ruudut.

Taulukko 13.5 Dependency Walkerin näytön neljä ruutua

Ruutu	Kuvaus
Module dependency tree	Hierarkkinen puu, jossa nähdään ohjelman riippuvuudet. Nimet esiintyvät puussa usein monesti, koska useammat modulit voivat olla riipuvaisia samasta tiedostosta.
Parent import function list	Module dependency treestä valitun modulin funktiot, joita sen kanta modulit kutsuvat. Valitun modulin tulee julkistaa kaikki sen kantamodulien kutsumat funktiot. Jos valittu moduli ei julkista kaikkia kutsuttuja funktiota, modulin latauksen aikan esiintyy ratkaisemattomia ulkoisia virheitä.
Export function list	Module dependency treestä valitun modulin julkistamat funktiot. Julkistetut funktiot ovat muiden modulien käytettävissä.
Module list	Luettelo kaikista ohjelman riippuvuuksista. Tämä luettelo määrittelee ohjelman suorittamiseen vähintään tarvittavat tiedostot.

Oppitunnin yhteenveto

Tällä oppitunnilla esiteltiin Dependency Walker -apuohjelma, joka kertoo tietoja moduleista, joita Windows-ohjelma vaatii toimiakseen. Dependency Walker näyttää:

- Ohjelman riipuvuudet puunäkymässä.
- Moduulin kantamoduulin kutsumien funktioiden luettelon.
- Moduulin julkistamien funktioiden luettelon.
- Luettelon ohjelman riippuvuuksista.

Oppitunti 6: Spy++

Tällä oppitunnilla tutustutaan Spy++:n, toisen Visual C++ -apuohjelman käyttöön. Tällä käytännöllisellä ja suositulla työkalulla voidaan tarkkailla suoritettavien prosessien, niiden säikeiden ja kaikkien avoimien — jopa kätkettyjen — tietoja.

Tämän oppitunnin jälkeen:

- Osaat käynnistää Spy++-apuohjelman ja saat sen avulla järjestelmän objektien kuten prosessien, säikeiden ja ikkunoiden väliset yhteydet näkyviin graafisessa puumuodossa.
- Osaat Spy++:aa käyttämällä hakea tietyn ikkunan, säikeen tai sanoman.
- Osaat tutkia valitun ikkunan, prosessin tai säikeen ominaisuuksia.
- Osaat Spy++ Finderia käyttämällä valita ikkunan tarkkailtavaksi.
- Osaat tarkkailla reaaliaikaista sanomakirjanpitoa ikkunan saamista ja lähettämistä sanomista.

Oppitunnin arvioitu kesto: 15 minuuttia

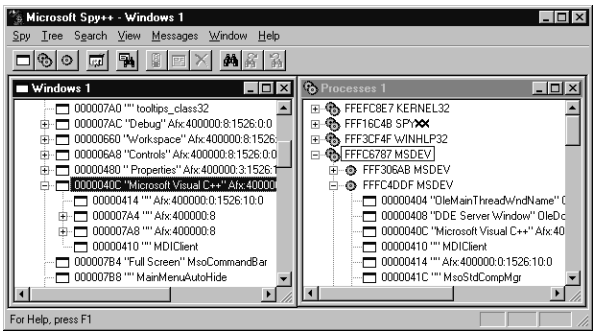
Spy++:n näkymät

Käynnistä Spy++ samoin kuin Dependency Walkerkin valitsemalla se **Microsoft Visual Studio 6.0 Tools** -valikosta. Apuohjelma näyttää neljä päänäkymää:

- **Windows view** esittää listan avoimista ikkunoista.
- **Processes view** luettelee nykyiset prosessit.
- **Threads view** luettelee nykyiset säikeet.
- **Message log** luettelee sanomat, jotka ikkuna vastaanottaa.

Nämä neljä näkymää voidaan aktivoida **Spy**-valikon komennoilla tai työkalurivin neljällä ensimmäisellä painikkeella. Jokaisella kerralla, kun komento annetaan, luodaan näkymä uuteen lapsi-ikkunaan, joten työkalupainikkeilla ei voi siirtyä avoimesta näkymästä toiseen. Näitä painikkeita tulisikin käyttää vain kerran luotaessa näkymiä ja sen jälkeen näkymien välillä liikutaan **Window**-valikon toimintojen avulla. Kuten Dependency Walkerkin, Spy++ on MDI-ohjelma, joten voit pinota näkymät työalueelle.

Kuvassa 13.10 näet, miltä Windows ja Processes -näkymät voisivat näyttää järjestettyinä Spy++-ikkunaan. Näkymästä selviää, että Visual C++ on käynnissä (nimellä **MSDEV** Processes-ikkunassa) ja että se on luonut useita lapsi-ikkunoita.



Kuva 13.10 Windows ja Processes -näkymät Spy++-ikkunassa

Spy++-näkymä on tilannevedos, joten sovellukset, joiden suoritus alkaa vasta Spy++:n käynnistyksen jälkeen, eivät näy automaattisesti tässä luettelossa. Vastaavasti sovellukset, jotka suljetaan Spy++:n suorituksen aikana, eivät poistu listalta itsekseen. Näkymän päivittäminen tehdään **Refresh**-komennolla, joka on **Window**-valikossa.

Windows-näkymä

Windows-näkymä esittää listan kaikista avoinna olevista ikkunoista. Sillä onko ikkuna näkyvissä tai piilotettuna ei ole väliä; tyypillisessä Windows-istunnossa monet ikkunat ovat piilotettuina ja toimivat vain sovellustensa sanomien vastaanottajina. Kun Spy++ avataan tai sen näyttö päivitetään, ohjelma muodostaa listan kaikista ikkunoista ja näyttää sen Windows-näkymässä.

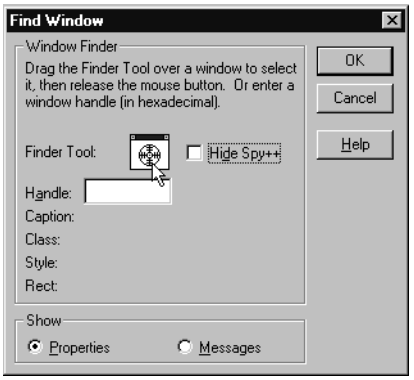
Luettelo esitetään normaalissa puunäkymässä, jossa pienillä plus- ja miinusmerkeillä voidaan tasoja avata ja piilottaa. Puun hierarkia perustuu ikkunoiden vanhemmuuteen — siis yhteyteen, jossa yksi ikkuna voi luoda itselleen lapsi-ikkunoita. Ota esiin minkä tahansa ikkunan lapsi-ikkunat napauttamalla sen nimen vieressä olevaa plusmerkkiä. Jos ikkunan nimen vieressä ei ole plusmerkkiä, ikkunalla ei ole lapsi-ikkunoita.

Tavallisesti Windows-näkymän luettelo sisältää useita ikkunoita, jolloin tietyn ikkunan löytäminen voi olla vaikeaa. Tällaisessa tapauksessa käytetään Window Finder -työkalua.

► **Window Finder -työkalun käyttö**

1. Järjestä näyttö niin, että Spy++ ja ikkuna, jota haluat tutkia, ovat molemmat näkyvissä ruudulla.
2. Napauta Spy++-työkaluriviltä **Find Window** tai valitse **Find Window** -komento **Search**-valikosta.

3. Raahaa **Finder Tool** -kuvake (kuvassa 13.11) ulos **Find Window** -dialogista. Kun raahattu kuvake sattuu näkyvissä olevan ikkunan päälle, ikkuna kehystetään ohuella mustalla viivalla ja **Find Window** -dialogi näyttää ikkunan kahvan. Pudota kuvake haluamasi ikkunan päälle ja sulje **Find Window** -dialogi. Voit tämän jälkeen etsiä ikkunaan liittyvän merkinnän hakemalla sen kahvaa.



Kuva 13.11 Finder Tool -kuvake

Processes-näkymä ja Threads-näkymä

Jokainen Windows-moniajoympäristössä toimiva prosessi luo yhden tai useamman säikeen; jokainen säie voi luoda rajoittamattoman määrän ikkunoita. Käytä Processes-näkymää tutkiaksesi tiettyä prosessia, joka tavallisesti vastaa suoritettavaa ohjelmaa. Prosessit tunnistetaan moduulin nimellä tai ne määrätään *järjestelmäprosesseiksi* (system processes).

Threads-näkymässä luetellaan kaikki käynnissä olevat säikeet varustettuina ne omistavien prosessien nimillä. Avaa säie nähdäksesi siihen liittyvät ikkunat.

Sanomaloki

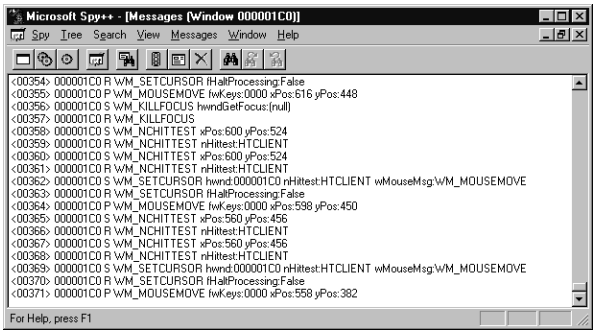
Debuggaus käyttöön ehkä kaikkein hyödyllisin Spy++:n toiminto on sanomaloki (message log). Se pitää reaaliaikaista kirjanpitoa ikkunan lähettämistä ja vastaan-ottamista sanomista. Tämä mahdollistaa sanomaluettelon ja sanomien saapumisjärjestyksen tutkimisen, mikä on erityisen käytännöllistä ikkunan alustuskomentojen nappaamisessa.

Aiemmin kuvatun Window Finder -työkalun avulla voidaan kätevästi aloittaa ikkunan sanomien kirjaaminen.

► Sanomien kirjaaminen Window Finderilla

1. Raahaa ja pudota **Finder Tool** -kuvake kohdeikkunan päälle kuten teit aikaisemmin. Napauta tällä kertaa **Messages**-valintapainiketta **Find Window** -dialogin alareunassa ennen dialogin sulkemista.

   Kun käytät kohdeikkunaa Spy++ muodostaa luettelon ikkunan lähettämistä ja vastaanottamista sanomista kuten kuvassa 13.12.



Kuva 13.12 Spy++ sanomaloki

Spy++:n sanomaloki on jaettu neljään sarakkeeseen, jotka on kuvattu taulukossa 13.6.

Taulukko 13.6 Spy++:n sanomalokin neljä saraketta

Sarake	Kuvaus
1	Indeksinumero, joka seuraa sanomien määrää
2	Ikkunan kahva
3	Sanomakoodi: joko S (lähetetty), R (vastaanotettu), P (postitettu), tai s (sanoma lähetettiin, mutta suojaus estää paluuarvon saamisen)
4	Sanoma, parametrit ja paluuarvot

   Oletuksena listaan poimitaan kaikki sanomat, mukaan lukien hiiren liikkeet ja näppäimen painallukset kuten WM_KEYDOWN ja WM_KEYUP. Tämän takia luettelo voi kasvaa erittäin nopeasti. Spy++ antaa onneksi mahdollisuuden suodattaa sanomista vain ne jotka kiinnostavat eniten.

2. Valitse **Messages**-valikosta **Options** ja napauta sitten **Messages**-välilehteä.
3. Tyhjennä kaikki valintaruudut napautamalla **Clear All** -painiketta ja valitse sitten listasta vain ne, joita haluat tarkastella.
4. Päätä sanomien seuraaminen napauttamalla pysäytyspainiketta **Spy++**:n työkaluriviltä tai napauttamalla **Stop Logging** komentoa **Messages**-valikosta.

Window-sanomien tutkiminen Spy++:lla

Tässä harjoituksessa tarkkailet toista sovellusta käynnistämällä Spy++-apuohjelman. Käynnistämällä apuohjelman sanomaloki-toiminnon, voit seurata valitulle ikkunalle kuuluvia sanomia reaaliaikaisesti.

► **Spy++:n käynnistäminen**

1. Valitse **Programs**-valikosta **Start**.
2. Napauta **Microsoft Visual Studio 6.0**, napauta **Microsoft Visual Studio 6.0 Tools**, ja napauta sitten **Spy++**.
3. Kun apuohjelma avautuu, napauta työkalurivin **Find Window** -työkalua ja raahaa ja pudota **Finder Tool** -kuvake toisen ikkunan päälle. Jos mikään muu sovellus ei ole avoinna, pudota **Finder Tool** -kuvake työpöydälle.
4. Napauta **Messages**-valintapainiketta, joka on **Find Window** -dialogin alaosassa ja sulje sitten dialogi.
5. Siirtele kokeeksi hiiren osoitinta valitun ikkunan päällä. Näet useita sanomia, joita ikkuna lähettää ja vastaanottaa hiiren liikkeisiin liittyen, esimerkiksi WM_MOUSEMOVE ja WM_SETCURSOR.

Oppitunnin yhteenveto

Tällä oppitunnilla kerrottiin Spy++:sta, Visual C++:n apuohjelmasta, jolla voidaan seurata toiminnassa olevien ohjelmien tietoja. Spy++ tarjoaa arvokkaan näkymän sovelluksen tietohin, joihin Visual C++:n debuggerin avulla ei päästä käsiksi.

Spy++:n näytöllä on neljä päänäkymää, jotka näyttävät luettelon ikkunoista, prosesseista, säikeistä ja sanomista, jotka liittyvät valittuun ikkunaan. Window Finder -työkalun avulla on helppo yhdistää mikä tahansa näkyvillä oleva ikkuna Spy++:n näytöllä olevaan ikkunaluetteloon. Sanomien seurantaominaisuus kaappaa ja kirjaa kaikki valittuun ikkunaan saapuvat ja siitä lähtevät viestit, näyttäen aikajärjestyksessä sanomien arvot ja parametrit. Sanomalokia voidaan suodattaa niin, että siinä näkyvät vain ne sanomat, jotka erityisesti ovat kiinnostavia.

Oppitunti 7: Sovelluksen testaaminen

Sovellusta testaamalla varmistetaan, että se todella toimii määritellyllä tavalla. Kun sovellus alkaa olla lähes valmis ja riittävän vakaa, vakavan testauksen ja debuggauksen tulisi alkaa — testaamalla etsitään virheitä ja debuggauksella korjataan niitä. Tällä oppitunnilla tutustutaan muutamiin testaustekniikoihin.

Testaus tulisi mahdollisimman suuressa määrin tehdä vaihtelevissa tosielämän tilanteita vaastaavissa olosuhteissa, joissa ohjelmaa tullaan käyttämään. Esimerkiksi sovellusta, joka on suunniteltu käytettäväksi sekä Microsoft Windows 95- että Microsoft Windows NT -ympäristöissä, tulisi testata näissä molemmissa. Samaan aikaan, kun ohjelmoijat pyrkivät tuottamaan toimivaa koodia, testaajat pyrkivät saamaan koodin toimimattomaksi — eli löytämään sen heikkoudet. Ohjelmoijien, joiden täytyy tehdä molempia, tulisi tehdä kummatkin työt yhtä suurella innostuksella.

Tämän oppitunnin jälkeen:

- Ymmärrät testaukseen liittyvää sanastoa.
- Osaat tehdä tehokkaan testaussuunnitelman.
- Osaat käyttää yleisimpiä testaustekniikoita sovelluksesi testaamiseen.

Oppitunnin arvioitu kesto: 15 minuuttia

Testaussanasto

Kuten monilla muillakin erikoisaloilla testauksellakin on oma sanastonsa. Tässä osassa selitetään testaukseen liittyviä termejä, jotta seuraavat osat olisivat selkeämpiä ymmärtää. Sanasto on jaoteltu tasoittain, alkaen yksinkertaisimmista testaustehtävistä siirtyen kohti monimutkaisempia toimintoja.

- **Moduulitestaus** varmistaa yhtenäisen koodin palan, kuten silmukan, lohkon, alirutiinin tai tapahtuman toimivuuden. Muodollisessa kielenkäytössä yksikkötestauksella viitataan testaukseen, joka kohdistuu pienimpiin koodin osiin, joita voidaan käytännössä testata.
- **Integrointitestaus** on seuraavaksi korkeampi testauksen taso ja sen avulla varmistetaan, että koodin osien liittäminen monimutkaisemmiksi prosesseiksi ei aiheuta ongelmia. Kaksi funktiota voi esimerkiksi toimia erillisinä yksikköinä täysin oikein, mutta toisen funktion toiminnan vaikutus toisen funktion tuloksiin täytyy myös tutkia. Säikeitä täytyy tarkkailla erityisesti, että voidaan olla varmoja siitä, että ne toimivat oikein, kun niitä suoritetaan samanaikaisesti.

- **Järjestelmätestaus** (System testing) suoritetaan sovellukselle kokonaisuudessaan. Tällä tasolla keskitytään vähemmän virheiden etsintään ja ennemmän siihen, että sovellus ja sen ympäristö toimivat yhdessä oikealla tavalla. Testauksen tulisi tällä tasolla olla järjestelmän laajuista — testattavia asiota ovat esimerkiksi rekisterin oikeat asetukset, toimintakyky, resurssien odottamatton loppuminen, kirjautumisvirheet ja virheistä toipuminen.
- **Rasitustestaus** (Stress testing) tutkii sovelluksen toimintaa tilanteissa, joissa toimintaympäristö aiheuttaa esteitä sovelluksen ajamiselle. Tällaisia tilanteita voi syntyä esimerkiksi vapaan muistin käydessä vähiin, verkkoliikenteen ollessa tavallista suurempaa tai vapaan levytilan loppuessa.
- **Regressiotestaus** (Regression testing) tarkoittaa aikaisempien testivaiheiden toistamista lähdekoodiin tehtyjen muutosten jälkeen. Regressiotestauksen tarkoituksena on varmistaa, että tehdyt bugikorjaukset ovat olleet onnistuneita ja järjestelmä toimii edelleen toivotulla tavalla. Toisin sanoen varmistetaan, että korjaukset eivät ole aiheuttaneet uusia ongelmia.
- **Beta-testaus** on vaihe, jossa ohjelmoija toimittaa sovelluksen esiversion valitulle käyttäjäjoukolle. Käyttäjät valitaan yleensä tuotteen aikaisempien versioiden käyttäjistä, jotka ovat halukkaita käyttämään uutta versiota erilaisissa olosuhteissa ja joiden tiedetään osaavan kertoa mielipiteensä uuden tuotteen hyvistä ja huonoista puolista.
- **Kenttätestaus** (User acceptance testing) suoritetaan jakamalla testattava jakelu joukolle ennalta määrättyjä käyttäjiä, joita on jo opastettu järjestelmän käytössä. Testin suorittavien käyttäjien tulisi käyttää järjestelmää aivan samoin kuin sitä käytettäisiin todellisissa tilanteissa.

Testaussuunnitelman tekeminen

Testaussuunnitelma (test plan) sisältää sovelluksen kaikki testit kirjallisessa muodossa. Kunnolla tehty testaussuunnitelma kuvaa täydellisesti kaikki vaadittavat testit ja kertoo, mitkä ovat kunkin testausvaiheen vaatimukset. Testisuunnitelman tulee olla kirjallinen, jotta myös muut kuin tekijä itse tietäisivät, kuinka testit viedään läpi. Hyvän testaussuuniltelman tunnistaa siitä, että sen avulla voidaan tarvittaessa uusi testaaja vaihtaa tekemään testausta kesken suunnitelman toteuttamista.

Räätälöityjen sovellusten testaus voidaan suorittaa tehokkaasti käyttämällä kolmea menetelmää:

- Ohjelmoijien ja testaus-/laadunvarmistusryhmän suorittama sisäinen testaus
- Valittujen käyttäjien suorittama beta-testaus
- Valittujen käyttäjien tekemä kenttätestaus

Testaussuunnitelman osat

Testaussuunnitelma tarjoaa muodollisen lähtökohdan, jonka perusteella voidaan kehittää takautuvasti toistettavia testejä. Kun sovellus muuttuu, tai kun tehdään uusia käännöksiä, on tärkeää, että olemassaoleva vakaus ei kokonaisuudessaan vaarannu. Testaussuunnitelman perusteella testausstrategia voidaan myös tarkastaa ja käydä eri osapuolien kanssa läpi.

Hyvä testaussuunnitelma alkaa sovelluksen ja sen testattavan toiminnallisuuden kuvauksella, jonka jälkeen käsitellään lyhyesti testauksen päämäärät. Suunnitelman tulisi sisältää seuraavat osat:

- Testauksen päämäärät.
- Kuvaus siitä, kuinka testit tulisi toteuttaa. Testin luotettavuuden kannalta tärkeät tekijät kuten testiskriptit, tarkistuslistat ja käyttäjien osallistuminen, tulee kuvata.
- Kuvaus testin toteutusympäristöstä, mukaan lukien käyttöjärjestelmä ja tarvittaessa sen versionumero. Esimerkiksi Windows 95:n ensimmäinen julkaistu versio on hieman erilainen kuin Windows 95, johon on asennettu Service Pack 1. Nämä erot saattavat olla testaussuunnitelman kannalta oleellisia.
- Luettelo testitiedoista, jotka tarvitaan pätevään testaukseen.
- Huomautukset testiryhmän rajoituksista, jotka voivat vaikuttaa testin luotettavuuteen. Jos esimerkiksi testataan sadoille käyttäjille tarkoitettua keskitettyä suurta tietokantaa, pienen organisaation voi olla mahdoton simuloida riittävää käyttäjämäärää.
- Kuvaus eri kriteerien tärkeysjärjestyksestä — jos esimerkiksi luotettavuus on tärkeämpi kuin nopeus.
- Luettelo toiminnoista, joita ei testata ja selvitys siitä, miksi niitä ei testata.
- Alustava testiaikataulu, jossa määritellään aikarajat. Aikataulu tulisi sitoa projektin kokonaissuunnitelmaan.

Testitapaukset

Kun testaussuunnitelma on valmis, seuraavaksi täytyy luetella testitapaukset käyttämällä samaa toiminnallisuuden jaottelua kuin suunnitteluvaiheen määrittelyissä. Jokaiseen tapaukseen tulee kuulua:

- Viittaus testattavaan kohteeseen.
- Odotettu testitulos.
- Kuvaus siitä kuinka testitulokset osoittavat, että testattava kohde toimii odotetulla tavalla.

Oppitunnin yhteenveto

Tällä oppitunnilla käsiteltiin ohjelmiston testaukseen liittyvää yleistä terminologiaa ja käytiin läpi tehokkaan testaussuunnitelman kirjoittaminen. Testaussuunnitelma kokoaa yhteen tiedot sovellukselle tehtävistä testaustoiminnoista, joihin kuuluvat:

- Yksikkötestaus lyhyille koodin osille.
- Integraatiotestaus, jolla varmistetaan, että koodin osat toimivat yhdessä.
- Järjestelmätestaus, jolla varmistetaan koko ohjelman yleinen vakaus.
- Rasitustestaus, jolla tutkitaan kuinka ohjelma reagoi toimiessaan järjestelmässä, jossa resurssit ovat rajoitetut.
- Regressiotestaus, jolla varmistetaan, että koodin uusi versio ei aiheuta lisävirheitä.
- Betatestaus käyttäjien palautteen saamiseksi.
- Hyväksymistestaus, jolla testataan käyttöä todellisessa tilanteissa.

Pienissä kehitysryhmissä, joissa on alle neljä tai viisi henkilöä, täytyy ohjelmoijien yleensä toimia myös testaajina. Laajemmissa organisaatioissa, joissa on enemmän henkilökuntaa, ohjelmoijat ja testaajat muodostavat kaksi erillistä ryhmää. Myös tällaisessa tapauksessa ohjelmoijat voivat hyötyä tässä luvussa olevista tiedoista, koska kehitystyö etenee jouhevammin, kun molemmat ryhmät ymmärtävät toistensa työtä.

Laboratorio 13: STUpload-sovelluksen debuggaus

Tässä osassa käytät Visual C++:n debuggeria STUpload-sovelluksen tutkimiseen sen suorituksen aikana. Laboratoriossa kokeillaan debuggerin Variables, Watch, Call Stack ja Disassembly ikkunoita ja havainnollistetaan, kuinka nämä ikkunat kaikki myötävaikuttavat virheiden poistossa. Asetat myös keskeytyskohtia eri paikkoihin ja kokeilet datakeskeytyskohtien vaikutusta.

Oppitunnin arvioitu kesto: 15 minuuttia

STUpload-sovelluksen ajaminen debuggerissa

Tässä harjoituksessa ajetaan STUpload-sovellusta debuggerissa kunnes MFC framework kutsuu sovelluksen **CMainFrame::OnCreate()**-funktiota. Tämä funktio suoritetaan ohjelman suorituksen alkuvaiheessa, ennen dokumenttien ja näkymien luomista ja ennen kuin STUpload-ikkuna ilmestyy näytölle. Tämän prosessin avulla saadaan mahdollisuus ajaa **OnCreate()**-funktio askel-askeleelta läpi ja nähdään kuinka MFC-ohjelma herää eloon.

Harjoituksen ensimmäinen tehtävä on kääntää ohjelman debug-versio, joka tarvitaan sen ajamiseen debuggerissa. Jos olet jo aiemmin luonut STUpload-sovelluksesta debug-version, siirry suoraan toiseen vaiheeseen.

► **Debug-version kääntäminen**

1. Avaa STUpload-projekti.
2. Valitse **Build**-välilehden suuremmasta muokkausruudusta **Win32 Debug**. Vaihtoehtoisesti voit napauttaa **Set Active Configuration** -komentoa **Build**-valikosta ja valita **Win32 Debug**.
3. Käännä ja linkitä koko sovellus napauttamalla **Build**-työkalurivin **Build**-painiketta tai napauttamalla **Build**-valikon **Build**-komentoa.

► **Suorituksen pysäyttäminen CMainFrame::OnCreate()-funktioon**

1. Avaa Visual C++:n Workspace-ikkunan **FileView**-välilehdeltä lähdetiedostojen luettelo. Avaa **MainFrm.cpp**-tiedosto luettelosta kaksoisnapauttamalla.
2. Etsi **OnCreate()**-funktion alku vierittämällä näkymää alaspäin. Alku näyttää lähdekoodissa seuraavalta:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

3. Aseta osoitin ensimmäisen kaarisulun kohdalle napauttamalla hiirellä. Tällä toimenpiteellä osoitetaan rivi, jolle ohjelman tulee pysähtyä.
4. Käynnistä debuggeri valitsemalla **Start Debug** -toiminto **Build**-valikosta ja valitse **Run to Cursor** -komento.

Debuggeri käynnistyy ensin ja sitten automaattisesti käynnistää STUpload-sovelluksen. Kun **CMainFrame::OnCreate()** tulee vuoroon, suoritus pysähtyy riville, jossa kohdistin on. Komentoa ei ole vielä suoritettu, mutta se on ensimmäisenä vuorossa ohjelman suorituksen jatkuessa.

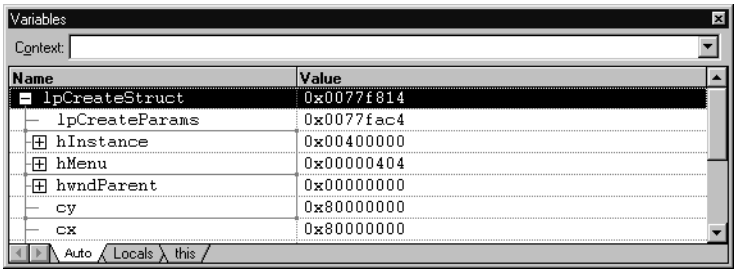
Vaikka mikään C++-koodi ei näytä olevan yhteydessä funktion alussa olevaan kaarisulkuun, koodia ei kuitenkaan ole olemassa ennen ensimmäistä C++-komentoa, kuten voit nähdä napauttamalla debugger-työkalurivin **Disassembly**-työkalua. Lähdekoodi-ikkunassa näkyvät assembly-kieliset komennot ovat funktiota alustavaa koodia, jossa pinon kehys luodaan ja luokan **this**-osoitin varastoidaan paikalliseen muuttujaan. Palauta normaali Source-ikkuna napauttamalla **Disassembly**-työkalua uudelleen.

Koodin tutkiminen askel-askeleelta

Tässä harjoituksessa näet, kuinka voit käydä **OnCreate()**-funktion askel-askeleelta läpi käyttämällä **Step Into**, **Step Over** ja **Step Out** -työkaluja. Harjoituksessa luetaan myös debuggerin Variables ja Call Stack -ikkunoiden näyttämiä tietoja samalla, kun koodissa edetään.

► **Variables ja Call Stack -ikkunoiden katsominen**

1. Jos Variables-ikkuna ei ole ruudulla näkyvissä, avaa ikkuna napauttamalla debugger-työkalurivillä olevaa **Variables**-työkalua. Telakoimattomassa tilassaan ikkuna näyttää samalta kuin kuvassa 13.13.

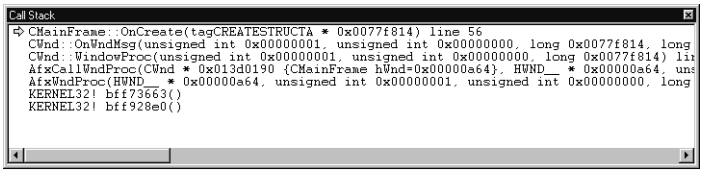


Kuva 13.13 Visual C++ -debuggerin Variables-ikkuna

2. Avaa lista napauttamalla plus-kuvaketta lpCreateStruct-nimen edessä Variables-ikkunassa. Koska funktion parametri on tällä hetkellä ainoa

muuttuja funktion pinossa, lpCreateStruct on ainoa ikkunassa lueteltu muuttuja. Se osoittaa **CREATESTRUCT**-rakenteeseen, joka koostuu erilaisista kentistä, jotka ilmestyvät Variables-ikkunaan, kun uusi näkymä laajennetaan. Näkymän laajentaminen näyttää, että esimerkiksi rakenteen **lpszName**-kenttä sisältää osoittimen sovelluksen nimeen.

3. Avaa Call Stack -ikkuna napauttamalla debuggerin **Call Stack** -työkalua (Kuvassa 13.14). Tässä ikkunassa nähdään, kuinka luettelon ensimmäisenä olevaan **OnCreate()**-funktioon on päädytty. Kutsujonon toinen funktio on kutsunut **OnCreate()**-funktiota ja suoritus palaa siihen **OnCreate()**-funktion suorituksen päätyttyä. Tässä tapauksessa kutsuja on **CWnd::OnWndMsg()**, joka on MFC frameworkin osa.



Kuva 13.14 Visual C++ -debuggerin Call Stack -ikkuna

4. Piilota Call Stack -ikkuna napauttamalla **Call Stack** -työkalua uudelleen.

► **OnCreate()-funktion läpikäyminen askeleittain**

1. Käy **OnCreate()**-funktion pari kolme ensimmäistä komentoa askeleittain läpi napauttamalla **Step Over** -työkalua tai painamalla F10-näppäintä. Aina käyttäessäsi **Step Over** -komentoa suoritus pysähtyy keltaista nuolta seuraavaan komentoon. Huomaa, kuinka Variables-ikkuna näyttää, kuinka sovelluksen tiedot muuttuvat kunkin käskyn seurauksena.
2. Jatka etenemistä askeleittain kunnes keltainen nuoli osoittaa riviä, joka asettaa sovelluksen työkalurivin telakoinnin:

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
```

3. Napauta **Step Into** -työkalua ja pääset frameworkin **CControlBar::EnableDocking()**-funktioon. Debuggeri etsii automaattisesti funktion lähdekooditiedoston Bardock.cpp ja avaa sen. Suoritus pysähtyy funktion ensimmäiselle riville.
4. Käy halutessasi **CControlBar::EnableDocking()** askeleittain läpi ja poistu siitä napauttamalla debuggerin **Step Out** -työkalua. Funktio suoritetaan loppuun ja ohjaus palautuu **CMainFrame::OnCreate()**-funktioon, suoritus pysähtyy **CControlBar::EnableDocking()** kutsua seuraavalle riville:

```
EnableDocking(CBRS_ALIGN_ANY);
```

Keskeytyskohtien asettaminen

Tässä harjoituksessa nähdään, kuinka keskeytyskohtia voidaa asettaa toiseen lähdekooditiedostoon. Keskeytyskohdat asetetaan yleensä ennen debuggerin käynnistämistä, mutta kuten tässä harjoituksessa näet, voit asettaa ja poistaa keskeytyskohtia koska vain, myös debuggerin ollessa toiminnassa.

► **Keskeytyskohdan käyttö tuotaessa tiedostoa**

1. Avaa STUploadDoc.cpp-lähdekooditiedosto ja etsi **CSTUploadDoc:: LoadData()**-funktio:

```
BOOL CSTUploadDoc::LoadData(CStdioFile &infile)
```

2. Etsi **LoadData()**-funktiosta **while**-silmukka, joka lukee luvussa 4 kuvatulla tavalla .dat-tiedoston rivit:

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

3. Siirrä tekstikohdistin **while**-silmukkaan napauttamalla sitä hiirellä.
4. Aseta keskeytyskohta painamalla F9. Pieni punainen piste ilmestyy osan marginaaliin osoituksena siitä, että rivi on merkitty keskeytyskohdaksi.
5. Napauta debuggerin työkalurivin **Go**-painiketta tai paina F5-näppäintä. STUpload-sovelluksen suoritus jatkuu ja keskeytetty **CMainFrame:: OnCreate()**-funktio suoritetaan loppuun. STUpload:in pääikkuna avautuu. Tässä vaiheessa debuggerin sijasta aktiivisena on STUpload-sovellus. Debuggeri aktivoituu vasta, kun ohjelma tulee keskeytyskohtaan.
6. Valitse ohjelman **Data**-valikosta **Import** ja etsi Test.dat-tiedosto \Chapter 4\Data-kansiosta. Kun avaat tiedoston, toiminta pysähtyy **CSTUploadDoc:: LoadData()**-funktioon asetettuun keskeytyskohtaan.
7. Napauta hiiren oikealla painikkeella seuraavaa riviä ja valitse pikavalikosta joko **Remove Breakpoint** tai **Disable Breakpoint**. Ensimmäinen komento poistaa keskeytyskohdan ja toinen poistaa keskeytyskohdan käytöstä, mutta ei hävitä sitä lopullisesti.

```
while(infile.ReadString(strTemp))
```

8. Käy muutamia komentoja läpi yksitellen ja seuraa Variables-ikkunasta, kuinka muuttujille kuten StrDate asetetaan alkuarvoja.

► **Data breakpointin asettaminen**

1. Avaa **Breakpoints**-dialogi napautamalla **Edit**-valikosta **Breakpoints**, tai painamalla CTRL+B.
2. Kirjoita **Data**-välilehdellä **bFirstLine**-muokkausruutuun **Enter the expression to be evaluated**. Tämä toiminto asettaa breakpointin, joka laukeaa vain function bFirstLine-muuttujan arvon muuttuessa.
3. Sulje **Breakpoints**-dialogi napauttamalla **OK**.
4. Jatka suoritusta napauttamalla **Go** kometoa.

Huomaat luultavasti pieniä taukoja **while**-silmukan jatkaessa toimintaansa, sillä data breakpointit verottavat suoritusnopeutta. Ohjelman suoritus pysähtyy joka tapauksessa pian, ja debuggeri näyttää viesti-ikkunan, joka kertoo, että bFirstLine-muuttujan arvossa on havaittu muutos. Napauttamalla viesti-ikkunan **OK**-painiketta saat näkyviin lähdekoodi-ikkunassa keltaisen nuolen, joka osoittaa komentoa, joka juuri tyhjensi bFirstLine-lipun.

Data breakpointien asettaminen bFirstLine:n kaltaisille yksinkertaisille muuttujille, joiden arvo muuttuu vain kerran, ei ole kovin hyödyllistä. Data breakpointit voivat olla kuitenkin hyödyllisiä, kun etsitään outoja virheitä, jotka aiheutuvat suorituksen aikana muuttujiin asetetuista vääristä arvoista ja kun ei voida muuten selvittää, missä vaiheessa muuttuja saa väärän arvonsa.

► **Suorituksen jatkaminen ja ohjelman päättäminen**

1. Avaa **Breakpoints**-dialogi painamalla CTRL+B, ja napauta dialogin alaosassa olevaa **Remove All** -painiketta. Tämä toimenpide poistaa bFirstLine-muuttujaan liitetyn data breakpointin varmistaen, että ohjelman suoritus jatkuu normaalilla nopeudella.
2. Jatka STUpload-sovelluksen suorittamista napauttamalla **Go** komentoa tai painamalla F5, ja sulje ohjelma normaalisti sen **Exit** komennolla. Kun ohjelma sulkeutuu, niin tekee myös debuggeri.

Kertaus

1. Mitä tarkoittaa Structured Exception Handling?
2. Mikä on HRESULT?
3. Nimeä kaksi tapaa, joilla COM-palvelin välittää tietoja virheestä asiakkaalle.
4. Kuinka sovelluksen debug-versio eroaa julkaisukäännöksestä?
5. Kuvaile MFC:n **ASSERT**, **VERIFY** ja **DEBUG_NEW** -makroja.
6. Kuinka debuggeri saa ajettavan ohjelman keskeyttämään toimintansa?
7. Mitä apuohjelmia Visual C++:ssa on ActiveX-kontrollien testaamista ja ajamista varten?
8. Mitä tarkoittaa riippuvuus?
9. Mitä tietoja sisältyy Spy++:n näyttämiin neljään listaan?
10. Mitä tarkoittaa regressiotestaus?