Sipos Marianna

...önálló tanuláshoz...

A Visual C++ és az MFC

Sipos Marianna

A Visual C++ és az MFC

SZAKMAI LEKTOR

Dr. Charaf Hassan

Zolnay András

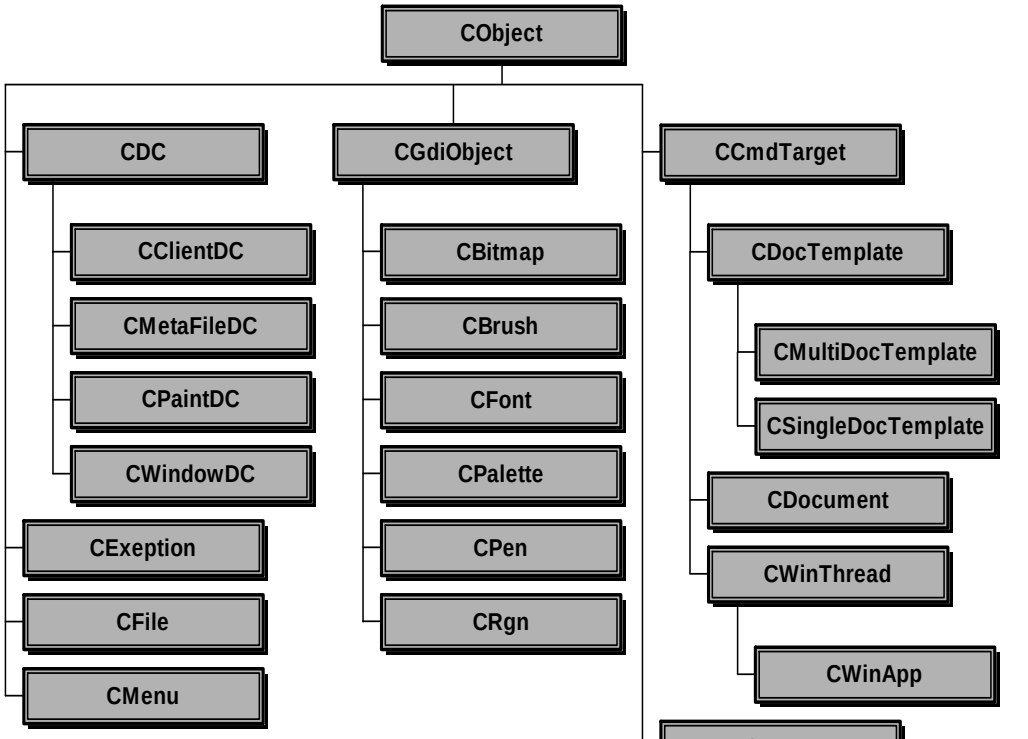
ANYANYELVI LEKTOR

Temesy Klára

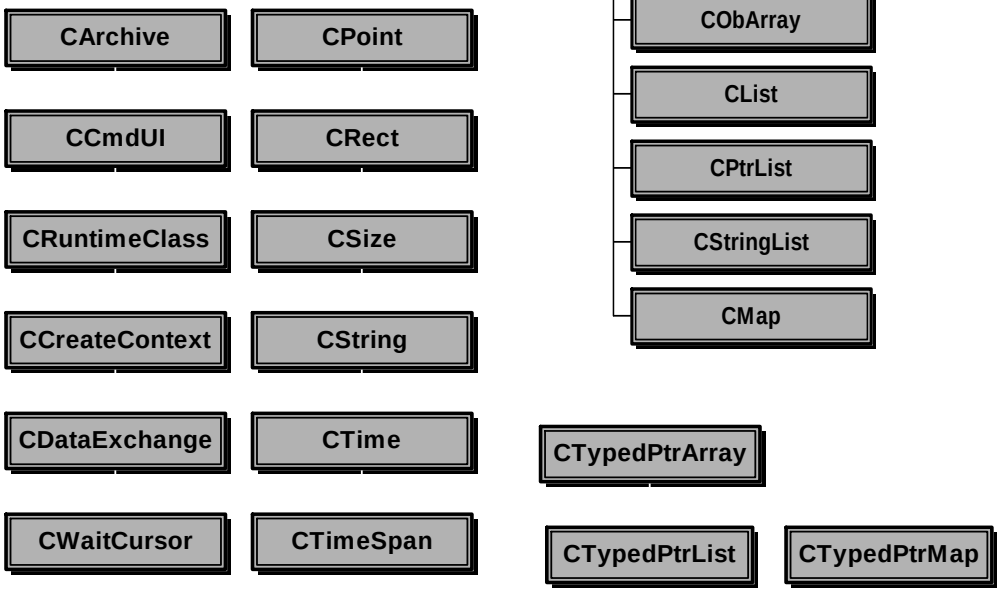
Második, javított, módosított kiadás, 2001. augusztus.

Kiadja:	Dr. Sipos Jenő
Szerkesztette:	Siposné Németh Marianna
Borítóterv:	Nagy Ágnes
ISBN:	963 640 979 X
Nyomdai munkák:	Akadémiai Nyomda, Martonvásár.

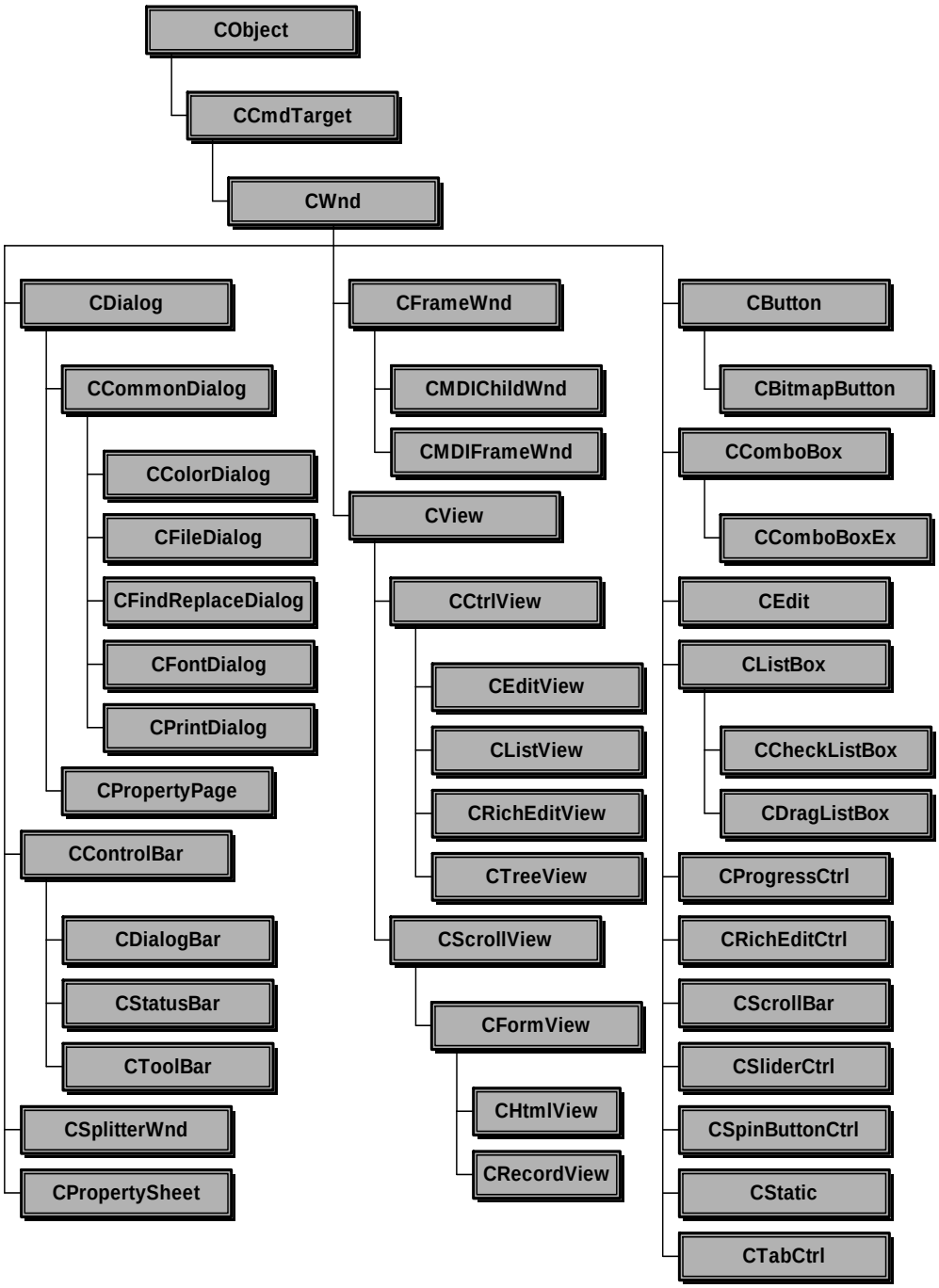
A könyvben szereplő MFC osztályok



Nem a CObject utódosztályai:



A könyvben szereplő MFC osztályok



Előszó

Egy jól használható tankönyv írásához elengedhetetlen a több éves oktatási gyakorlat az adott témában. E könyv készítését is megelőzte a tananyag különböző képzési formákban – nappali tagozatos szakképzés, tanfolyami képzés – történő többszöri gyakorlati kipróbálása. Sajnos az informatika gyors változása miatt nincs mód arra, hogy a könyvet olyan szinten teszteljük, ahogy azt egy hagyományos általános érvényű témában tehetnénk, mert mire elkészülne, a benne található tananyag már elavulttá válna. Remélem, sikerült megtalálnom azt a pontot, amikor már elegendő gyakorlati tapasztalat áll rendelkezésemre, és még mindig izgalmas témát jelent a könyv tartalma.

A könyv Visual C++ 6.0 fejlesztői környezetben, a háttérben Windows NT 4.0 operációs rendszerrel készült. Az oktatás során használtam a tananyagot Visual C++ 4.2-től, Win 95 és későbbi operációs rendszerek alatt is. A megadott konfigurációtól eltérő felhasználás esetén előfordulhat néhány megoldás kisebb módosítása. Amit ezek közül észrevettem, azt beépítettem a szövegbe.

További információk az ’Útmutató a könyv használatához’ fejezetben találhatóak.

A könyvet egyaránt ajánlom diákoknak és gyakorló programozóknak, akik az objektumorientált programozás alapfogalmaival már megismerkedtek, és új programfejlesztő környezetként a Visual C++-t választották.

Ezúton kérem az olvasó elnézését a könyvben mégis fellelhető esetleges hibákért!

"Always make new mistakes!"
(Dyson, 1998.)

Előszó

Szeretnék köszönetet mondani családomnak, akik az utóbbi hónapokban csak a hátamat látták a dolgozószoba ajtajából, s a rám vonatkozó mondatok többsége így kezdődött: "Majd, ha anyu végez a könyvével...". Köszönet illeti szüleimet, akik megtanították, hogy az eredményekhez kitartó munkára van szükség, és a könyv kiadása is a bővebb család összefogásának köszönhető.

A szakmai munkában elsősorban szakmai és anyanyelvi lektoraim segítségét köszönöm, akik nagyon alapos elemzéssel, kritikus véleménnyel, tanácsokkal járultak hozzá a könyv tartalmához, segítették elő az érthetőséget, az áttekinthetőséget. A feladatok megoldását tanítványaim: Bihari Sándor és Szabó Péter tesztelték lelkiismeretesen. Köszönet illeti azokat a tanítványokat, olvasókat is, akik immár az első kiadást használva felhívták figyelmemet a kevésbé érthető részekre, esetleg módosításokat javasoltak a szövegbe, a feladatok megoldásába.

Angster Erzsébet és Baga Edit, akik SZÁMALK oktatóként kollégáim és a könyvírásban elődeim voltak, példát mutattak, tanácsokkal láttak el és bátorítottak. Hálás vagyok Gregorics Tibornak, tőle tanultam a programozás első lépéseit, ő szerettette meg velem új szakmámat; és Illés Zoltánnak, akinek találkozásomat köszönhetem a Visual C++ nyelvvel. A programozásoktatás elméletével Zsakó László, Szlávi Péter és az ELTE tanárai ismertettek meg. Az új tananyagok fejlesztésekor ma is emlékezetembe idézem tanácsaikat. A gyakorlatban pedig a Neumann János Számítástechnikai Szakközépiskola és Gimnázium tanári kara nyújtott kiváló szakmai és emberi segítséget a kezdéshez. Koncz Mártit és kollégáit ma is inkább érzem barátaimnak, mint egykori munkatársaknak.

Ezúton szeretnék köszönetet mondani Biszterszky Elemér professzor úrnak azért, hogy a könyv és a hozzá tartozó tananyag elkészítésekor a BME Szakképzés-pedagógiai doktori (PhD) program professzorainak és doktorandusz társaimnak szakképzésre, távoktatásra vonatkozó kutatási eredményeit is figyelembe vehettem. A hazai és nemzetközi konferenciák előadásai elősegítették, hogy az objektumorientált programozás olyan szakembereit sorolhatom az ismerőseim, sőt barátaim közé, mint Juhász István és Ivar Jacobson, akiktől ma is több száz, több ezer kilométerre dolgozom, és akiknek már megismerkedésünk előtt nagy csodálója voltam. Az emberi kapcsolatok a szakmai munkát is elősegítik, inspirálják.

Kedves Olvasó!

Jó tanulást, jó programozást kívánok!

Budapest, 2001. augusztus.

Sipos Marianna

Tartalomjegyzék

GYAKORLATOK

Útmutató a könyv használatához

A könyv két fő részre osztható, elméleti rész és gyakorlati rész. **Az adott elméleti résszel megegyező sorszámú gyakorlatok az elmélethez kapcsolódó feladatokat és megoldásokat tartalmazzák.**

Eredeti terveim szerint a könyv része lett volna egy objektumorientált programozás fejezet is, de terjedelmi okok miatt ebből önálló kötet készül: 'Objektumorientált programozás a C++ nyelv lehetőségeivel' címmel. (Az azonos című jegyzet az önálló kötet egy részét tartalmazza.) A könyv OOP hivatkozásai e kötet fejezeteire vonatkoznak.

Úgy gondolom, ma nem célszerű a Visual C++-hoz magyar nyelvű referenciakönyvet írni. Egyszerűen a hatalmas méretű, de mindenki számára rendelkezésre álló súgó helyettesíti azt, ráadásul oly gyorsak a változások, hogy a könyv ára nem lenne arányos használhatóságával. **A könyv a Visual C++ 6.0 felhasználásával készült.** Könyvemben néhány helyütt úgy éreztem, elengedhetetlen bepillantást adni a lehetőségekbe, pl. 2.2. gyakorlat window styles, de itt is csak az alapeseteket írtam le, az extended styles már nem szerepel, s az egyéb utódablakok stílusaira sem került sor (Button styles...) Úgy gondolom, ezek a hosszadalmas leírások részekre tördelik mondanivalómat, rontják a lényegre koncentrálást és csekély haszonnal járnak, hisz a gép előtt ülő programozó bármikor elérheti őket a súgóban. Ugyanakkor **részletes tárgymutató tartozik a könyvhöz**, melyben a tárgyalt fogalmak, rövidítések osztály- és metódusnevek... felsorolásra kerülnek, lehetővé téve a könyv kézikönyv jellegű használatát. Az ilyen típusú

felhasználást támogatják az önálló, kis méretű feladatok, melyek megértéséhez nincs szükség az előző fejezetek végrehajtására.

A könyv szerkezete lehetővé teszi az egyénre szabott felhasználást.

- **Azok, akik előbb az elmélet**et szeretik megtanulni, **aztán** alkalmazni **a gyakorlat**ban, megtehetik, mert az elméleti rész a feladatoktól függetlenül íródott. Minden elméleti fejezet végén kérdések és összefoglalás segíti az elmélyítést. Az elméleti rész számítógéptől független, akár pl. utazás közben is olvasható.
- **Aki viszont inkább a gyakorlat oldaláról közelíti a feldolgozást**, a feladatok megoldása során **visszalapozhat** az azonos sorszámú fejezethez.

A gyakorlatok elején mindig szerepel legalább egy **lépésről lépésre megoldott feladat**, melynek célja nemcsak egy megoldás részletes ismertetése, hanem az **eszköz**ök **használat**ának bemutatása is. A kész forráskódból nem biztos, hogy kiderül, az adott kódrészletet az osztályvarázslóval, a workspace ablakaival, vagy kézzel írtuk meg a kódot. A szövegszerkesztő a kód írásakor nemcsak a nyelv kulcsszavait ismeri fel és írja más színnel, hanem a tabulálást is automatikusan végzi. A '{' után beljebb kezdi a kódsorokat és a '}' automatikusan az utolsó '{' alá kerül. Az is látható a lépésenként végrehajtott feladatból, **hogyan oldjuk meg először egyszerűbben aztán részletesebben az adott feladatot.**

Van, aki az önálló munkát kedveli. Megoldhatja önállóan is a gyakorló feladatokat, s ha elakad, segítségért fordulhat a könyvhöz, vagy más ötletet láthat a megoldásra.

Ahhoz, hogy érezzük, ismerjük a nyelvet, tudunk bánnia a fejlesztőeszközzel, elengedhetetlen az **önálló munka**. **A gyakorlati fejezetek végén** található feladatok az adott tudásszinttel megoldható önálló munkát támogatják. A *-gal jelölt **feladatok megoldás**ához **ötlete**ket találunk a rákövetkező szakaszban. Ezek néha csak egy-két mondatot, máskor viszonylag részletes megoldást, vagy a forráskód közlését jelentik. Az ötleteket csak akkor nézze meg, ha már saját tudásával és a súgó segítségével megpróbálta megoldani a feladatot.

Ne feledjük! A "korlátlan" lehetőségek világában járunk. Más speciális területek gyors fejlesztésére kidolgozott fejlesztőeszközök a lehetőségük határára érve azt mondják, e probléma megoldása nem áll módjukban. **A C++ nyelv annak köszönheti közkedveltségét, hogy ha egy probléma egyáltalán megoldható**, akkor – bár lehet, hogy küzdelmes munkával – de jó eséllyel **megvalósítható benne**.

A könyv feladatainak forráskódja a

www.gdf.hu/segedletek.htm

Internet címen a **Visual C++ alapjai címszó alatt** érhető el, vagy a következő ftp címről tölthető le:

ftp://ftp.gdf.hu/public/vcpp

➢ A könyvben használt szakkifejezések

Amikor az angol szavak magyar megfelelőjét kerestem, gyakran éreztem együtt Shopenauerrel, aki azt mondta: ***"A gondolat elhal, mihelyt szavakban ölt testet."***

A könyv magyar nyelvű, tehát a szakkifejezések is magyar nyelven szerepelnek benne. Azonban **a szoftver**, amiről szól, **nem rendelkezik** (várhatóan a jövőben sem fog) **magyar nyelvű változattal**, így elengedhetetlen az angol menüpontok, a súgóban megtalálható kifejezések használata. **A Visual C++ ismerőjének mind a magyar, mind az angol kifejezésekkel gyakorlottan kell bánnia, így az angol szavakat gyakran szinonimaként használja a szöveg**.

Több olyan angol kifejezés van, aminek már létezik magyar megfelelője, de az vagy nem egészen takarja a fogalmat, vagy nem használatos még általánosan. Például a **default** konstruktor lehetne **alapértelmezett**, de ez könnyen összekeverhető lenne az alapértelmezett paraméteres konstruktorral, (mely default is, de nem csak az), így inkább a default szót használom. Vannak olyan fogalmak, melyeket a különböző szakirodalom más-más néven nevez. Például a **property** sok Delphi könyvben **jellemző**-ként szerepel, míg a magyar nyelvű súgók **tulajdonság**nak fordítják. Itt egyértelmű volt a tulajdonság szó választása.

Az **overloading** fogalmát szokás **túlterhelés**nek, de **átdefiniálás**nak is nevezni. Sok helyütt túlterhelésnek fordították, de ez a szó tartalmilag nem egészen azzal a jelentéssel bír, mint az overloading. Angolul a terhelhetőség fokozódásáról van szó (többet tud, mint az egy paraméterlistájú változat). A magyar túlterhelt tartalma más, már nagyon sokat vállal, s nem bírja tökéletesen ellátni feladatát. A közelmúltban megjelent Kondorosi, László, Szirmai-Kalos: Objektumorientált szoftverfejlesztés könyv (Kondorosi, 1998, 222.old.) talán épp emiatt inkább az átdefiniálást használja. **A függvények értelmezésének kiterjesztése** kifejezéssel él a Benkő, Benkő, Poppe: Objektum-orientált programozás C++ nyelven (Benkő, 1998, 190. old.), és maradnak a túlterhelés mellett a Young: Visual C++ 6.0 mesteri szinten (Young, 1998/I. 106. old.) fordítói. Ráadásul mindez kiegészül az **overriding** fogalmával, ami **felülírás**ként, **felüldefiniálás**ként használatos. **A szintaxis minden esetben hasonló.** Ha egy függvénynek, operátornak több különböző argumentumlistájú változatát adjuk meg, ez overloading, túlterhelés, átdefiniálás. Ha ezt a függvényt az öröklés során írjuk felül az utódosztályban, az overriding, felüldefiniálás, mely az ősosztály különböző argumentumlistájú függvényeit eltakarja. De egy osztályon belül több különböző argumentumlistás függvény megadása overloading, átdefiniálás.

Vannak olyan magyar szakkifejezések, melyek C++ környezetben más tartalommal bírnak. Ilyen a **statikus** függvény, melyet az Objektumorientált tervezés és programozás könyv (Angster, 1997) a **nem virtuális** függvény megjelölésére használ. A C++ **static** kulcsszó az osztályszintű függvényeket jelöli. A Delphi

programozók a **vezérlő**ket **komponens**eknek nevezik (Baga, 1998). Az **ActiveX** vezérlők valóban komponensek, de Visual C++-ban, ha komponensről beszélünk, az ActiveX-nél általánosabb fogalomra, **COM** komponensekre gondolunk.

A **framework** kifejezésnek nem találtam megfelelő kifejezést. Szó szerint fordítva keretmunkát jelent, tartalmilag az MFC által biztosított, a háttérben elvégzett tevékenységet. A magyar keretmunka szó negatív politikai hátterű. Inkább nem fordítottam le a kifejezést. Ellenkező az eset a **gyorsmenü**vel, melyre több magyar megfelelőt is használnak: **felbukkanó**, **előbukó**, **lebegő**, **helyi menü**. De angolul sem egyértelmű a kifejezés, hisz a komponensgalériából **pop-up** menüt választunk, hogy megvalósítsuk a **shortcut** menüt.

Egyik megoldási lehetőség, hogy több kifejezés esetén az egyiket kiválasztva, következetesen azt használja a könyv. Előnye az egyértelműség, hátránya, hogy más szakirodalomban nehezen ismeri fel az olvasó a fogalmat, valamint a sok szóismétlés, ami szakirodalomban bocsánatos bűn lehet. Egy másik megoldás, hogy mondandónk és a magyar nyelv változatossá tétele érdekében használjuk a különböző elnevezéseket. Hátránya, hogy ez az érthetőség rovására mehet.

A könyv írásakor igyekeztem mindig a megfelelő megoldást választani.

Sipos Marianna

Bevezetés

A bevezetőben Jeffrey Richter gondolatait idézném, melyeket az Inside Disrtributed COM könyv (Eddon, 1998) előszavában találunk, s melyek rámutatnak az informatika hihetetlen gyors fejlődésére s arra a problémára, milyen nehézségek előtt áll a programozó, amikor választania kell, merre induljon vagy merre haladjon tovább.

A Windows operációs rendszer óriásit fejlődött az utóbbi tíz évben. Eleinte könnyű volt áttekinteni és megérteni az egész rendszert, amely csak néhány száz függvényből állt. A Windows 2.11 (Kernel 283, User 141 és a GDI 213) összesen 637 függvény ismeretét követelte. 10 év alatt a Microsoft óriási mértékben kiterjesztette moduljait, és számos alrendszert épített be a Windows-ba. (Az alrendszerek nevei angolul szerepelnek, mert legtöbbjüknek nincs magyar megfelelője) pl. telephony, remote access, print spoolers, 3-D graphics, Internet access, security, registry, services, multimedia, networking és így tovább. Ma lehetetlen egyetlen személy számára a teljes operációs rendszer megértése, és akkor még nem beszéltünk más operációs rendszerek programozásáról. Tehát specializálódnunk kell. Ez persze nem jelenti, hogy a rendszer más részeit el kellene hanyagolni.

E kötet a Windows-programozás alapjainak elsajátítását tűzte ki céljául, mely a későbbiekben bármely modul mélyebb megismerését megalapozza. A fejezetek kódjának s a kód működésének megértéséhez elengedhetetlenül szükséges az objektumorientált programozás fogalmainak ismerete. Ezért készült – az eredetileg a könyv függelékébe szánt, majd számos okból külön kötetben kiadásra kerülő –

Objektumorientált programozás a C++ nyelv lehetőségeivel könyv (ezentúl **OOP**), melynek első hat fejezete Gábor Dénes Főiskolai jegyzetként már megjelent. E kötet ismerteti az objektumorientált programozási szemléletmód alapfogalmait, lehetővé téve azok számára is a megértést, akik még nem találkoztak ezzel a fogalomkörrel. Az olvasók többsége úgy gondolom – nem ebbe a kategóriába sorolható. Lesznek, akik más programozási nyelv segítségével ismerték meg az objektumorientált programozás alapfogalmait, vagy nem minden e könyvben felhasznált részletével találkoztak ennek a filozófiának. Az OOP kötet az ő használatukra is készült, ahol utánanézhetnek a C++ nyelv más nyelvekhez képest többnyire jelentősen bővebb lehetőségeinek, más szemléletének. Ahhoz, hogy el tudjunk indulni a könyv anyagának feldolgozásával, elengedhetetlen az OOP első és harmadik fejezetének, az alapfogalmaknak és az öröklésnek az ismerete. A gyakorlatok során az osztálydiagramok elkészítéséhez pedig a Visual Modelerről szóló OOP 2. fejezetet ajánlanám az olvasó figyelmébe. A többi OOP fejezet a későbbiekben is elsajátítható.

A terjedelmi korlátok a könyv tartalmát is korlátozzák. **E kötet egy sorozat második része**, első rész a fent említett OOP könyv. További kötetek foglalkoznak majd az osztott alkalmazások fejlesztésével és az adatbázis-kezelés lehetőségeivel Visual C++-ban.

1. A Visual C++ bemutatása

Amikor a programozó programozási nyelvet választ, meg kell válaszolnia önmagának néhány kérdést. Ezek közül áll itt kettő.

Miért épp Windows alatt?

- Egyre több gépen fut Windows operációs rendszer.
- Windows alatt **meghajtó független programok** írhatók. Vagyis, amikor a programot írjuk, nem kell odafigyelnünk, hogy milyen típusú az egér, a monitor, a hangkártya, a nyomtató, a CD-meghajtó…
- **A Windows tartalmazza a meghajtókra vonatkozó kódrészleteket**, így ezekre nincs szükség programunkban.
- A kész alkalmazásnál adott **a felhasználó megszokott környezete**. Az általunk írt alkalmazásban is ismeri az ikonok nagy részének jelentését, tudja, milyen menüpontot hol keressen, tudja, hogyan kezelheti az ablakokat.

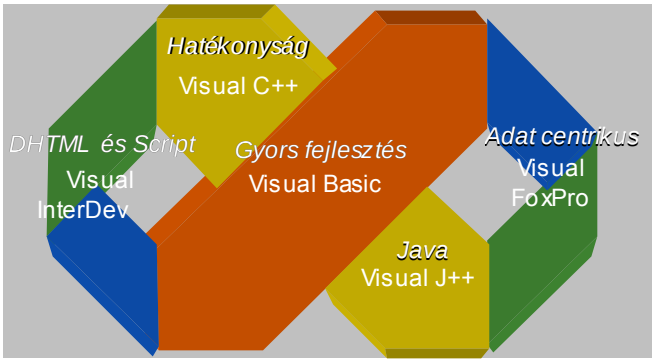
Miért épp Visual C++?

- A **C++ nyelv**re épül, az objektumorientált fejlesztést támogatja.
- **Vizuális felület**et biztosít a program írásához, egy negyedik generációs nyelv lehetőségeit kínálja a kód elkészítéséhez.

  ↳ Lehetőség van keretprogram generálására az alkalmazásvarázsló (AppWizard) segítségével.

- A képernyőkép és a vezérlők kényelmesen tervezhetőek az erőforrás nézetben (ResourceView).
- Az osztályok gyors és áttekinthető létrehozását, illetve módosítását az osztály nézet (ClassView) és az osztályvarázsló segíti.
- Támogatja **a komponens alapú programozás**t.

- "**We use it before you do**" elve, ami a gyakorlatra lefordítva azt jelenti, hogy a Visual C++-t Visual C++-ban fejlesztik.
- Operációs rendszer is írható vele. A Windows fordításához a Visual C++ fordítóját használják.
- Állandó fejlesztés alatt áll, így egyre hatékonyabb, és az újabb lehetőségek azonnal beépülnek. (Ez persze azt is jelenti, hogy kb. 2 évente új verziót kellene megvásárolnunk.)
- A **Visual Studio** szoftverfejlesztő eszközcsomag része, amely további a fejlesztést támogató eszközöket biztosít számunkra.



1.1. ábra A Visual Studio elemei

Mi a Visual C++ feladata a Visual Studioban?

- A hatékony és rugalmas programfejlesztést támogatja.
- Hatékony COM komponensek fejleszthetőek vele.
- Az MFC lehetőségeinek maximális kihasználását teszi lehetővé.
- Az Active Template Library (ATL) maximális kihasználását teszi lehetővé.

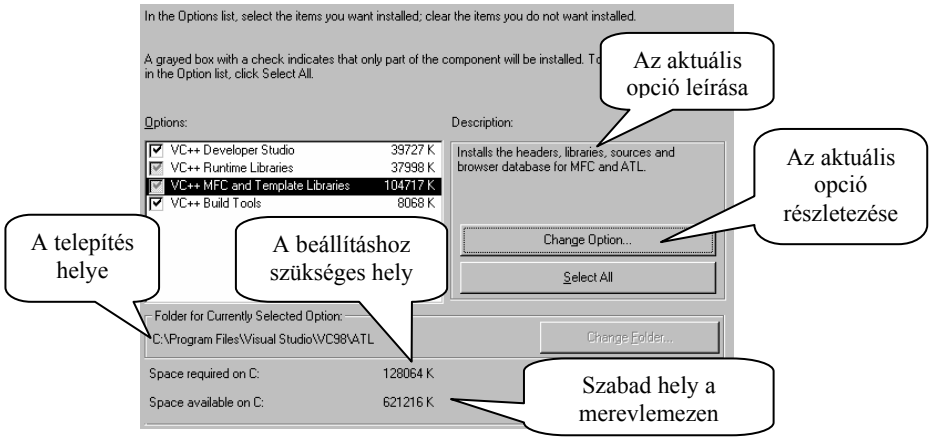
1.1. A Visual C++ telepítése

A Visual C++ önállóan is megvásárolható, de a Visual Studio részeként is hozzájuthatunk.

A ma használatos (3.0-nál későbbi) verziói 32 bites operációs rendszert igényelnek, tehát Windows 95, vagy újabb verziót; illetve Windows NT 3.51, vagy újabb verzió szükséges. Az első verziók még 16 bites Windows-hoz készültek. 16 bites fejlesztéshez használjuk a Visual C++ 1.5x-et!

Ha számítógépünkön engedélyezve van a CD AutoRun, akkor a CD-lemezt a CD-meghajtóba helyezve elindul a telepítő program. Ha nem indul automatikusan, akkor a CD-meghajtó gyökérkönyvtárából indítsuk a setup.exe-t. A telepítő párbeszédfelülete lehetővé teszi, hogy igényeinknek megfelelően telepítsük a programot.

A felajánlott lehetőségek függnek attól, hogy a Visual Studio részeként, vagy önálló szoftverként vásároltuk meg a Visual C++-t. Visual Studio részeként először a Visual C++-t kell választanunk. Más és más komponenseket kapunk, ha a Standard, a Professional vagy az Enterprise változatot vásároltuk meg. (1.2. ábra)



1.2. ábra A telepítő párbeszédablaka

Minden telepítendő opció mellett találunk egy **jelölőnégyzet**et, melynek többnyire három állapota lehet; üres, szürke pipa, fekete pipa. Ha üres a jelölőnégyzet, azt az opciót nem választottuk, ha fekete, akkor minden hozzátartozó részt kiválasztottunk. Ha szürke pipa van az aktuális opció jelölőnégyzetében, akkor választhatóvá válik a **Change Option** nyomógomb. Az egérrel rákattintva egy újabb hasonló szerkezetű párbeszédablakkal találkozunk, mely az előbb aktuális opció részletezése. A **Select All** gomb segítségével a teljes telepítést állíthatjuk be. A **Description** az aktuális komponens leírását adja. A **Folder for Currently Selected Option** tájékoztat bennünket az aktuális opció telepítésének helyéről a könyvtárstruktúrában. A helyet a **Change Folder** nyomógomb segítségével módosíthatjuk.

Az opciók jobb oldalán az adott opció telepítéséhez szükséges méret látszik, a párbeszédablak alján tájékoztatást kapunk a beállított telepítés szükséges méretéről és a meghajtón elérhető szabad terület méretéről. A hatékony fejlesztés érdekében

legalább 50Mb szabad helyet hagyjunk a meghajtón, s ne feledkezzünk meg az 1.1. szakasz végén ismertetett súgó telepítéséről sem!

E könyv használatához elegendő az előre megadott beállításokat telepíteni.

Látványos grafikák készítéséhez esetleg érdemes a Graphics opcióból válogatni, melyben metafájlok, bittérképek, kurzorok, ikonok és videóklippek találhatók jelentős mennyiségben. (Pl. közel 100 kurzor és több mint 450 ikon)

Új elemek telepítéséhez később is bármikor lefuttatható a telepítőprogram.

Megjegyzés:

A 6.0-ást megelőző verzióknál a telepítő felajánl úgynevezett 'Typical, Minimum, CD-ROM, Custom' változatokat. Mivel párbeszédfelületünkön alapértelmezett beállításokat találunk, a jelenlegi telepítő a Custom és Typical esetet magában foglalja, a többit pedig a hardver gyors fejlődése miatt fölöslegesnek tekinti. Természetesen mi bármely alapbeállítást felülbírálva szükség szerint csökkenthetjük vagy növelhetjük a telepített alkalmazás méretét.

A régebbi verziók a súgót is magukban foglalják, mely a Custom + Help opciók kiválasztásával telepíthető, vagy a telepítő CD-ről érhető el.

A 6.0 verziótól a súgó nem része a telepítő csomagnak. Regisztrált felhasználók számára az Internetről letölthető, vagy CD-ken illetve DVD-n hozzáférhető az **MSDN** (Microsoft Developer Network) Library csomag, mely a teljes Visual Studio-hoz tartozó súgót (**SDK**-val Software Development Kit együtt) tartalmazza. Előnye, hogy a súgó egy önálló alkalmazásként önálló főablakban fut. A túl sok információ szűréséhez egy kombipanel áll rendelkezésünkre, melyhez magunk is készíthetünk egyéni szűrés beállítást.

Az MSDN telepítése önállóan vagy a Visual C++ telepítőből indítható. A telepítés elve a Visual Studio telepítőjéhez hasonló, és a beállításoktól függően újabb jelentős helyet foglal a merevlemezen. A súgó használata elengedhetetlen alkalmazásaink fejlesztése során. Mérete is mutatja, rengeteg információt tartalmaz, melyet lehetetlen fejben tartani – így minél részletesebb telepítése javasolt. A nem telepített elemek a CD-kről érhetők el.

A Visual C++-ról aktuális információkat szerezhetünk a http://www.microsoft.com/visualc honlapról.

1.2. A Visual C++ fejlesztői környezet

A Visual C++ a fejlesztéshez olyan nagy számú lehetőséget biztosít, hogy az alapozás során ezek ismertetése értelmetlen lenne, majd elsajátítjuk használatukat a

feladatok megoldása során. E szakasz azért került mégis a könyv elejére, hogy segítse a kezdeti tájékozódást a programozó számára.

A következő listában láthatunk néhány általunk használt eszközt:

- AppWizard
- ClassWizard
- WizardBar
- Kód- és erőforrás-szerkesztők
- Workspace ablak
- Fordító és szerkesztő
- Debugger

Mit értünk **varázsló** alatt? Netán a mesék varázsszavakat mormoló figuráját, aki varázspálcájával egy pillantás alatt valósággá változtatja a lehetetlent? Vagy David Copperfieldhez hasonló valamit, aki nagyon is valóságos fizikai törvényeket felhasználva kápráztatja el a közönségét? A Visual C++ varázslói jóval kevesebbel is megelégszenek, hisz "csak" munkánk rendszeresen ismétlődő, mechanikus résztevékenységeit végzik el helyettünk. **A varázsló egy speciális formája a felhasználó támogatásának, mely párbeszédablakok segítségével végigvezeti az alkalmazás fejlesztőjét a nehéz és összetett feladatokon**. A mi varázspálcánk az egér, s a varázsszavakat a párbeszédablakok szerkesztődobozaiba kell begépelnünk, vagy a jelölőnégyzetek kiválasztásakor kell megadnunk. Az eredmény mégis varázslatos! Elkészül a kód, melyet unalmas lenne begépelni. Így hát hálásak vagyunk a varázslatokért, melyek egy Demo esetén hasonlíthatnak David Copperfield előadásához, a hétköznapokban viszont rendszeres, megbízható segítséget, kényelmesebb és hatékonyabb munkát biztosítanak számunkra.

1.2.1. Használt varázslóink:

- AppWizard (Application Wizard) (alkalmazásvarázsló)
- ClassWizard (osztályvarázsló)

WizardBar (varázslósor)

1.2.1.1. Az alkalmazásvarázsló

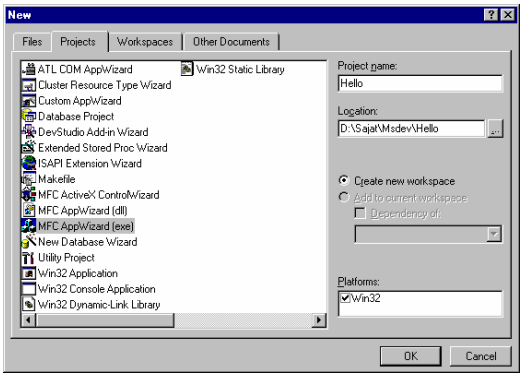
Az **alkalmazásvarázsló** feladata az alkalmazás vázának elkészítése. Beállításaink alapján elkészíti az alkalmazás "csontvázát", a súgó **skeleton files**-ként utal rájuk.

Hogyan érhetjük el az alkalmazásvarázslót? (1.3. ábra):

File menü

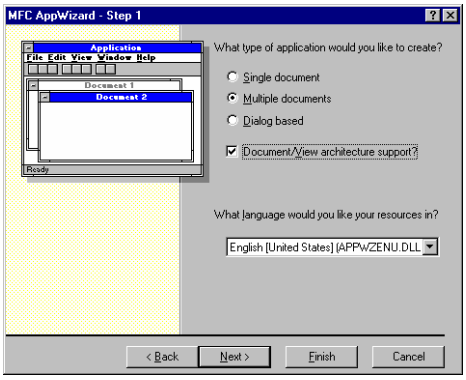
New menüpont

Projects fül



1.3. ábra Az alkalmazásvarázsló elérése

A projekt nevének és útvonalának megadása után elindul a varázsló (1.4. ábra). Az alkalmazás típusának megválasztásától függően a különböző lépésekben (párbeszédablakokban) a megfelelő opciók beállítása után elkészül az alkalmazás csontváza.

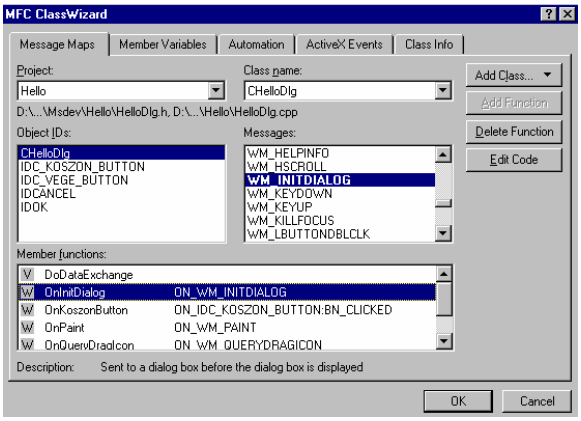


1.4. ábra Az alkalmazásvarázsló első ablaka

Egy egyszerű alkalmazás elkészítésének teljes leírását az 1.1. gyakorlat tartalmazza. Az 1.4. ábrán látható típusok, a hozzájuk tartozó lépések, 'skeleton' fájlok, osztályszerkezet leírását a megfelelő típusok tárgyalásánál találjuk.

1.2.1.2. Az osztályvarázsló 

Az osztályvarázsló egy többablakos párbeszédfelület, melyen több fül biztosítja a tájékozódást meglevő osztályainkról, az új osztályok létrehozását, módosítását. (1.5. ábra)



1.5. ábra Az osztályvarázsló

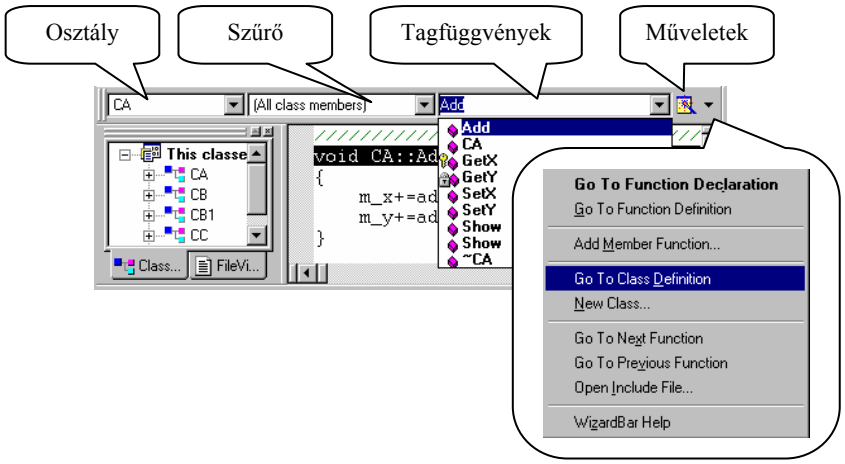
Az osztályvarázslót e könyvben a következő műveletek elvégzésére használjuk:

- Az MFC-ből származó új osztályok létrehozása.
- Új üzenetkezelő illetve virtuális tagfüggvények létrehozása, a meglevők törlése.
- Az üzenetek és az osztályokban létrehozott üzenetkezelők kapcsolatának vizsgálata, a kurzor az aktuális tagfüggvény kódjára állítása, az osztályról információk szerzése.
- A párbeszédfelületen a vezérlőkhöz hozzárendelt adattagok (más szóhasználatban tagváltozók) létrehozása, törlése, értékhatár-ellenőrzés beállítása.

COM komponensek esetén további lehetőségeket kínál az osztályvarázsló.

A varázslósor

A varázslósor nem önálló varázsló, az osztály varázsló egy felülete, mely néhány funkció kényelmes és gyors elérését teszi lehetővé. A varázslósor az eszközsor egy eleme. Ha nem látjuk az eszközök között, mint bármely más eszköztárat az eszközsor bármely mezőjén jobb egérrel kattintva a helyi menüből kiválaszthatjuk a WizardBar eszköztárat (DialogBar). Az 1.6. ábra a varázslósor elemeit mutatja.



1.6. ábra A varázslósor

A varázslósor szolgáltatásai:

- Kiválaszthatjuk az aktuális osztályt.
- Kiválaszthatjuk az aktuális metódust.
- Váltogathatunk a deklaráció (.h) és a definíció (.cpp) között.
- Létrehozhatunk új osztályt, tagfüggvényt.
- Megnyithatjuk az include fájlt (a beszerkesztett fájlok listájából kiválasztva).

A műveletek (Actions) legördülő menü az 1.6. ábrán látható lehetőségeket kínálja a felhasználónak.

Ha aktuálisan a kiválasztott függvény definícióján állunk, akkor a menüben a 'Go To Function Declaration', tehát az 'ugorj a függvény deklarációjára' hajtódik végre, ha a deklaráción állunk, akkor a definícióra ugrás. Ez azért fontos, mert a varázspálcára kattintva ez a művelet hajtódik végre. Tehát ezzel az eszközgombbal gyorsan váltogathatunk a különböző (.h, .cpp) fájlok különböző soraiban található függvény deklarációja és definíciója között.

Az osztályvarázsló funkcióihoz további elérési lehetőségeket biztosítanak a ClassView ablak és az erőforrás-szerkesztőben a párbeszédfelületek szerkesztésekor elérhető gyorsmenü.

1.2.2. A Workspace ablak

A workspace lehetővé teszi, hogy egyszerre akár több projekten (.dsp) is dolgozzunk. Együtt módosíthatjuk alkalmazásunkkal a hozzá tartozó .dll állományokat is. Még a projektek közötti függőséget, azaz fordítási sorrendet is beállíthatjuk (dependencies).

A .dsp fájl, a make fájl utóda. Egy fordítási egységet fog össze, tehát segítségével állíthatunk elő .exe fájlokat, és .dll-eket. Tárolja, hogy milyen forrásfájlokat kell lefordítani, milyen sorrendben és hogyan kell őket összeszerkeszteni (linkelni).

A konfigurációk (configuration) a .dsp fájl önálló részei. Akárhány konfigurációt definiálhatunk. A két alapvető a Debug és a Release. (Lásd még 1.2.5. szkasz!)

Nézzük meg elkészült alkalmazásaink .dsp fájlját szöveges állományként megnyitva (Open as: text)!

1.7. ábra A Workspace ablak három füle

- **ClassView** (osztály nézet) Az osztályviszonyok megjelenítése a feladata. Mutatja az osztályok tagfüggvényeit, adattagjait, azok hozzáférési szintjét. Támogatja tetszőleges új osztály felvételét vagy más projektben már létrehozott osztály beszúrását a projektbe. Az MFC osztályok utódait többnyire osztályvarázslóval hozunk létre.

A ClassView jelei megegyeznek a Visual Modeler UML jelöléseivel (OOP2. fejezet):

 Osztály

 Privát (private) tagfüggvény

 Védett (protected) tagfüggvény

 Nyilvános (public) tagfüggvény

 Privát (private) adattag (tagváltozó)

 Védett (protected) adattag

 Nyilvános (public) adattag

 Interfész

- **ResourceView** (erőforrás nézet) A projekt erőforrásait mutatja, lehetővé téve kényelmes szerkesztésüket, új erőforrások felvételét.

- **FileView** (fájl nézet) A projekt fájljait mutatja, külön csoportosítva a forrásfájlokat, a fejlécfájlokat és az erőforrásfájlokat. Lehetővé teszi új fájl létrehozását, már létező fájlok beszúrását a projektbe. Az OOP gyakorlatokon gyakran használjuk.

A 6.0-nál régebbi verziók esetén még egy tagja volt, az InfoView, mely a Help Contents ablakának közvetlen elérését támogatta. Jelenleg a súgó önálló MSDN alkalmazásként fut, mely a Help menüpontból is indítható.

A projekt munkaterület fájljai a projekt alkönyvtárában generálódnak, de a projektbe beszerkeszthetünk másutt található fájlokat is. A Workspace leírását a .dsw (5.0-nál régebbi verziónál .mdp) kiterjesztésű fájl adja, melyet a projekt létrehozásakor generálunk, s ezt nyitjuk meg, ha dolgozni akarunk projektünkön. Az .opt kiterjesztésű fájl options beállításokat tartalmaz.

1.2.3. További ablakok a képernyőn

A képernyő közepén találhatóak az alkalmazás egyes elemeinek szerkesztését lehetővé tevő ablakok. Egyszerre több forrásfájlt nyithatunk meg, melyek ablakai más alkalmazásoknál megszokott módon kezelhetők. Az erőforrásfájlok megnyithatók az erőforrás-szerkesztőben, szerkeszthető nézetben és forráskód alakban is.

1.2.3.1. Az Output ablak

Alapértelmezésben a képernyő alján található. A **View** menü **Output** segítségével ki-be kapcsolható. A fordítás szerkesztés során itt olvashatjuk el, hol tart a fordító a feladat végrehajtása során. Ide írja ki a hibaüzeneteket és a figyelmeztető üzeneteket, és mi is írathatunk ki a kódunk végrehajtása során tesztadatokat ebbe az ablakba. (Lásd az 1.2.5. szakaszban!)

Az ablak alsó fülei segítségével válthatjuk a mutatott adatokat. Például ha egy definícióra rákerestünk a forrásfájlokban, a talált elemek (fölülírásukig) a **Find in Files** fülben később is elérhetőek.

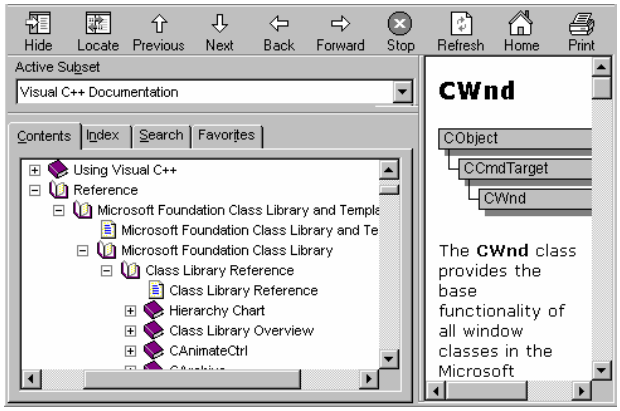
Az Output és Workspace ablakok alapértelmezésben az eszközsorokhoz hasonlóan dokkolt állapotúak, ami azt jelenti, hogy az eszközsorokhoz hasonlóan mozgathatók, és az ablak széléhez "ragaszthatók". Ez a tulajdonság (**Docking View**) a jobb egérgomb hatására legördülő menü segítségével kikapcsolható. Az ablak szélét lenyomva majd a Ctrl gombot választva a dokkolás megszűntethető. Ha az ablak mozgatása közben a Ctrl gombot lenyomva tartjuk sem érvényesül a dokkolás.

1.2.3.2. Eszköztárak

Az eszközsoron vagy a menüsoron jobb egérrel kattintva a legördülő menüből kiválasztható a megjeleníteni kívánt ablak (Workspace, Output) vagy eszközkészlet. Készíthetünk saját eszköztárat is.

1.2.4. A súgó használata

Futtathatjuk közvetlenül a Start / Programs / Microsoft Developer Network segítségével, vagy a Visual C++ Help menüpontjában a Contents, Search illetve Index almenüt választva, feltéve, hogy nincs bejelölve a 'Use Extension Help' – elindul önálló alkalmazásként az **MSDN**. (1.8. ábra)



1.8. ábra Az MSDN súgó

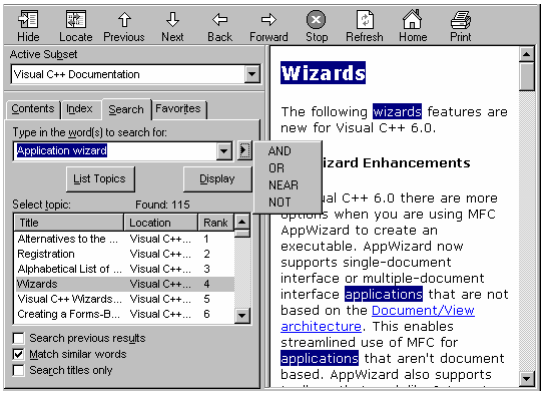
Az ábrán jól láthatóak a workspace fülek, amikkel eldönthetjük, a tartalomjegyzékben (**Contents**) kívánunk-e tájékozódni (a képen ez az aktuális), **Index** alapján keresünk, vagy a **Search** funkció segítségével. E lehetőségek más Windows alkalmazásoknál is hasonló módon működnek. A leírásokat könyvek tartalmazzák, melyeket az előttük elhelyezett '+' ikonra kattintva nyithatunk ki, és a '-' ikonra kattintva zárhatunk be. A jobb oldali ablak az aktuálisan kiválasztott témát mutatja.

Az **Active Subset** ablakban a legördülő listából választhatjuk ki, hogy az egész anyagban, vagy csak valamely fejezetében kívánjuk a keresést végrehajtani. Az ábrán a Visual C++ Dokumentation van beállítva. Létrehozhatunk saját szűrőt is a View menü Define Subset kiválasztásával.

Az **eszközsor** első ikonjai jelzik, hogy a workspace ablak nyitható és zárható. Ikonok segítik a szövegben az előre-hátra mozgást. A Previous / Next gombokkal a tartalomjegyzékben haladhatunk az előző / következő elemre, míg a Back / Forward segítségével saját választásunk előző / következő elemét kapjuk.

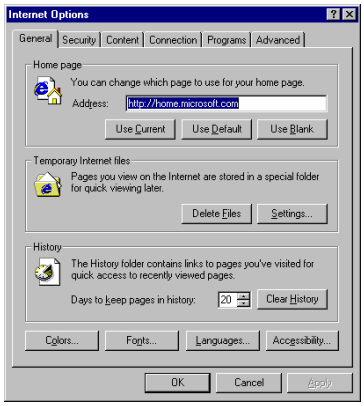
Az **Index** fül kulcsszavak listájában keres, mintha egy tárgymutatóban keresnénk, a **Search** megadja mindazokat az oldalakat, ahol a szűrőfeltételnek megfelelő anyagrészben a megadott szó szerepel. A keresett szót kijelöli a szövegrészekben, így könnyen megtalálható. A List Topics gomb hatására kezdődik a keresés, mely ha túl sokáig tart, Cancel gomb lenyomására leállítható.

Összetettebb keresésekhez az And, Or, Near és Not szavakat használhatjuk. (1.9. ábra)



1.9. ábra Az MSDN Search füle

Az Interneten is kereshetünk információt, ha van Internet csatlakozásunk. A View / Internet Options menüpont segítségével állíthatjuk be a paramétereket.

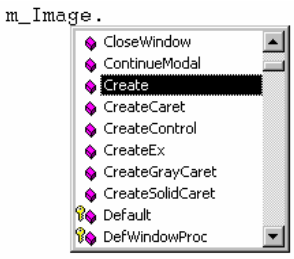


1.10. ábra Az MSDN Internet Options ablaka

A súgót hagyományos módon is elérhetjük. A keresett szövegre állítva a kurzort F1 gomb hatására az aktuális szót meghatározó súgó oldalt tölti be az alkalmazás. Ez a módszer jól használható a forrásfájl ablakban és a kimeneti ablakban a hibaüzenetekkel kapcsolatos súgó eléréséhez.

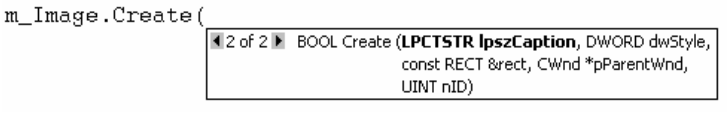
A súgó használatáról szól még a 4.2. szakasz Információk az MFC osztályokról címmel.

A kód írását könnyíti a következő funkció: ha egy objektum neve mögé a '.' taghivatkozás operátort tesszük (mutatónál a -> után), felkínálja egy listában a meghívható tagfüggvényeket, adattagokat. (1.11. ábra) A tagfüggvény kiválasztása után elegendő a következő karaktert leütni, '(' és a függvény neve a kódba kerül.



1.11. ábra A tagok listája

A függvények írásakor a '(' után a paraméterek típusát jelzi. Ha egy függvénynek több különböző paraméterlistával rendelkező alakja van, ezt egy kis szám mutatja, s a szám melletti nyíllal léphetünk az egyik változatról a másikra. (1.12. ábra)



1.12. ábra A függvények argumentum típusai

Ha a változó neve fölé visszük a kurzort, megkapjuk a változó típusát, s hogy melyik osztály adattagja.

1.2.5. Nyomkövetés

A kódolás idejének jelentős részét hibakeresésre fordítjuk. A tesztelés során megtaláljuk a hibát, de ki kell derítenünk a kód melyik sora okozza. Szintaktikai hibák esetén általában egyszerűbb a javítás, a fordító gyakran abban a sorban észleli a hibát, ahol a kódot módosítanunk kell. Időnként persze itt sem minden ilyen egyértelmű.

A szemantikai hibák felderítése általában nehezebb feladat. Ilyen esetekben a kód lefordul, de futáskor nem a várt végeredményt kapjuk. A szemantikai hibák felderítésében segítségünkre lehet a nyomkövetés, melynek segítségével lépésenként hajthatjuk végre programunkat, s közben ellenőrizhetjük a változók pillanatnyi értékét, láthatóságát, a programsorok hívási sorrendjét stb.

A hibakeresés folyamatának komoly szakirodalma van, a könyv e szakaszában csak a kötetben szereplő egyszerű feladatok tesztelése során feltétlen szükséges eszközöket mutatja be.

Ahhoz, hogy programunkat 'debuggolni' tudjuk:

Build

Set Active Configuration

Win32 Debug beállítás szükséges (ez az alapértelmezés).

Debug beállítás mellett a fordító a projekt alkönyvtárában létrehoz egy Debug alkönyvtárat, és abba menti a tárgykódokat (.obj) és az exe fájlt. Ezek a fájlok tartalmazzák a nyomkövetéshez szükséges információkat is, így az alkalmazás fejlesztése közben támogatják a tesztelést, viszont nagyobb méretűek. Ugyanitt **Win32 Release** választás esetén a projekt alkönyvtárban egy Release alkönyvtár jön létre, mely kisebb fájlokat tartalmaz, mivel nem támogatja a nyomkövetést. A fejlesztés során a Debug használata ajánlott, míg a telepítéshez a Release verziót alkalmazzuk. A két módot egymással párhuzamosan is használhatjuk, az aktuális alkönyvtár mindig az adott módon utoljára fordított állapotot tárolja. A tesztelés elősegítésére újabb konfigurációkat is készíthetünk, pl. a meglevők valamelyikének lemásolásával, majd a beállítások módosításával.

A nyomkövetést a **Build** menü **Start Debug** menüpontjának választásakor legördülő menüből, vagy gyorsgombok segítségével indíthatjuk. Nyomkövető üzemmódban megjelenik a nyomkövetés eszközsora (1.13. ábra), melynek jól érthető ikonjai a nyomkövetés következő lépésének kiválasztásában, és a megjelenített ablakok választásában segítenek. Ha az ikonok fölé visszük a kurzort, szöveggel is tájékozódhatunk az eszközgomb funkciójáról. A sárga nyíl például a végrehajtásra váró sort mutatja, ha nagyon elgörgetjük a forráskódot, erre az ikonra kattintva találhatjuk meg a forráskódban is így jelzett sort.

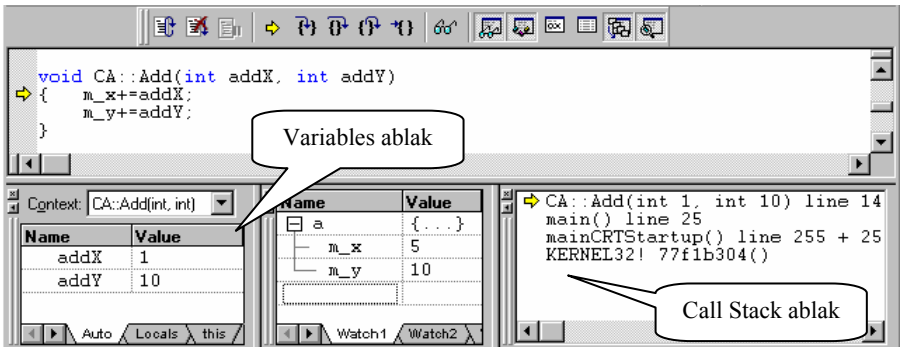
1.13. ábra A nyomkövetés eszközsora

Ha az alkalmazás futása során bizonyos helyeken szeretnénk megszakítani az alkalmazás végrehajtását, ezekre a helyekre töréspontot tehetünk. A programot az F5 gombbal nyomkövetés módban futtatva (Build / Start Debug / Go) az első töréspont elérésekor megszakad a futás, és innét a nyomkövetés lehetőségeit felhasználva vizsgálhatjuk az alkalmazás állapotát. Újabb F5 választásra a következő töréspont eléréséig fut a program.

Töréspontot (breakpoint) legegyszerűbben alapbeállítás esetén az F9 gyorsgombbal tehetünk ki vagy vehetünk el az aktuális sorból. Bővebb szolgáltatást kínál az Edit menü Breakpoints menüpontja. Érdemes megnézni még a kódban a jobb egérgomb hatására legördülő menü kínálta lehetőségeket!

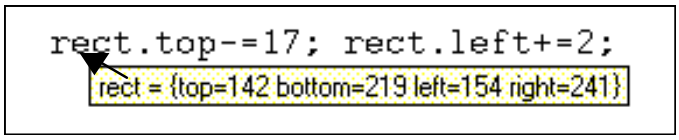
Az 1.13. ábra eszközsorának alján a megjeleníthető ablakok ikonjai láthatók, melyek az eszközgombok segítségével ki-be kapcsolható dokkolt ablakok. Közülük néhány fülekkel is rendelkezik, ezzel tovább bővítve a megjelenítés lehetőségét. (1.14. ábra) Például a különböző függvények változóinak vizsgálatára különböző Watch

ablakokat használhatunk. A Variables ablakban a jobb oldali egérgomb menüjében a változók értékének hexadecimális megjelenítése állítható be, vagy kapcsolható ki.



1.14. ábra A Debug üzemmód ablakai

Nyomkövetés üzemmódban az egyes változók értékeit láthatjuk, ha a kurzorunkat a változó fölé visszük. (1.15. ábra) Osztályok esetén azok címén kívül az ősosztályt is leolvashatjuk ily módon. A fejlesztett kódban megírt metódusok címére, a metódus deklarációjának osztályára kaphatunk információt segítségével. Jobb oldali egérgomb segítségével előhívható a QuickWatch ablak, mely lehetővé teszi a kijelölt kifejezések aktuális értékének vizsgálatát, de mi magunk is írhatunk be ide különböző kifejezéseket, meghívhatunk függvényeket. A változók, vagy kifejezések kijelölés után drag & drop technikával a Watch ablakba vonszolhatók.



1.15. ábra Nyomkövetésnél a változó értéke leolvasható

További lehetőséget kínál az MFC csak Debug módjában használható afxDump objektum, mely lehetővé teszi, hogy a futás során az Output ablakba (1.2.3.1. szakasz) küldjünk üzeneteket.

```
#ifdef _DEBUG
  afxDump << "A dokumentum címe: " << pDoc->GetTitle() << "\n";
#endif
```

Kényelmes lehetőséget kínál a dump szolgáltatások elérésére a TRACE makró. Paramétereként a printf-nek megfelelően adhatjuk át az argumentumokat. A TRACE makró Release módban nem csinál semmit, míg Debug módban az afxDump csatornára ír.

```
TRACE(" A dokumentum címe: %s", pDoc->GetTitle());
```

A nyomkövetés az objektumorientált programozás egyik kritikus pontja, melynek bonyolultságával fizetünk az osztálykönyvtárak felhasználása során a kódfejlesztés felgyorsulásáért. A készen kapott osztályok belsejét nem ismerve, a hibakeresés során, a függvények végrehajtásának vizsgálata közben minden lépés egy másik fájl másik sorára kalauzol bennünket. Javaslom, hogy a nyomkövetés elemeit az olvasó az objektumorientált programozás (OOP) kötet tanulmányozása során kezdje megismerni, mert ott még egyszerű, áttekinthető feladatokkal találkozik. A nyomkövetés a gyakorlatban sajátítható el igazán. A könyv gyakorlatok részében több helyütt részletes leírást talál az olvasó erre vonatkozóan.

 A Visual C++ környezet elemeinek kipróbálását az 1.1. gyakorlat segíti.

1.3. Kérdések

1. Mik az előnyei a Windows alkalmazásoknak a DOS programokhoz képest?
2. Ismertesse a Visual C++ fejlesztői környezet elemeit!
3. Milyen lehetőségeket biztosít az MSDN és a felhasználói környezet az információszerzésre?
4. Ismertesse a nyomkövetés során használható eszközöket!

1.4. Összefoglalás

- Visual C++-t **Windows alkalmazások fejlesztésére használják**. Ez egyben azt is jelenti, hogy **programjaink grafikus felületen futnak**. Alkalmazkodnak a Windows alkalmazások **egységes felület**ehez, tehát a felhasználó ismerős környezetben érzi magát már akkor, amikor alkalmazásunkat először futtatja. **Meghajtófüggetlen alkalmazások**at készíthetünk segítségével. Amikor az alkalmazásokat írjuk, nem kell odafigyelnünk, milyen típusú az egér, a monitor, a hangkártya, a nyomtató, a CD-meghajtó… Az operációs rendszer biztosítja a meghajtókra vonatkozó kódrészleteket is.
- A Visual C++ a **Visual Studio** szoftverfejlesztő eszközcsomag része, így további, a csomagban megtalálható eszközök támogatják az egyéni, illetve csoportmunkában történő fejlesztéseket. Ezek egy részét e könyvben is használjuk. (pl. Visual Modeler)
- A Visual C++-t **párbeszédes felületek segítségével telepíthetjük**. Nagyméretű, részletes **súgó** áll rendelkezésünkre, amiben tartalom, tárgymutató és kereső segítségével is információhoz juthatunk. A kód írása során és a nyomkövetőben is **helyzetérzékeny help** segíti munkánkat.

2. Üzenetkezelés

Eseményvezérelt programozásban (**event-driven programming**) MFC használata esetén a programozó feladata "mindössze" az eseményekre válaszoló kódrészletek elkészítése. Mi történjen, ha a Hello alkalmazás (1.1. gyakorlat) Köszön gombjára kattint a felhasználó? Amint azt a 'Min' alkalmazásunk (2.1. gyakorlat) kódolása során tapasztalhatjuk Win32 alkalmazás készítésekor, ha az MFC-t használjuk, már nem a programozó írja a főprogramot. Tehát az MFC nemcsak az osztályokat biztosítja, hanem egy teljes framework-öt, mely ott dolgozik a háttérben (4.1.3. elmélet), és hívogatja az MFC osztályok virtuális metódusait (OOP 5.3.2. szakasz). Emellett az operációs rendszer biztosítja az üzenetkezelő mechanizmust, a programozónak "csak" a kezelő metódusokat kell elkészítenie.

E fejezet megismertet bennünket az esemény, üzenet, üzenetkezelő metódus fogalmával. Az üzenetek fajtáival, feldolgozásuk menetével.

2.1. A Visual C++ alkalmazás szerkezete

Az **alkalmazásvarázsló** elkészíti számunkra az alkalmazás csontvázát (**skeleton files**), mely tartalmazza a főbb szerkezeti elemeket.

- Beszerkeszt a kódba néhány **afx** (Active Framework Extension) fejlécfájlt. Az 'stdafx.h' (standardafx) tartalmazza az 'afxwin.h'-t, mely a legalapvetőbb MFC header, a főbb MFC osztályok deklarációit tartalmazza.

- Deklarál egy CWinApp-ből származó CMyApp alkalmazásosztályt.
- A CMyApp alkalmazásosztályhoz tartozó MyApp.cpp fájlban a következőket találjuk:
  - Az alkalmazás üzenethurkának üzenettérképét (**MESSAGE_MAP**).
  - A **theApp** objektum definícióját, mely az alkalmazásosztály példánya.
  - Az **InitInstance** standard megvalósítását és az üres konstruktort.

Megállapodás:

Az osztálynevek a C (class) kezdőbetűvel indulnak, ezután következik az adott alkalmazásra utaló rész (ezt e könyv a továbbiakban a My szóval helyettesíti), majd az ősosztályra, vagy inkább az osztály jellegére utaló szakasz következik. Pl. CMyApp; ez a My alkalmazás Application osztálya, a CHelloDlg; a Hello alkalmazás Dialog ablak osztálya.

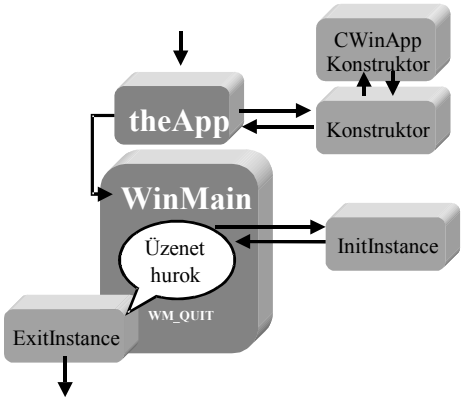
A CMyApp osztály a **CWinApp** osztályból származik. A CMyApp **theApp** objektum globális, így konstruktora a többi C++ globális objektum konstruktorával együtt még a WinMain blokkjának végrehajtása előtt meghívódik, tehát az alkalmazásobjektum a WinMain-ből már elérhető. (2.1. ábra)

Az alkalmazásobjektum, a theApp feladata:

- Az alkalmazás inicializálása.
- Document/View alkalmazásoknál a dokumentum sablon szerkezet felépítése.
- Az fő szálhoz tartozó üzenetkezelő ciklus biztosítása.

Egy alkalmazás csak egy CWinApp-ból származó objektumot tartalmazhat!

Mint minden Windows alkalmazás a mi alkalmazásunk is tartalmaz egy **WinMain** függvényt. Nem kell megírnunk, az osztálykönyvtár (MFC) biztosítja az afxwin fejlécfájlon keresztül (#include <afxwin.h>). Meghívódik, amikor elindul az alkalmazás. Feladata például az ablakok regisztrációja, meghívja az alkalmazás objektum InitInstance és Run metódusait. (2.1. ábra)



2.1. ábra A WinMain szerepe

A CWinApp osztály legfőbb függvényei a következők:

- **InitInstance**

  Az alkalmazás inicializálása két részben történik, az első futásakor on-time inicializáció az **InitApplication** metódus hívásával, majd minden új példány létrejöttekor (az elsőnél is) InitInstance.

  Az InitInstance feladata létrehozni és beállítani a főablakot. A CWinApp osztályból örökölt **m_pMainWnd** adattag mutat a főablakra.

  Ha visszatérési értéke FALSE, az alkalmazás futása befejeződik, csak TRUE esetén hívódik meg a Run metódus.

- **Run**

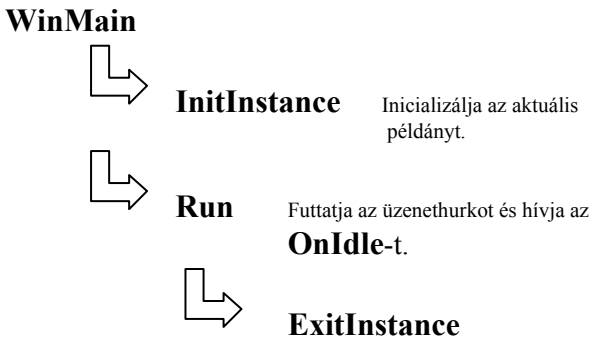
  Gondoskodik az alkalmazás üzenetfeldolgozásáról az üzenetkezelő ciklus segítségével. Ha nincs feldolgozásra váró üzenet, akkor a pihenőidős feldolgozások következnek az **OnIdle** metódusban kódolva. A processzorok nagyságrendekkel gyorsabbak, mint az ember, így a processzor számára rengeteg pihenőidő (idle time) marad. A Windows készítői ezt az időt bizonyos tevékenységek elvégzésére használják, pl. az ablakok újrafestésére, ha szükséges. (Nem a legszerencsésebb a pihenőidő elnevezés, hisz a program ilyenkor gyakran nem pihen, hanem a 'nem túl fontos feladatait' végzi el. Csak ha nincs ilyen feladat, akkor pihen. De valójában mi magunk is gyakran dolgozunk a pihenőidőnkben!)

  Ha az üzenetsorba **WM_QUIT** üzenet érkezik, meghívja az ExitInstance metódust. Az üzenetkezelő ciklus működését a 2.3. ábra mutatja.

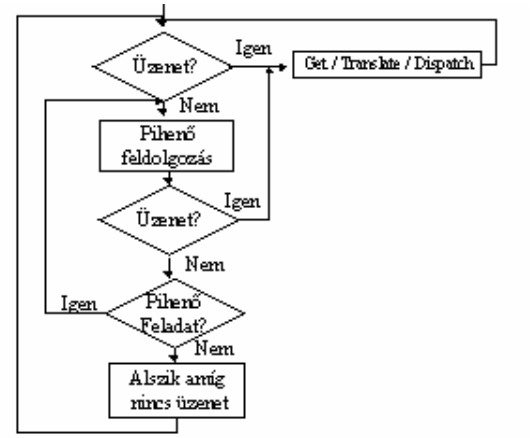
- **ExitInstance**

  Feladata az alkalmazáspéldány befejező tevékenységeinek elvégzése.

A végrehajtás szekvenciája az 2.2. ábrán követhető végig.



2.2. ábra A végrehajtás sorrendje



2.3. ábra Az alkalmazás üzenetkezelő ciklusa (message loop)

(MSDN OnIdle Member Function)

Az üzenetek feldolgozása mindaddig tart a **GetMessage, TranslateMessage**, **DispatchMessage** hívásokkal, amíg van üzenet az üzenetsorban. Ha nincs elkezdődhet a pihenőidős feldolgozás az **OnIdle** hívásával. Minden pihenőidős feladat végrehajtása után ránéz az üzenetsorra, van-e feldolgozandó üzenet, s ha igen feldolgozza, ha nincs, folytatódik az **idle time** feldolgozás. Ha a pihenőidős feladatok is elfogytak, akkor befejeződik a pihenőidős feldolgozás.

Az alkalmazás szerkezetét az 2.1. gyakorlat segíti megismerni.

2.2. Esemény, üzenet, kezelő metódus

Az **esemény** (**event**) azonosítható pillanatszerű történés. (Kondorosi, 1998, 60. old.) Egy eseményt egy objektum észlel, s erről üzenetküldéssel értesít egy másik objektumot. Az üzenet érkezése önmaga is egy esemény. Ha egy objektum képes fogadni egy adott üzenetet, akkor erre az üzenet neve által meghatározott metódus végrehajtásával reagál. Az üzenet megmondja, hogy mit kell csinálni, a metódus pedig, hogy hogyan.

Az **üzenet (message)** egy struktúra, mely egy azonosítót és további adatokat tartalmaz. Az **MSG** struktúra leírása:

```
typedef struct tagMSG {     // msg
    HWND   hwnd;     // Az ablak aminek az üzenet érkezett.
    UINT   message;  // Az üzenet azonosító száma.
    WPARAM wParam;   // További információk, üzenetfüggő.
    LPARAM lParam;   // További információk, üzenetfüggő.
    DWORD  time;     // Az üzenet elküldéséenk ideje.
    POINT  pt;       // A kurzor pozíciója
// képernyőkoordinátákban az üzenet elküldésének pillanatában.
} MSG;
```

Például: bal egérgomb lenyomása esetén az üzenet:

```
message = WM_LBUTTONDOWN
fwKeys = wParam;        // billentyű flag-ek
xPos = LOWORD(lParam);  // a kurzor x pozíciója
yPos = HIWORD(lParam);  // a kurzor y pozíciója
```

Ha szeretnénk tudni az azonosító mögötti UINT számot is, rá kell keresni (Find in Files) a forráskódban a #define WM_LBUTTONDOWN -ra és a 0x0201 számot találjuk mögötte.

Ha az egér nincs elkapva (captured), az üzenetet az az ablak kezeli, mely fölött az egér tartózkodik.

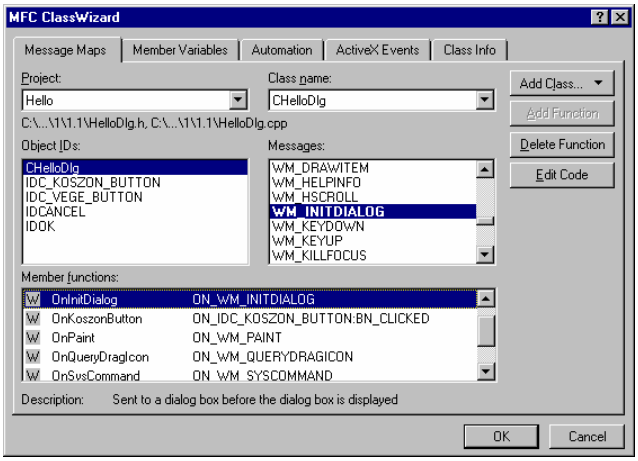
```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point );
```

Nézzünk néhány példát az esemény -> üzenet ->metódus kapcsolatra!

- Ablak
  - Esemény: Lenyomtuk a bal oldali egérgombot.
  - Üzenet: WM_LBUTTONDOWN
  - Metódus: OnLButtonDown

- Párbeszédablak
  - Esemény: Lehúztak róla egy másik ablakot
  - Üzenet: WM_PAINT
  - Metódus: OnPaint
- A File menü New menüpontja:
  - Esemény: Egérrel a menüpontra kattintunk
  - Üzenet: WM_COMMAND
  - Metódus: OnFileNew

Az üzenetekhez metódust az osztályvarázsló segítségével csatolhatunk. (2.4. ábra) A Message Maps fül választása után a Class name mezőben legyen a kívánt osztály az aktuális! Ekkor magához az osztályhoz, illetve a rajta elhelyezett vezérlőkhöz csatolhatunk kezelőket (Object IDs). A Messages ablakban **kiválasztjuk a kívánt üzenetet**, s az Add Function gomb segítségével **hozzárendeljük a metódust**. Az osztályvarázsló felkínálja a függvény nevét, de ha akarjuk, módosíthatjuk a nevet. A Member functions ablakban látni lehet az üzeneteket és az osztályvarázsló segítségével hozzájuk csatolt tagfüggvényeket. Az osztályvarázsló segítsége nélkül is kapcsolhatunk eseményhez kezelőmetódust, erre a 3.1. gyakorlat feladatában látunk példát, amikor több vezérlőt kezelünk ugyanazzal a metódussal. (Több vezérlő együttes kezelése.)



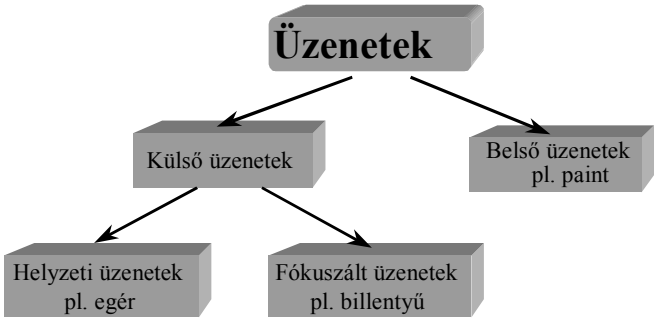
2.4. ábra Üzenethez metódus rendelése osztályvarázslóval

Egy esemény hatására tehát üzenet(ek) kerül(nek) valamelyik üzenetsorba. Az üzenetek információkat tartalmaznak az eseményről, pl. a típusa, az időpont amikor bekövetkezett, a hely ahol bekövetkezett. A mi feladatunk olyan kódot írni, mely képes értelmezni az üzeneteket, és megfelelő módon válaszolni rájuk. De hogyan jut el az üzenet a kezelőig?

2.3. Az események feldolgozása

A Windows alkalmazások **eseményvezérelt**ek (**event-driven**). Nem függvényhívások egymásutánjával működnek, hanem arra várnak, hogy a Windows üzeneteket küld nekik. Minden ablaknak van egy ablakkezelő függvénye (window procedure) amely gondoskodik az ablaknak érkező üzenetek végrehajtásáról.

2.3.1. Üzenetek a szálak üzenetsorában

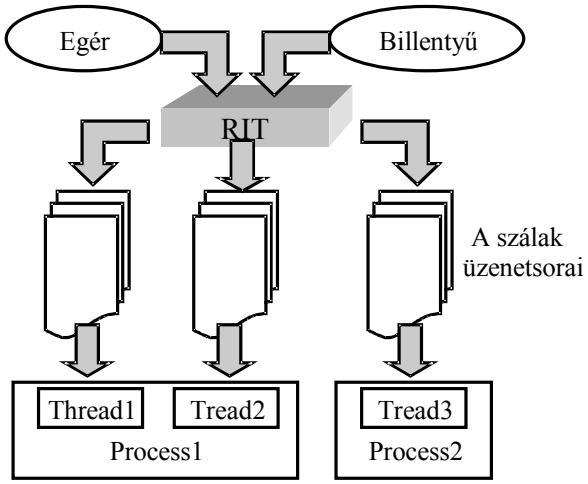


2.5. ábra Üzenetek csoportosítása

A belső üzenetek a futó alkalmazások által küldött, az operációs rendszer által küldött… üzenetek. Külső üzenetet küldhet a felhasználó a billentyűzetről vagy egér segítségével. A külső üzeneteket két csoportra oszthatjuk (2.5. ábra). A helyzeti üzenetek annak az ablaknak érkeznek, amely fölött az esemény bekövetkezett, például az egér mozgatása. A fókuszált üzenetek annak az ablaknak érkeznek, amely a fókuszban van, például ha leütünk egy 'A' gombot a billentyűzeten, az a fókuszban levő ablakban fog megjelenni.

A belső üzenetek közvetlenül a szálaknak érkeznek, míg a külső üzeneteket a **RIT** (Row Input Thread) osztja el a szálak között. A súgó egyes helyein, vagy más szakirodalomban a RIT System Queue néven szerepel. A RIT nem tartalmaz ablaküzeneteket, csak az interrupt időben bekövetkező hardver üzeneteket. Az összes futó alkalmazás az egyetlen RIT-ből kapja a rendszerüzeneteket.

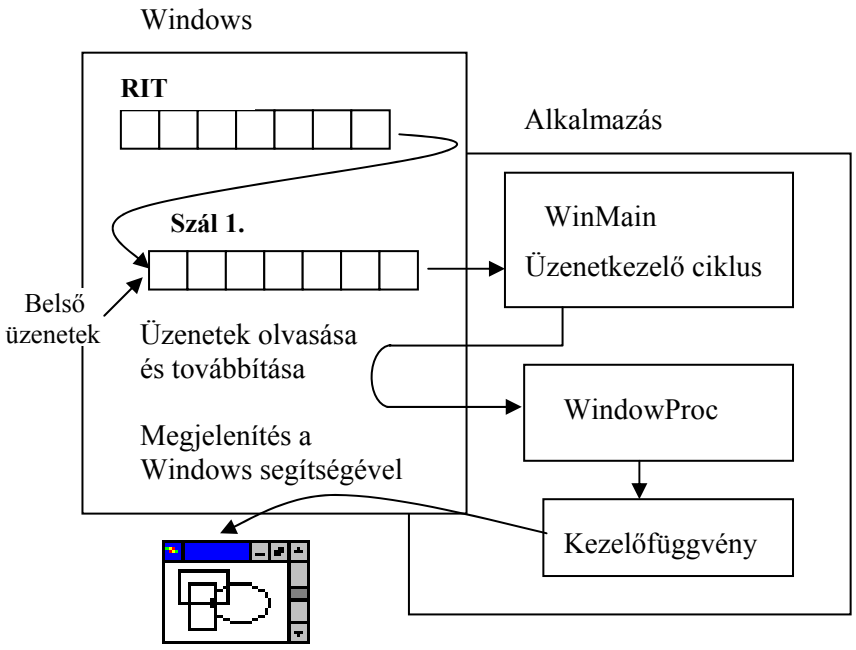
Process(folyamat) pl. egy végrehajtható alkalmazás. A processnek saját virtuális címe, kódja, adatai, operációs rendszer erőforrásai vannak. Egy vagy több szálból (**thread**) áll, mely a processhez tartozóan fut. Az operációs rendszer 32 bites Windows-ban a szálak közt osztja meg a CPU időt. Minden szálnak önálló üzenetsora van, mely tartalmazza az üzenetfeldolgozó ciklus (message loop, lásd 2.6. ábra) számára várakozó üzeneteket. A RIT egy speciális beviteli szál. Ha valamelyik szálon az üzenet végrehajtása hibás, a RIT továbbra is küldi az üzeneteket. A többi szál fel is dolgozza azokat, így a felhasználót más alkalmazások futtatásában nem zavarja az egyik szál "lefagyása". (Ilyenkor szoktunk szűnni nem akaró homokóra kurzort látni a szál ablaka fölött.)



2.6. ábra A RIT

2.3.2. Az üzenetek feldolgozása

A külső üzeneteket a Windows megvizsgálja, s az üzenetről eldönti, mely szálnak és milyen formában kell továbbítani. Az 'Alt +Tab' billentyűkombináció valószínűleg több alkalmazás működését is befolyásolja, míg egy egyszerű billentyűleütés csak az aktív ablak üzenetsorának szól. A szál üzenetsorának feldolgozásáért az alkalmazás szálának üzenetkezelő ciklusa felel.



2.7. ábra A Windows üzenetfeldolgozása

Az üzenetkezelő ciklus (message loop) a Windows segítségével **az ablakkezelő függvénynek (WindowProc) továbbítja az üzenetet**, mely megkeresi az üzenethez tartozó kezelőmetódust. (Lásd bővebben 2.6. szakaszban!) A kezelőfüggvény a Windows segítségével jeleníti meg a képernyőn az eredményt.

2.3.3. Az üzenetkezelő ciklus függvényei

A ***GetMessage()*** (vagy ***PeekMassage()***) függvény veszi ki a hWnd paraméterében (4.8.3. szakasz) megadott ablakhoz, vagy gyermekablakaihoz tartozó üzeneteket az üzenetsorból. Ha a hWnd paraméter NULL, akkor minden – a hívó szálhoz tartozó, (a további paraméterekben megadható szűrőfeltételeknek megfelelő) – üzenetet visszaad. Más szálakhoz tartozó ablakok üzeneteit nem kapjuk meg a segítségével. A GetMessage **WM_QUIT** üzenet esetén FALSE, egyébként TRUE értékkel tér vissza.

A ***TranslateMessage()*** feladata az üzenetek átalakítása, például a virtuális billentyűüzeneteket alakítja át karakteres üzenetté. (Lásd 2.7.1. szakasz!)

A ***DispatchMessage()*** feladata az üzenetek továbbítása az ablakkezelő függvénynek (WindowProc()). Időzítő esetén, ha megadtuk az időzítő függvény címét, a függvény a **WM_TIMER** üzeneteket közvetlenül az időzítő függvénynek továbbítja.

Az üzenetkezelő ciklus (message loop) az üzenetstruktúrát az ablakkezelő függvénynek továbbítja. ("Az ablakkezelő függvényt a Windows hívja (callback) és az üzenet feldolgozása után visszatér a vezérlés a Windowshoz." (Benkő, 1995, 33. old.)) A szinkron üzeneteket közvetlenül a ***CWnd::WindowProc()*** kapja (2.4. szakasz). E függvényt az MFC biztosítja számunkra. Az ablakkezelő függvény az üzeneteket a message map segítségével továbbítja a kezelőfüggvényeknek, melyek többnyire On*Message* nevűek. Keressük meg őket a súgóban! Minden ablaküzenetnek van egy alapértelmezett kezelője: a ***CWnd::DefWindowProc().*** Ha nem rendelünk kezelőfüggvényt az üzenethez, az alapértelmezett kezelő hajtódik végre.

2.4. Aszinkron – szinkron üzenetek

Aszinkron üzenetkezelés

Az aszinkron üzenetkapcsolás olyan, mint egy levél bedobása egy postaládába. Ha üzenetet küldünk, a vezérlés visszatér anélkül, hogy megvárná az üzenet végrehajtását. Tehát ha billentyűzet- vagy egéresemény történik, azonnal reagál rá a rendszer.

Aszinkron üzenetet a ***PostMessage()*** függvénnyel küldhetünk. Az üzenet a szál üzenetsorába kerül, és várakozik a feldolgozásra. A küldő szál tovább végzi a saját üzenetsorának feldolgozását.

Szinkron üzenetkezelés

A szinkron üzenet olyan, mint amikor valakit telefonon hívunk, és nem csinálunk semmi mást a beszélgetés befejezéséig. A szinkron üzenetek nem kerülnek az üzenetsorba, azonnal megkezdődik feldolgozásuk. Az üzenetet küldő szálnak várnia kell az üzenetfeldolgozás befejezéséig, hogy folytathassa saját üzeneteinek feldolgozását.

- Ha a szál saját magának küldi, akkor ez nem más, mint egy hagyományos függvényhívás.
- Ha a szál egy másik szálnak küldi, a küldő szál blokkolva lesz a feldolgozás végéig.

Szinkron üzenetet a ***SendMessage()*** függvény segítségével küldhetünk.

Hang csatolásakor kipróbálhatjuk a szinkron és aszinkron üzenetküldés közti különbséget. Hang csatolásához szükségünk lesz a winmm.lib library függvényeire.

Ehhez:

Project menü

Settings

Link fül

Object / library modules szövegdobozba: **winmm.lib**

A fájlba, ahol meg akarjuk hívni, be kell szerkesztenünk az mmsystem.h fejlécfájlt.

```
#include "mmsystems.h"
```

A hangot az ***sndPlaySound(***.wav file neve, mód***)*** függvénnyel hívhatjuk meg. Az első paraméter NULL esetén leállítja az éppen lejátszott zenét. Ne feledkezzünk meg az útvonal megadásánál a dupla \\ jelekről! (Sztringben a \ a speciális karakterek előtt áll.) A második paraméterben adhatjuk meg a hang lejátszásának módját. SND_ASYNC aszinkron lejátszást, SND_SYNC szinkron lejátszást jelent.

2.5. Az üzenettérkép (MESSAGE_MAP)

Mi történik a kódban, amikor üzenethez metódust csatolunk?

Megállapodás:

Az üzenetek azonosítóit csupa nagybetűvel írjuk. A WM_ az üzenetek elején a Windows Message rövidítése. Ez alól a WM_COMMAND a kivétel. (Részletek a 2.6.3. szakaszban.) A vezérlést bejelentők között BN_ a Button Notification (gomb bejelentő), az EN_ a Edit box Notification (szerkesztőmező bejelentő), LBN_ ListBox Notification (listadoboz bejelentő) és CBN_ ComboBox Notification (kombipanel bejelentő).

A bal egérgomb lenyomásának hatására létrejön a **WM_LBUTTONDOWN** üzenet. Az üzenethez kezelőt csatoló makró neve az ON_ és az üzenet nevéből tevődik össze. A makrókról részletesebben a következő (2.6.) szakaszban olvashatunk. Ebben az esetben ez **ON_WM_LBUTTONDOWN()** és a csatolt tagfüggvény neve: ***OnLButtonDown(UINT nFlags, CPoint point)***. A függvény paraméterei tájékoztatnak bennünket arról, hogy az egérgomb lenyomásának pillanatában milyen vezérlőbillentyűk voltak leütve (nFlags paraméter) és hogy az ablak mely pontján történt a gomb megnyomása (point paraméter). Természetesen ezeket az adatokat az üzenet struktúrából kapjuk kényelmesen kezelhető formában. A függvények többnyire tartalmazzák az ősosztályban elérhető azonos nevű üzenetkezelő metódus hívását. (A 5.2. gyakorlaton találkozhatunk az OnLButtonDown függvény használatával.)

Minden ***CCmdTarget***-ből származó osztály, tehát a CWnd utódai is (bővebben lásd a 4. fejezetben) tartalmaznak üzenettérképet. **A MESSAGE_MAP kapcsolja hozzá az üzeneteket az üzenetkezelő függvényekhez.**

A MESSAGE_MAP deklarációt az osztályvarázsló (alkalmazásvarázsló) hozza létre az osztállyal együtt. Ha kézzel írjuk meg a kódot, nekünk kell a fejlécfájlban az osztálydeklaráció végére írni **DECLARE_MESSAGE_MAP()**; Ez a makró az <afxwin.h> fejlécfájlban található. Deklarálja a tömböt, mely tartalmazza a message

map kezdetét, s néhány mutatót az ősosztály message map eléréséhez. Keressük meg a DECLARE_MESSAGE_MAP leírását az ...\ MFC \ Include \ afxwin.h fejlécfájlban!:

```
#define DECLARE_MESSAGE_MAP() \
private: \
 static const AFX_MSGMAP_ENTRY _messageEntries[]; \
```

Itt megtaláljuk azt a statikus (OOP 1.12) (_messageEntries[]) tömböt, mely biztosítja, hogy az osztály számára egyetlen helyen tároljuk az üzenetkezelő függvényeket, és ne külön-külön minden egyes objektumban. A tömb indexei az üzenetek és értékei a kezelőmetódusokra mutató pointerek.

Megjegyzés:

Az üzenetkezelők virtuális függvények is lehetnének, de akkor az osztályok VMT-je óriási méretű lenne, hisz az összes üzenetkezelő függvény címét tartalmazná. Ez a statikus tömb lehetővé teszi, hogy osztályonként egyszer, és csak az adott osztályban saját kezelőmetódussal rendelkező üzenetek kezelőjének címét tároljuk. A módszer további előnye, hogy a programozó a saját üzeneteit (2.6.5, 2.6.6. szakasz) a Windows üzenetekkel azonos módon tudja kezelni.

Az üzenettérkép az osztály implementációs fájljában (.cpp) található.

BEGIN_MESSAGE_MAP(osztálynév, ősosztálynév)

//{{AFX_MSG_MAP(osztálynév)

Az osztályvarázsló e megjegyzéssorok közé írja a csatoló függvényeket.

//}}AFX_MSG_MAP

Mi csak ide írhatunk kézzel csatoló függvényt, ha az osztályvarázslót továbbra is használni akarjuk.

END_MESSAGE_MAP

Megjegyzés:

A //{{ és //}}megjegyzések közötti sorok az osztályvarázsló számára szolgáltatnak információt. Sem a megjegyzéseket, sem a köztük található sorokat kézzel nem törölhetjük, ha az osztályvarázsló szolgáltatásait továbbra is igénybe kívánjuk venni. E jelek közé csak olyan sorokat írhatunk, amilyeneket maga az osztályvarázsló is elhelyezne ide.

Az osztályvarázsló e megjegyzéssorok segítségével készíti el felületén a párbeszédablakok információit. Ha letöröljük véletlenül vagy szándékosan az osztályvarázsló fájlját (.clw), e megjegyzéssorok segítségével néhány kérdés feltevése után az osztályvarázsló felépíti önmagát.

Itt következik néhány sor a Hello alkalmazás (1.1. gyakorlat) üzenettérképéből:

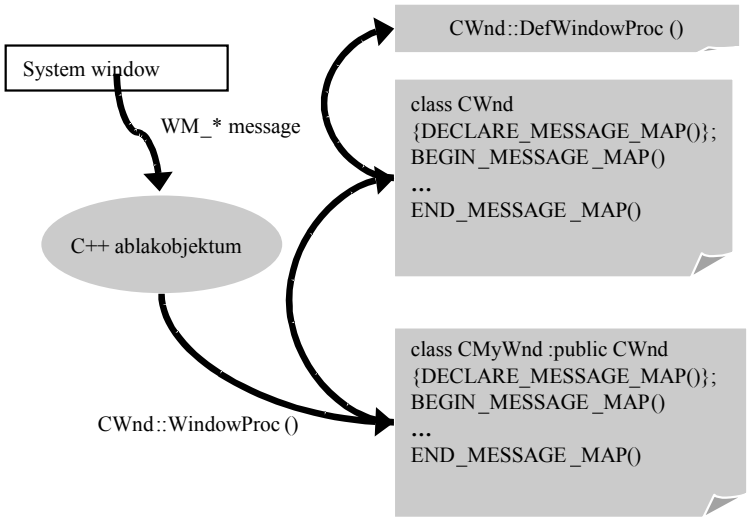
```
BEGIN_MESSAGE_MAP(CHelloDlg, CDialog)
  //{{AFX_MSG_MAP(CHelloDlg)
  ON_WM_PAINT()
  ON_BN_CLICKED(IDC_KOSZON_BUTTON, OnKoszonButton)
  ON_BN_CLICKED(IDC_VEGE_BUTTON, OnVegeButton)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

2.6. Üzenet kategóriák kezelése

2.6.1. Ablaküzenetek kezelése

(Windows messages)

Az ablaküzenetek WM_-mal kezdődnek. Pl. WM_LBUTTONDOWN, WM_PAINT... (Kivétel a WM_COMMAND.) Ablakok kezelik őket, a CWnd és utódosztályai. Az ablaküzenetek kezelését a 2.8. ábra mutatja.



2.8. ábra Az ablaküzenetek kezelése

Az ábra mutatja, hogy a WindowProc() az üzenetkezelő függvényt először az ablakosztály üzenettérképén, majd az ősosztályok message map-jében keresi. Ha nem talál az üzenethez hozzárendelt függvényt, az alapértelmezett kezelő hajtódik végre.

Az ablaküzeneteket előredefiniált üzenettérkép makrók (predefined message map macros) csatolják a kezelőmetódusokhoz (afxmsg.h). Például a WM_PAINT üzenethez az **ON_WM_**PAINT() makró csatolja az OnPaint() kezelőfüggvényt. A többi makró is hasonló módon lett megvalósítva. Keressük meg őket a súgóban!

2.6.2. Vezérlést bejelentő üzenetek kezelése

(Control Notifications)

Minden ablakhoz rendelhetünk egy ablakosztályból egy objektumot, és az objektum osztálya kezeli az adott ablaknak küldött üzeneteket. Mivel minden gyermekablak újabb objektumot, és többnyire újabb osztályt jelentene, így rengeteg kis osztályban kellene megírnunk az egyes üzenetek kezelőit. Ahány nyomógomb van az alkalmazásban, annyi új osztály kellene, hisz mindegyik másképp reagál, ha az egérrel kattintunk rajta. Ez áttekinthetetlenné tenné a programot. Ezért a Windows megvalósított egy vezérlést bejelentő mechanizmust (notification) amelyben a gyermekablakok továbbküldik az üzeneteiket a szülőablaknak. Nekünk tehát elegendő a szülőablak osztályában kezelőt írni az adott vezérlő kívánt üzenetéhez.

Mivel a vezérlők gyermekablakok, az MFC az áttekinthetőség és az egyszerűség kedvéért a vezérlőknek küldhető üzenetek jelentős részét a vezérlőt tartalmazó párbeszédablakban (CDialog, CForm) kezeli le. **A vezérlést bejelentők** (control notifications) **olyan WM_COMMAND üzenetek, melyeket a gyermekablakok küldenek szülőablakuknak**. A vezérlőnek, mint ablaknak érkező üzenetet egy olyan WM_COMMAND továbbítja szülőablakához, mely tartalmazza a vezérlést bejelentő kódot. Például, ha egy szerkesztőmező tartalma megváltozott, a szerkesztőmező küld egy **EN_CHANGE** kódot tartalmazó WM_COMMAND üzenetet a szülőablakának, többnyire egy párbeszédablaknak (Dilaogbox). A framework biztosítja a vezérlést bejelentők kezelését is (2.12. ábra).

Az üzenetküldés során a WM_COMMAND üzenet **wParam** paramétere tartalmazza a vezérlést bejelentő kódot, így az üzenet további paramétereinek átadására már csak az **lParam** marad. Ez nem mindig bizonyult elegendőnek. Eleinte ilyen esetekben külön üzenetet kellett készíteni (pl. **WM_VSCROLL**). A Win32 API-ban az egyre több és összetettebb vezérlő megjelenésével ezeket az üzeneteket a **WM_NOTIFY** üzenetek továbbítják.

Az ablaküzenetek mellett a vezérlést bejelentő üzeneteket is ablakok kezelik.

Az üzenettérkép az **ON_CONTROL()** makró segítségével csatolja a vezérlést bejelentő üzenetekhez a kezelőt.

```
ON_CONTROL( BN_CLICKED, IDC_MY_BUTTON, OnMyButtonClicked)
```

A makró helyettesíthető az

```
ON_BN_CLICKED(IDC_MY_BUTTON, OnMyButtonClicked)
```

hívással. Az osztályvarázsló ez utóbbit használja. Mindkét makró a void visszatérési típusú, paraméter nélküli kezelőmetódust csatolja:

```
afx_msg void OnMyButtonClicked();
```

Nem minden a vezérlőnek küldhető üzenethez tartozik vezérlést bejelentő, így a szülőablakban a gyermekablakok üzeneteinek egy része dolgozható csak fel. Pl. nyomógomb esetén, bár a gomb egy ablakosztály mégis csak két üzenetet kínál fel az osztályvarázsló a szülőablakban kezelésre. A további üzenetekhez egy a vezérlőhöz tartozó osztályból származó utódosztályban rendelhetünk kezelőt, ahogy ezt a notification nélkül tennénk. (Lásd 4.2.3. feladat!)

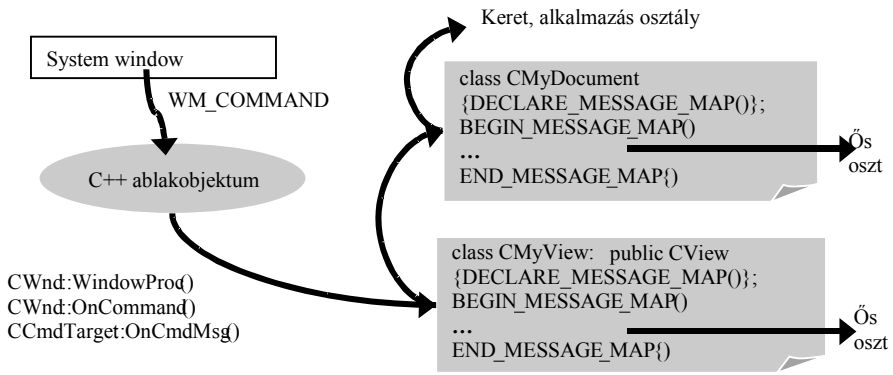
2.6.3. Parancsüzenetek kezelése

(Command messages)

A parancsüzenetek a felhasználói interfész WM_COMMAND kezdetű üzenetei, de míg a vezérlést bejelentő üzenetek vezérlőktől származnak, a parancsüzenetek **menükből, eszközsorokból, gyorsgombokról érkezhetnek.** Több objektum kezelheti őket, nem csak az ablakok. Például a File menü Open üzenet kezelője a dokumentumosztály, vagy az alkalmazásosztály tagfüggvénye lehet. A kezelés sorrendje:

- Az éppen aktív gyermek parancskezelő objektum.
- A parancsüzenetet kapó objektum.
- Más parancsüzenetet fogadó objektumok.

További információkat a súgó Visual C++ Programmer's Guide Command Routing (Search Command Routing) fejezetében találunk. A parancsüzenetek kezelésének egyik lehetőségét a 2.9. ábra mutatja.



2.9. ábra A parancsüzenetek kezelése

A parancsüzenetekhez az **ON_COMMAND()** makróval csatolhatunk kezelőt az üzenettérképben.

```
ON_COMMAND( ID_FILE_NEW, CWinApp::OnFileNew)
```

A kezelőmetódus paraméter nélküli és void visszatérési típusú.

A parancsüzenetek kezelésével bővebben a 6.3. fejezetben foglalkozunk.

A 6.6.6. feladat teszteli, milyen sorrendben történik a kezelő függvény keresése az osztályok között.

2.6.4. Üzenetek csoportos kezelése

Lehetőségünk van parancsok és vezérlők együttes kezelésére is. Ez azt jelenti, hogy parancsok vagy vezérlők egy csoportjához ugyanazt a kezelőfüggvényt rendeljük. A csatolást megvalósító makrókat nem tehetjük ki az osztályvarázsló segítségével, nekünk kell kézzel az üzenettérképbe írni. A rendelkezésünkre álló makrók a következők:

```
ON_COMMAND_RANGE(ID_MY_FIRST_COMMAND, ID_MY_LAST_COMMAND,
                                        MyCommandHandler)
ON_UPDATE_COMMAND_UI_RANGE(ID_MY_FIRST_COMMAND,
                           ID_MY_LAST_COMMAND, MyCommandHandler)
ON_CONTROL_RANGE(BN_CLICKED, IDC_FIRST_BUTTON, IDC_LAST_BUTTON,
                                          MyButtonHandler)
```

A kezelőfüggvények a következő formátumúak:

```
afx_msg void MyCommandHandler(UINT nID)
```

! A megvalósítás során figyeljünk rá, hogy a csoportosan kezelt azonosítók számai egymás után következzenek!

Parancsüzenetek kezelésével a 6.3. szakaszban foglalkozunk.

 Vezérlők csoportos kezelésével a 3.1. gyakorlatban találkozunk.

2.6.5. A fejlesztő által definiált üzenetek kezelése

Saját üzeneteket is kezelhetünk az **ON_MESSAGE()** makró segítségével. A makrónak át kell adnunk az üzenet azonosítóját és a kezelőfüggvény nevét.

```
ON_MESSAGE(WM_USER+1, OnMyFirstMessage)
```

A **WM_USER** azonosító a fejlesztők által definiált saját üzenetek számára fenntartott címek kezdetére mutat (0x0400). Az üzenetek a 0x7FFF értékig címezhetők. Ezeket az üzeneteket ne küldjük más alkalmazásoknak!

A kezelő metódust a következő formában kell deklarálni!

```
afx_msg LRESULT OnMyFirstMessage( WPARAM wParam, LPARAM lParam);
```

2.6.6. Regisztrált üzenetek

Ha az üzenetet alkalmazások közötti kommunikációra használjuk, tehát több alkalmazás használja ugyanazt a fejlesztő által létrehozott üzenetet, akkor az üzenetet a RegisterWindowMessage() függvény segítségével hozzuk létre! A függvény visszatérési értéke 0xC000 és 0xFFFF közötti érték. Az üzenetek nem az alkalmazáshoz, hanem egy előredefiniált vezérlőosztályhoz tartoznak.

A ***RegisterWindowMessage()*** függvény által visszaadott üzenetekhez az **ON_REGISTERED_MESSAGE()** makróval kapcsolhatunk üzeneteket. Paramétere az üzenet azonosítója és a kezelőfüggvény neve.

```
const UINT ID_MY_REGISTERED = RegisterWindowMessage(MSGSTRING);
//...
ON_REGISTERED_MESSAGE( ID_MY_REGISTERED, MyRegHandler)
```

A kezelőfüggvénye a következő formátumú:

```
afx_msg LRESULT MyRegHandler(WPARAM wParam, LPARAM lParam);
```

Ne használjuk olyan üzenetek esetén, melyeket csak az alkalmazáson belül küldünk!

2.7. Külső üzenetek

2.7.1. Billentyűüzenetek

Ha a rendszerbillentyűk kivételével bármely billentyűt leütünk, egy **WM_KEYDOWN** üzenet keletkezik. Az üzenetet CWnd::**OnKeyDown(UINT** *nChar***, UINT** *nRepCnt***, UINT** *nFlags* **)** metódus felülírásával tudjuk kezelni. Az nChar paraméter tartalmazza a leütött billentyű virtuális kódját, az nRepCnt a lenyomva tartott billentyű hatására létrejövő ismétlések számát. Az nFlags egy billentyű átalakító kódot ad meg, melyből értesülhetünk az előző billentyűkombinációról, illetve az egyszerre lenyomva tartott billentyűkről. A kezelőfüggvény végén az osztályvarázsló által generált függvény meghívja az ősosztály azonos nevű metódusát.

A **WM_KEYUP** üzenet a billentyű felengedésekor keletkezik, és az előzőhöz hasonlóan kezelhető (OnKeyUp()).

A ***TranslateMessage()*** függvény a billentyűlenyomás és -felengedés üzenetek közé a lenyomott billentyűnek megfelelő WM_CHAR üzenetet helyez.

A karakterbillentyűk feldolgozása a **WM_CHAR** üzenet által meghívott ***OnChar()*** metódusban kényelmesebb, mivel a billentyűk ANSI kódját kapja paraméterként. Az nChar paraméter sztring végére fűzhető, vagy ***CDC::TextOut()*** metódussal egy ablakba, vagy más eszközre közvetlenül is kiírható. Az OnChar() hívását megelőzi az ***OnKeyDown()*** és követi az ***OnKeyUp()*** végrehajtása.

Az általános vezérlőbillentyűk nChar értéke:

Billentyű	nChar érték
Back pace	8
Tab	9
Ctrl+Enter (soremelés)	10
Enter (kocsi vissza)	13
Esc	27

A WM_CHAR üzenetet nem generáló billentyűk virtuális billentyűkódjai (Young, 1998, II/55.old.):

Virtuális billentyűkód	Billentyű
VK_CLEAR	Numerikus 5 (ha a Num Lock ki van kapcsolva.)
VK_SHIFT	Shift
VK_CONTROL	Ctrl
VK_PAUSE	Pause
VK_CAPITAL	Caps Lock
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_END	End
VK_HOME	Home
VK_LEFT / UP / RIGHT / DOWN	←↑→↓
VK_INSERT	Insert
VK_DELETE	Delete
VK_F1...F12	F1...F12
VK_NUMLOCK	Num Lock
VK_SCROLL	Scroll Lock

Ha a felhasználó egy rendszerbillentyűt nyom le (Print Screen, Alt, vagy Alt-tal együtt lenyomott billentyű), a **WM_KEYDOWN** helyett, egy **WM_SYSKEYDOWN** / **WM_SYSKEYUP** üzenet generálódik. Ennek feldolgozását általában a Windows végzi.

2.7.2. Egérüzenetek

A legalapvetőbb egérüzenetek az egérgomb lenyomása és az egér mozgatása hatására keletkeznek. Pl. a bal oldali egérgomb üzenetei: **WM_LBUTTONDOWN**, **WM_LBUTTONUP** és **WM_LBUTTONDBLCLK**. Az egér mozgatása a **WM_MOUSEMOVE** üzenetet generálja. A kezelőmetódusok a következő formájúak: ***CWnd::OnMouseMove*(UINT** *nFlags*, **CPoint** *point* **),** ahol az nFlags paraméter az egér mozgatásával egyidőben lenyomva tartott egérgomb vagy

billentyűgomb kódját tartalmazza. A point az egérkurzor helye a küldés pillanatában az ablak bal felső sarkához viszonyítva.

Az nFlags bitjei egyszerre több gomb lenyomását is tárolják, így ha azt kérdezzük, hogy a baloldali egérgomb le volt-e nyomva a mozgatás alatt, azt a következő módon tehetjük:

```
if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
```

A Ctrl gombot az **MK_CONTROL**, a Shift gombot az **MK_SHIFT** azonosítóval tesztelhetjük.

Az egérüzenet annak az ablaknak érkezik, amelyikre az egér mutat. A ***CWnd::SetCapture()*** üzenettel "elkaphatjuk" az egeret. **Ezután az aktuális szál összes egérüzenete annak az ablaknak érkezik, amelyik meghívta a függvényt függetlenül az egérkurzor helyétől.** Visszatérési értéke annak az ablaknak a mutatója, amelyiknek az egérüzenetek érkeztek előzőleg. NULL, ha nem volt ilyen ablak. A ***CWnd::ReleaseCapture()*** "elengedi" az egeret. A ***CWnd::GetCapture()*** segítségével lekérdezhetjük, melyik ablak kapja az egérüzeneteket.

```
if (GetCapture() != this ) return;
```

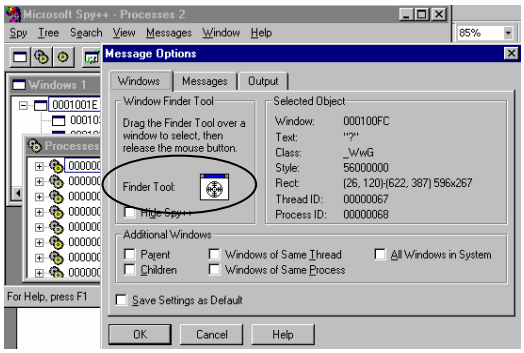
Egyidőben csak egy ablak lehet "captured". Ha az egérkurzor egy másik szál ablaka fölé ér, a rendszer csak addig küldi vissza az üzeneteket, amíg az egérgomb le van nyomva. Háttérben levő ablak nem kaphatja meg az egérüzeneteket, akkor sem, ha "captured".

Az egér elkapása jól használható a drag & drop technikánál.

2.8. A Spy++

A Visual C++ telepít a könyvtárába egy Spy++ nevű eszközt. Segítségével megnézhetjük az éppen futó alkalmazások ablakait, a futó processeket, szálakat, az adott ablaknak küldhető üzeneteket.

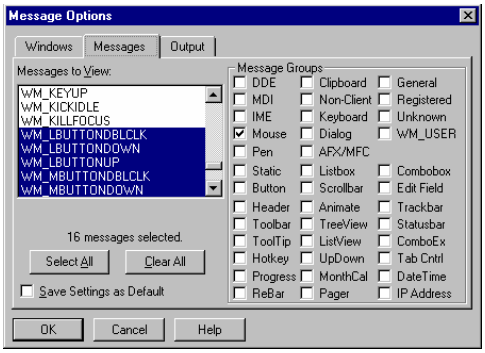
Minket most az üzenetek érdekelnek. A program futtatásakor a Spy menü Messages menüpontját választva a 2.10. ábra képét láthatjuk.



2.10. ábra A Spy++ Message Options ablaka

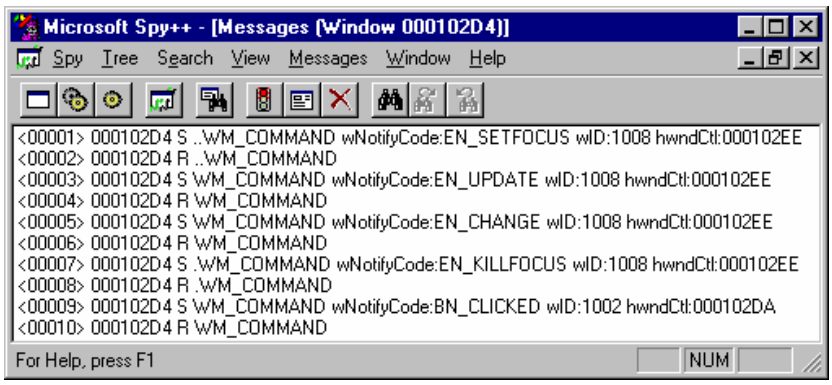
A Finder Tool-t az egérrel a vizsgálni kívánt ablak fölé húzva információkat kaphatunk az adott ablakról, a Messages fület választva vizsgálhatjuk, szűrhetjük az ablaknak küldhető üzeneteket. A 2.11. ábra az egérüzeneteket mutatja.

A szűrést a jelölőnégyzetekben beállítva, a Messages to View ablakban kiemelve jelennek meg az üzenetek. Ez azért is előnyös, mert a Messages to View ablak üzenetei között gyors görgetéskor könnyebben vesszük észre a keresett üzenetet.



2.11. ábra A kiválasztott ablaknak küldhető üzenetek a Spy++-ban

Az ablakot és a vizsgált üzeneteket kiválasztva a Spy++ mutatja az ablaknak érkező üzeneteket. A 2.12. ábra a vezérlők által a szülőablaknak küldött WM_COMMAND üzeneteket mutatja. Egy szerkesztőmező módosítás után egy nyomógomb választása következett.



2.12. ábra A szülőablak gyermekablakától kapott üzenetei

2.9. Kérdések

1. Hogyan épül fel egy Visual C++ alkalmazás?
2. Ismertesse az alkalmazásosztály főbb tagfüggvényeit, azok felelősségét!
3. Ismertesse az üzenetkezelő ciklus szerkezetét!
4. Mondjon példákat az esemény - üzenet - üzenetkezelő metódus kapcsolatra!
5. Milyen eszközzel, és hogyan csatolhatunk üzenetet egy eseményhez?
6. Mi a message map, és hogyan kerülhetnek bejegyzések a message mapbe?
7. Milyen üzenetkategóriákat ismer, és hogyan csatolunk kezelőt az egyes kategóriák üzeneteihez?
8. Hogy történik a külső és belső üzenetek kezelése?
9. Hogyan dolgozzuk fel a billentyűzet üzeneteket?
10. Mit jelent az egér "elkapása"?
11. Mi a különbség a szinkron és aszinkron üzenetküldés között?
12. Mely osztályok kezelhetik, és milyen sorrendben az ablak- és a parancsüzeneteket?

2.10.Összefoglalás

- MFC-t használó alkalmazásainkban nincs szükség főprogram írására. A **WinMain** kódját az MFC biztosítja. A WinMain hívja meg az alkalmazásobjektumra a főbb függvényeket.
- Az **InitInstance** feladata az alkalmazáspéldányok inicializálása, a **Run** futtatja az üzenetkezelő ciklust, biztosítja az üzenetek feldolgozását az üzenetsorból, és

a pihenőidős üzenetek számára. WM_QUIT üzenet esetén hívja az **ExitInstance** metódust kilépés előtt.

- Az üzenetkezelő ciklust az 2.3. ábrán követhetjük végig.

- Az **esemény** azonosítható **pillanatszerű történés**. Esemény hatására üzenet kerül az üzenetsorba. **Az üzenet egy struktúra**, mely tartalmazza az üzenet címzettjeit, azonosítóját és más, az eseményre vonatkozó adatokat. Az üzenetekhez **üzenetkezelő metódusokat** kapcsolunk, melyek végrehajtása az üzenet hatására történik meg. Az üzenet és a kezelő metódus közti kapcsolatot a message map tárolja.
- A **külső üzenetek** előbb a **RIT**-be, majd a szálak üzenetsorába kerülnek, míg a **belső üzenetek** közvetlenül **a szálak üzenetsorába** érkeznek. **A billentyűzet üzeneteit a fókuszált ablak kapja**, míg **az egérüzeneteket** többnyire **az az ablak, amire az egérkurzor mutat**. Ezt az elvet az egér "elkapásával" oldhatjuk fel. Ilyenkor annak az ablaknak érkeznek az üzenetek amelyik meghívta a SetCaptured() függvényt. Az egér elkapását a drag & drop technikánál használhatjuk.
- **Aszinkron üzenet**ek a CWnd::PostMessage() függvénnyel küldhetők, a szál üzenetsorába kerülnek, a hívónak nem kell megvárni végrehajtásukat. **Szinkron üzenet**et a CWnd::SendMessage() függvény segítségével küldhetünk. Az üzenet közvetlenül a WindowProc metódusba kerül, végrehajtása végéig a hívó szál nem dolgozik tovább.
- Az **ablaküzenet**eket ablakok kezelik. A **parancsüzenetek**et nemcsak ablakok, hanem más osztályok is kezelhetik.
- Készíthetünk **saját üzenetek**et, melyeket a Windows üzeneteihez hasonlóan a message map segítségével kezelhetünk. Lehetőség van a parancsok és vezérlők **csoportos kezelés**ére is.

3. Párbeszédablakok

A párbeszédablakok vezérlőinek megismerésével a gyakorlatban is változatos lehetőségek kínálkoznak az üzenetek feldolgozására. Áttekintjük a különböző módszereket, melyekkel a Visual C++ a vezérlőket el tudja érni, és megvizsgáljuk a párbeszédalapú alkalmazások osztályszerkezetét.

3.1. Erőforrások

Az ikonokat, kurzorokat, bittérképeket, sztringeket, menüket, párbeszédablakokat, eszköztárakat, karakterkészleteket és gyorsgombokat összefoglaló néven **erőforrás**oknak nevezzük. Az erőforrások az .exe fájl adatai, de nem a program adatszegmensében találhatóak.

Az erőforrásokat csak akkor tölti be a memóriába a futó alkalmazás, ha szükség van rájuk. Az erőforrások a programpéldányok között megosztásra kerülnek, tehát csak egy példányban lesznek a memóriában.

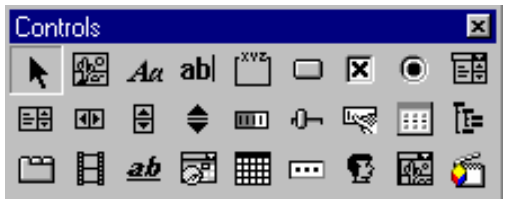
Az erőforrások létrehozására, szerkesztésére a Visual C++ környezet az **erőforrás-szerkesztő**t biztosítja. A resource editor .rc kiterjesztésű erőforrásleíró állományokat hoz létre. Ezek felelnek meg a forrásnyelvű kódnak, text módban megnyitva őket szöveges fájlként olvashatóak. Az .rc fájlok lefordításával jönnek létre a bináris .res állományok.

Az erőforrások elemeire azonosítók segítségével hivatkozhatunk. Az azonosítók define konstansok, valójában egy-egy számot jelentenek. Az azonosítókat a resource.h fejlécfájlban elhelyezve az azonosítókra való hivatkozás a programból és

az erőforrásleíró fájlból azonos módon történhet. Az erőforrás-azonosítókra vonatkozó megállapodásokat az adott erőforrás tárgyalásánál ismerteti a könyv.

3.2. Vezérlők

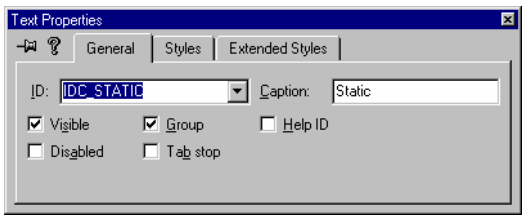
A párbeszédablakok felületét az erőforrás-szerkesztő (resource editor) segítségével szerkeszthetjük kényelmesen. Ha a Workspace / ResourceView / Dialog listájából kiválasztunk egy párbeszédablakot, a szerkesztéshez megjelenik a **Controls** eszköztár. (3.1. ábra)



3.1. ábra A Controls eszköztár

Az eszközikonok drag&drop technikával a kívánt helyre húzhatók, méretük az ablak méretének állításával megegyező módon állítható.

A vezérlők kezdő tulajdonságai - ha a kijelölt vezérlőn jobb egérgombbal kattintunk - a kinyíló menüből a **Properties**-t választva beállíthatók. (3.2. ábra)

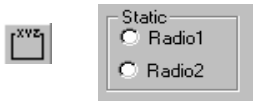


3.2. ábra A tulajdonságablak

A következő vezérlők mindegyike **IDC_STATIC** azonosítót kap. Megjelenítésre szolgálnak, így nincs szükségük egyedi azonosítóra.


Szöveg (**Static Text**) megjelenítésére szolgál. Kijelölt állapotban begépelhetjük a megjelenítendő szöveget.


Kép (**Picture**) vezérlő. Alapértelmezésben egy keretet ad, de a Properties / General fül / Type listájából ikont vagy bitmapet is választhatunk. Az utóbbi két választásnál, ha van az alkalmazásunknak ikon vagy bitmap erőforrása, az Image listájából kiválasztható a megjelenítésre váró kép.



Csoportablak (**Group Box**). Nem hoz létre valóságos csoportot, csak a látvány része. A Static szöveg átírható a kijelölés után közvetlenül vagy a Properties **Caption** mezőjében.

Ha a program futása közben mégis módosítani szeretnénk a megjelenített szöveget vagy képet, az azonosítót (ID) egyedire kell állítanunk. Csak az egyedi azonosító jelenik meg az osztályvarázsló azonosítói között, így annak küldhetünk üzenetet, ahhoz rendelhetünk változót. A **CStatic** osztály tartozik a vezérlőkhöz, (a csoportablakhoz a CButton), melynek segítségével a megjelenített adat módosítható.

A **nyomógomb** (**Push Button**) az alkalmazás futásának irányítására szolgál.

A **jelölőnégyzet**tel (**Check Box**) igen / nem választásokat tehetünk. A szöveg tájékoztat bennünket, milyen opciót állít be az adott gomb.

A **választógomb** (**Radio Button**) több egymást kizáró lehetőség közül választ. Többnyire megjelenítéskor a csoportablakkal jelöljük az egy csoporthoz tartozó gombokat. A csoport első gombját a Properties / General / Group jelölőnégyzet kiválasztásával határozzuk meg. A csoport végét új csoport kezdetének kijelölésével adhatjuk meg. Az egy csoportba tartozó gombok azonosítói egymás után következnek.

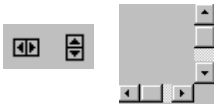
A gombokhoz **CButton** típusú vezérlő tartozik. A BS_ kezdetű Button styles segítségével adható meg, milyen típusú gombot jelenítünk meg. Ha azt akarjuk, hogy egy gombon kép jelenjen meg, a **CBitmapButton** osztályt kell hozzárendelnünk. (3.3.6. feladat) Ehhez a gomb stílusát **BS_OWNERDRAW**-ra kell állítanunk. (Properties / Styles / Owner draw jelölőnégyzet.) Létrehozhatunk különböző alakú gombokat is e stílus beállításával. Ez esetben, ha a gombot frissíteni kell, meghívja a ***DrawItem()*** tagfüggvényét, melyben mi adhatjuk meg a kirajzolt alakzatot. (5.5.4. feladat) Duplakattintás eseményt is csak owner draw stílusú gomb tud feldolgozni. (5.5.5. feladat)

A **szerkesztőmező**, **szerkesztődoboz** vagy beviteli mező (**Edit Box**) feladata a szöveges adatbevitel biztosítása. Jellegzetes üzenetei az **EN_CHANGE**, mely a doboz tartalmának megváltozására hívódik meg, és az **EN_UPDATE**, melyet a tartalom frissítése érdekében hívunk. A Properties / Styles / Multiline beállításra (Ctrl Enter segítségével) többsoros szöveg is bevihető az ablakba. A hozzá kapcsolható objektum **CEdit** típusú.

Hasonló célú a **Rich Edit** ActiveX vezérlő, mely karakterformázásokat tesz lehetővé. Objektuma **CRichEditCtrl** típusú.

Az **IP cím** (**IP Address**) vezérlő olyan szerkesztődoboz, melybe egy Internet Protocol cím írható. Az objektum osztálya a

CIPAddressCtrl osztály.



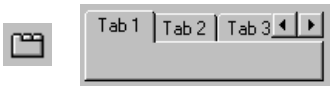
Vízszintes és függőleges görgetősávok (**Vertical and Horizontal Scroll Bar**). Amennyiben az ablak mérete meghaladja a megjeleníthető mértéket, lehetővé teszi az ablak tartalmának görgetését. A vezérlők is ablakok, tehát vezérlők mellett is alkalmazható. Objektuma a **CScrollBar** osztályba tartozik.

A **léptetőgomb** (**Spin**) és a hozzá tartozó ablak (buddy window) gyakran tűnnek úgy, mintha egy vezérlő lenne az ablakban. A léptetőgomb beállítható úgy, hogy automatikusan állítsa a hozzá tartozó ablak szövegét. Gyakran használatos szerkesztődobozzal együtt numerikus értékek bevitelére. A fölfelé nyíl a maximum érték felé mozdítja a hozzá tartozó ablak értékét. Alapértelmezésben a minimum 100, a maximum érték 0. Tehát lehetséges, hogy a felfelé nyíl csökkenti az értéket, ahogy az alapértelmezett esetben is teszi. Ha nem tartozik hozzá ablak, egyszerű görgetősávként működik. Pl. tab kontrollok esetén. Lásd a Visual C++ Output ablakának alsó tab füleit! A **CSpinButtonCtrl** osztály kezeli. Szélső értékei a ***CSpinButtonCtrl::SetRange()*** függvénnyel állíthatók.

A léptetőgomb használata az 5. gyakorlat 5.3.4. szakaszában, 'A vonalvastagság beállítása' lépésben követhető végig.

A **folyamatkijelző** (**Progress**) lassú tevékenységek követését teszi lehetővé, tájékoztatva a felhasználót a folyamat végrehajtásának állásáról. A **CProgressCtrl** osztályban találjuk a kezelőmetódusokat.

A csúszka (**Slider**) segítségével megváltoztathatunk egy értéket, például egy szín árnyalatát. A **CSliderCtrl** osztály tartozik hozzá.



Többfülű ablakot hozhatunk létre a (**Tab Control**) vezérlő segítségével. A **CTabCtrl** osztály tartalmazza a kezelőfüggvényeket.

Kényelmesebb kezelhetőséget biztosít a tulajdonságablakok használata. A **CPropertySheet** a keretablakosztály, míg a **CPropertyPage** a lapok osztálya, melyeket a fülek határoznak meg. (Lásd: 4.8.7. szakasz!)

A listadoboz (**List Box**) lehetővé teszi a lista elemeinek kiválasztását. A **CListBox** osztály kezeli. Gyakrabban használjuk a kombipanelt.

A **kombipanel** (**Combo Box**) a listadobozt kombinálja egy szerkesztődoboz vagy egy statikus szövegmezővel. A Properties / Styles / Type kombipanelben beállíthatjuk a típusát. **Simple** esetén a listadoboz fix méretű, szerkesztődoboz tartozik hozzá. A **Dropdown** típus a szerkesztőmező jobb

oldalán elhelyezkedő nyíl kiválasztására nyitja le a listát. A **Drop List** is a nyíl hatására nyitja a listát, de itt adat bevitelére nincs lehetőség, csak a listából választhatunk értéket. Jobb helykihasználása miatt közkedvelt vezérlő. A tulajdonságablak Data fülében lehet a lista elemeit megadni, új sort Ctrl Enter segítségével vihetünk fel. A szerkesztődobozba adatokat a **CComboBox** osztály ***AddString*** metódusa segítségével futás közben is felvehetünk.

A kombipanel továbbfejlesztett változata az **Extended Combo Box**, mely képek megjelenítésére is alkalmas. Osztálya a **CComboBoxEx**.

Hogyan vihetünk fel új elemet futás közben a kombipanel listájára? Ehhez el kell döntenünk, mely esemény hatására történjen a felvitel. Ha egyszerűen a szövegmező megváltozása eseményhez csatoljuk a felvitelt, a listába karakterenként kerül be a felvinni kívánt szöveg. Elhelyezünk az ablakon egy nyomógombot, melyet kiválasztva kerül az adat a listára.

```
void CMyDlg::OnFelvitelButton()
{
 CString szoveg;
 GetDlgItemText(IDC_COMBO,szoveg);
 ((CComboBox*)GetDlgItem(IDC_COMBO))->AddString(szoveg);
}
```

Természetesen a felvitel kódjában ellenőrizhetjük, hogy ne vigyük föl többször ugyanazt az értéket!

Megtehetjük, hogy a felvitel gomb csak akkor jelenik meg, ha írni kezdünk a kombipanel szövegdobozába. Ez a **CBN_EDITCHANGE** üzenet kezelését jelenti.

```
void CMyDlg::OnEditchangeCombo()
{
 GetDlgItem(IDC_FELVITEL_BUTTON)->ShowWindow(SW_SHOW);
}
```

Ekkor a felvitel végén gondoskodni kell a gomb eltüntetéséről.

```
 GetDlgItem(IDC_FELVITEL_BUTTON)->ShowWindow(SW_HIDE);
```

A ResourceView erőforrásainál már találkoztunk az azonosítókkal, de vezérlők felvitelekor elkerülhetetlen az azonosítók megadása. Egy új vezérlő felvételekor az erőforrás-szerkesztő generál az adott vezérlőnek egy azonosítót, de mint a programozásban mindenütt célravezetőbb a "beszédes" azonosítók használata. Tehát vezérlőink azonosítóit nevezzük át úgy, hogy a nevük utaljon a feladatukra!

Megállapodás:

Az azonosítókat csupa nagybetűvel írjuk, és az _ jellel választjuk el a szavakat egymástól. Az azonosító első tagja utal az azonosító típusára.

IDC_ Vezérlő (Control, Child window)

IDP_ Üzenetablak (Message box) prompt

IDD_ Párbeszédablak (Dialog box)

ID_, (IDM_) Eszközsor vagy menüsor eleme

IDR_ A kerettel kapcsolatos erőforrások (Frame-related resources)

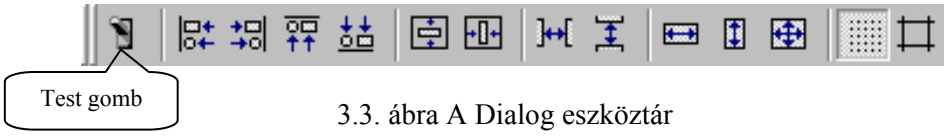
IDW_ Eszköztárak

IDB_ Bitmapek

További információk az azonosítók elnevezéséről és a hozzájuk tartozó számértékek tartományairól a súgó Technical Note 20. pontjában találunk.

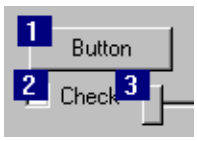
3.2.1. A párbeszédfelület szerkesztése

A vezérlők elrendezését segíti a Dialog eszköztár (3.3. ábra), mely automatikusan megjelenik a képernyő alján, ha egy Dialog erőforrást szerkesztünk. Segítségével beállíthatjuk a vezérlők méretét, a távolságot közöttük, a szélső elemhez rendezhetjük az eszközgombokat, megjeleníthetjük a rácspontokat és fordítás nélkül megnézhetjük, hogy futás közben milyen lesz az ablakunk képe (Test gomb) (5. Gyakorlat 25. ábra).



3.3. ábra A Dialog eszköztár

A vezérlőkön a TAB gomb segítségével végiglépegethetünk. E sorrend beállítását segíti a **Layout menüpont**, mely Dialog erőforrás szerkesztése esetén jelenik meg a főmenüben. Tab Order (Ctrl D)-t választva a vezérlők sarkában megjelenő számok jelzik a sorrendet. (3.4. ábra) Az egérrel a vezérlőkre kattintva a sorrend átállítható. A tabulátor csak azokra a vezérlőkre lép rá, melyekre a tulajdonságablakban be van állítva a Tab stop (General fül). Ezt nekünk nem kell beállítanunk, hisz ez az alapértelmezés, de ha kivesszük a pipát a Tab stop jelölőnégyzetből, akkor a vezérlőt átlépi a tabulátor.



3.4. ábra A tabulátor sorrend beállítása

3.3. Párbeszédablak

A párbeszédablak (**Dialogbox**) a feladat elvégzéséhez szükséges paraméterek beszerzésére használatos. Általában több vezérlőt tartalmaz. Segítségükkel a

felhasználó szöveget... vihet be, adatot választhat a felkínált adatok közül, közvetlen parancsokra adhat utasítást...

3.3.1. Előredefiniált párbeszédablak

Az előredefiniált párbeszédablak (**predefined dialog box**) vagy **üzenetablak** (**message box**) függvény segítségével hívható. Ha egy ablakban vagyunk, akkor a ***CWnd::MessageBox*** tagfüggvénye hozza létre, ha nem ablakosztályban írjuk a kódot (pl. alkalmazásosztály), a globális ***AfxMessageBox*** hívással küldhetünk üzenetet.

Az ablak bizonyos jellemzőit, mint függvényparamétereket mi is meghatározhatjuk a híváskor. Megadhatjuk:

- az üzenet szövegét
- az ablak címét (fejlécét, caption mező)
- a benne megjelenített választógombokat
- a benne megjelenített ikonokat.

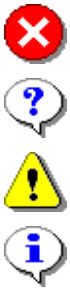
Az utóbbi kettőt egy előre megadott listából választhatjuk ki.

Az üzenetkezelő függvények alapértelmezett paraméterekkel (OOP 1.4.2. fejezet) rendelkeznek, ezért volt elegendő az eddigi feladatainkban csak a kiírt üzenet szövegét megadni.

int CWnd::MessageBox(LPCTSTR *lpszText*, **LPCTSTR** *lpszCaption* = **NULL, UINT** *nType* = **MB_OK);**

int AfxMessageBox(LPCTSTR *lpszText*, **UINT** *nType* = **MB_OK, UINT** *nIDHelp* = **0);**

Az nType paraméter értéke a message box (MB_) kezdetű flagek megadásával lehetséges. Az ikonokat a 3.5. ábra mutatja.



MB_ICONHAND,
MB_ICONSTOP,
MB_ICONERROR

MB_ICONQUESTION

MB_ICONEXCLAMATION
MB_ICONWARNING

MB_ICONASTERISK
MB_ICONINFORMATION

3.5. ábra Az üzenetablak system ikonjai

A gombok MessageBox esetén a következő értékek közül választhatók ki:

- **MB_ABORTRETRYIGNORE** Három nyomógomb: Abort, Retry, és Ignore.
- **MB_OK** Egy gomb: OK.
- **MB_OKCANCEL** Két gomb: OK és Cancel.
- **MB_RETRYCANCEL** Két gomb: Retry és Cancel.
- **MB_YESNO** Két gomb: Yes és No.
- **MB_YESNOCANCEL** Három gomb: Yes, No és Cancel.

Az nType értéke alapértelmezésben egy OK gomb, de a fentiek kombinációjaként is megadható pl.: `MB_ICONERROR | MB_OK`.

Megjegyzés:

Az MFC biztosít készen néhány CDialog osztályból származó párbeszédablakot, ezeket a 4.8.7. szakasz tárgyalja.

3.3.2. Szerkesztett párbeszédablak

A szerkesztett párbeszédablak (custom-made dialog box) összes jellemzőjét mi adjuk meg.

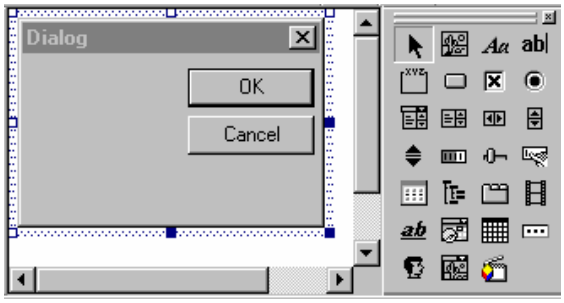
- méretét
- címét
- a benne szereplő vezérlők, ablakok
  - fajtáit
  - számát
  - elhelyezkedését
  - jellemzőit.

A párbeszédablak felületét az erőforrás-szerkesztővel szerkesztjük.

Workspace

ResourceView

Dialog Megnyitjuk a kiválasztott ablakot a szerkesztésre. (3.6. ábra)



3.6. ábra Párbeszédablak az erőforrás-szerkesztőben.

A 3.1. fejezet leírása alapján a párbeszédablak szerkeszthető.

A vezérlők kezelésének lehetőségeit a 3.2.4. szakasz tárgyalja.

3.3.3. Változó hozzárendelése a vezérlőhöz

A párbeszédablakok megjelenítése és megszűnése között kommunikálni szeretnénk a vezérlőkkel. Ehhez szükségünk van egy ideiglenes változóra vagy tagváltozóra, melynek a vezérlőt irányító tagfüggvényhívásokat elküldhetjük. A vezérlőhöz rendelt objektummutatót a CWnd::GetDlgItem() metódussal kérdezhetjük le, míg tagváltozót az osztályvarázsló segítségével rendelhetünk hozzá.

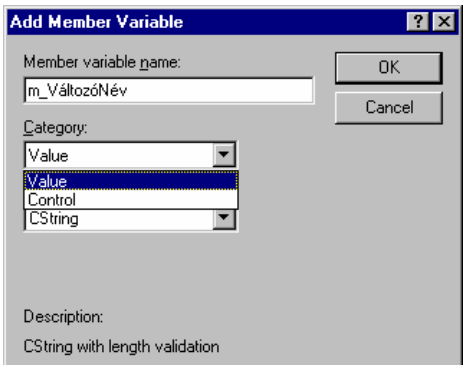
A párbeszédablak feladata, hogy a program adatait a képernyőre vigye, és a felhasználó által megadott értékeket olvassa be. Az adatokra többnyire szükségünk van az ablak megjelenítése előtt és után is, tehát tárolnunk kell őket az ablakosztály tagváltozóiban. Az MFC biztosítja a vezérlők és az adattagok közti kapcsolatot fenntartó függvényeket. Mivel a tagváltozók és a vezérlők között többnyire kölcsönösen egyértelmű a kapcsolat, a hozzárendelés az osztályvarázsló segítségével könnyen megvalósítható.

3.3.3.1. Hozzárendelés az osztályvarázsló segítségével

Osztályvarázsló

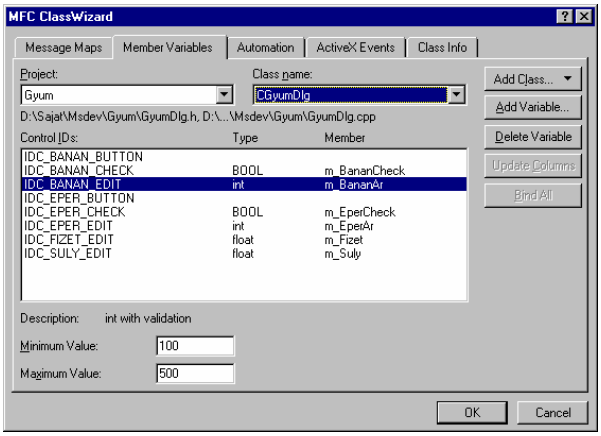
Member Variables fül, az osztály és az ID kiválasztása

Add Variable



3.7. ábra Vezérlőhöz változó rendelése

A vezérlőhöz több különböző típusú változót rendelhetünk. Eldönthetjük, hogy a változó vezérlő lesz-e, vagy értéket tárol. (3.7. ábra) A hozzárendelhető változók típusa a vezérlőtől függ. Control kategória esetén a vezérlőnél (3.1. fejezet) említett típust választhatjuk, míg érték esetén általában több típus közül választhatunk. Példaként vizsgáljuk meg egy szövegdoboz (Edit box) vezérlő esetét!



3.8. ábra. ábra Az osztályvarázsló Member Variables füle

- A Value kategória típusai
  - CString: Beállítható a sztring maximális hossza is, az osztályvarázsló Member Variables fül Maximum Characters mezőjében.
  - int / UINT / long / DWORD / float / double / BYTE / short
  - BOOL / COleDateTime / COleCurrency

  Mindegyik numerikus érték alsó és felső határral. (3.8. ábra)

- A Control kategória egy CEdit objektumot rendel a vezérlőhöz, melyre értelemszerűen hívhatók a metódusai, miközben a megjelenítése fordításidőben szerkeszthető.

Egy vezérlőhöz azonos kategóriából csak egy változó rendelhető, de hozzárendelhetünk egy Value és egy Control kategóriájú adattagot egyszerre is.

Megjegyzés:

A ResourceView-ban a vezérlőn Ctrl + Dupla kattintás hatására, ha adat rendelhető a vezérlőhöz (pl. szerkesztődoboz, combo box...), az osztályvarázsló Add Member Variable (3.7. ábra) ablaka hívódik meg. Ha a vezérlőhöz inkább üzenetkezelő metódust rendelünk (pl. nyomógomb) az üzenethez csatolt metódus kódja jelenik meg, és szerkeszthető. Ha még nincs kezelő hozzárendelve, alapértelmezett néven metódust csatol, s ennek kódját szerkeszthetjük.

3.3.3.2. Kapcsolattartás a vezérlő és a tagváltozó között

Az adatok kezelését a következő MFC függvények teszik lehetővé:

- A ***DDX (Dialog Data Exchange)*** függvények feladata a párbeszédablak vezérlője és a hozzá rendelt változó közti adatcsere-kapcsolat létrehozása. Egymáshoz rendeli a tagváltozót és a vezérlőt.

- A ***DDV (Dialog Data Validation)*** függvények feladata az osztályvarázslóval a változónak beállított értékhatárok ellenőrzése.

- Az ***UpdateData(TRUE / FALSE)*** függvény feladata az adatcsere biztosítása az adott ablak összes vezérlője és a hozzájuk rendelt változók között. A függvény **az adatok módosítását az ablak összes vezérlőjére elvégzi** egymás után. TRUE paraméter esetén a vezérlőkbe beírt adatokat olvassa be a hozzájuk rendelt tároló adattagokba, míg FALSE paraméter esetén a változókban tárolt értékeket jeleníti meg a vezérlőkön. Paraméter nélkül hívva az alapértelmezett TRUE értékkel hívódik meg, tehát az adatok a képernyőről az adattagokba kerülnek.

  Az UpdateData használata elkerülhetetlen, ha a felhasználó által megadott adatokkal valamilyen műveletet kívánunk elvégezni. A lépések ilyen esetben a következők:

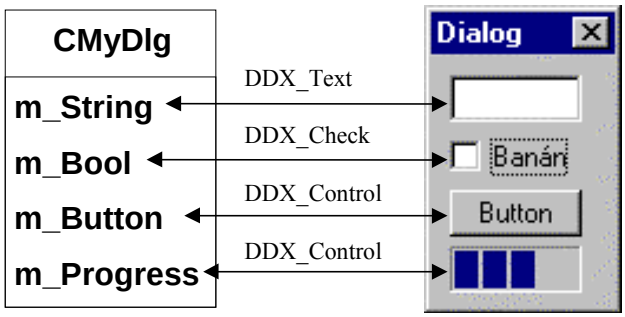
  - ♦ A felhasználó megadja az értéke(ket)t.
  - ♦ UpdateData(TRUE) hívással beolvassuk azokat a tagváltozókba.
  - ♦ Elvégezzük a számításokat. Új értékek kerülnek a tagváltozókba.

- Az eredményt UpdateData(FALSE) hívással írjuk ki a képernyőre.

Az UpdateData hívásokat jól meg kell gondolni. Ne feledjük, hogy az UpdateData() az összes adattagba beolvas / kiír. Tehát, ha még ki nem írt adatot tárolunk valamelyik tagváltozóban, s egy újabb vezérlőből szeretnénk értéket beolvasni, a tárolt adatunk is fölülíródik a beolvasás során. Fordított esetben, ha a felhasználó által bevitt értékek beolvasása nélkül akarunk egy számított értéket megjeleníteni, a felhasználó adatait felülírja a tagváltozók tartalma. A 3.1. gyakorlatban a súly alapján a fizetendő érték kiszámítása mutatja az UpdateData kezelését. (Az események kódja.)

Az osztályvarázsló a CMyDlg::DoDataExchange függvényében adja meg a DDX és DDV metódusokat.

```
DDX_Text(pDX, IDC_BANAN_EDIT, m_BananAr);
DDV_MinMaxInt(pDX, m_BananAr, 100, 500);
DDX_Check(pDX, IDC_BANAN_CHECK,m_BananCheck);
DDX_Control(pDX, IDC_PICTURE, m_Picture);
DDX_CBIndex(pDX, IDC_COMBO, m_Combo);
```



3.9. ábra Változók és vezérlők

Megjegyzés:

Az osztályvarázsló nem minden DDX lehetőséget kínál föl a hozzárendeléshez. Az összes lehetőség felsorolása nem célja a könyvnek. A gyakorlati rész feladati közt látunk néhány példát arra, hogy milyen további lehetőségeink vannak. (Tömb kezelés 3.1.2. gyakorlat, 3.3.6. feladat CBitmapButton, 4.2.6. feladat CDragListBox...)

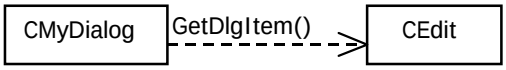
3.3.4. Vezérlők kezelésének lehetőségei

3.3.4.1. Ideiglenes mutatón keresztül

A vezérlőt tartalmazó ablak bármely metódusában a **GetDlgItem**(IDC_vezérlő) függvénnyel egy CWnd* mutatót kapunk a vezérlőre, melyre a CWnd osztály metódusai hívhatók, s melyet szükség esetén átkonvertálhatunk a vezérlőnek megfelelő osztályra. (3.10. ábra)

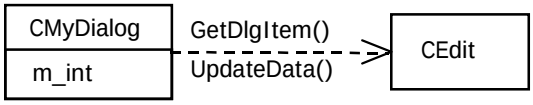
```
GetDlgItem(IDC_BANAN_EDIT+m_Banan)->EnableWindow(FALSE);
((CButton*)GetDlgItem(IDC_BANAN+m_Banan))->SetCheck(0);
```

Ha több metódust is hívunk a vezérlőre, a GetDlgItem által visszaadott mutatót tárolhatjuk egy ideiglenes mutatóban.



3.10. ábra Vezérlő elérése GetDlgItem metódussal

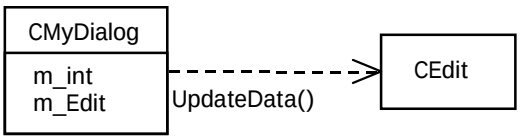
Ide sorolható még az az eset, amikor a vezérlőhöz Value kategóriájú változót csatolunk, mert a vezérlőben megjelenített adatokat az **UpdateData** metódussal frissíthetjük, míg közvetlenül a vezérlőnek szóló tagfüggvényhívásokat a GetDlgItem segítségével valósíthatjuk meg. (3.11. ábra)



3.11. ábra Vezérlő elérése GetDlgItem és UpdateData metódussal

3.3.4.2. Objektum segítségével, üzenetkezelő metódusok a szülőablakban

A vezérlő osztályának megfelelő objektumot is hozzárendelhetünk a vezérlőhöz, ha az osztályvarázslóval a Control kategóriát választjuk. Ez az objektum a szülőablakosztály minden metódusából elérhető, és a Control osztály összes metódusa hívható rá. A Control objektum (OOP 1.8.2.) ugyanahhoz a vezérlőhöz kapcsolt Value kategóriájú változó mellett is használható. (3.12. ábra)



3.12. ábra Vezérlő elérése Control objektummal és UpdateData metódussal

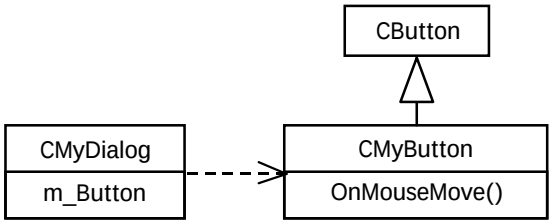
Control objektumot vagy lokális objektumot egyszerűen fel is vehetünk az osztályban (pl. CEdit edit). Ekkor a Create metódus segítségével tudjuk az ablakba kitenni. A Create metódus használata körülményesebb, mint az eddig említett módszerek, pl. nekünk kell kiszámolnunk a vezérlő pontos helyét az ablakban, de bizonyos feladatokhoz hasznos az ilyen feldolgozás.

Az előzőekben a vezérlőknek érkező üzeneteket az őket tartalmazó ablak kezelte. Az osztályvarázsló Message Maps fülében, a Class name listából kiválasztva az ablakosztályt, az Object IDs listából pedig a vezérlő azonosítóját a Messages lista mutatja az üzeneteket, melyekhez metódust rendelhetünk. (2.4. ábra) Például nyomógomb esetén az ON_CLICKED, ON_DOUBLECLICKED üzenetet. (A vezérlőn jobb egérrel kattintva, s az Events menüpontot választva hasonlóan láthatóak a kezelhető üzenetek.)

3.3.4.3. Származtatással, új üzenetkezelő metódusok a vezérlő osztályban

A gomb fölött mozgatva az egeret a **WM_MOUSEMOVE** üzenet keletkezik. Ehhez szeretnénk kódot rendelni. (4.2.3. feladat) Ekkor a CButton osztályba kellene megírnunk az OnMouseMove metódust, de ezt az osztályt az MFC-vel kaptuk, nem a sajátunk. Megoldás: származtassunk egy saját CMyButton osztályt, s kezeljük benne az előbbi üzenetet! Mivel a CButton osztály a CWnd utóda, így CMyButton meg fogja kapni a WM_MOUSEMOVE ablaküzenetet. (3.13. ábra)

A létrehozott új osztály bekerül az osztályvarázsló Add Variable ablakának Control kategória Variable Type listájába, így választható a vezérlőhöz rendelt objektum osztályaként.



3.13. ábra. ábra Származtatott vezérlő pl. nyomógomb

Ha már nagyon szeretnénk a tanultakat a gyakorlatban kipróbálni, a 3.1. gyakorlat 3.1.2. részét végigkövethetjük eddigi ismereteinkkel.

3.3.5. A párbeszédablak életciklusa

- **Létrehozása** a konstruktor feladata. Ha egy adattagja (tagváltozója) állandó kezdőértékű, annak értékét a konstruktorban állítsuk be!
- **Megjelenítése** (a 4.8.3. fejezetben részletesebben tárgyalva) a ***DoModal*** vagy a ***Create*** tagfüggvény hívásával történik. (Create esetén csak WS_VISIBLE stílus esetén lesz látható.)

  A konstruktor által, vagy később az ablakobjektum tagváltozóiban beállított értékek – ha a változóhoz vezérlőt rendeltünk – DoModal hívásakor megjelennek az ablakban. **A DoModal tartalmaz egy UpdateData(FASE) hívást.**
- Az adatok módosítása után az ablakot bezárjuk.

  Ha a megjelenített ablak adatait nemcsak egyszerűen módosítjuk, és értéküket beolvassuk, hanem pl. számításokat végzünk az adatokkal, s azok eredményét is meg akarjuk jeleníteni az ablak vezérlőiben, akkor szükség lesz az adatok megszerzésére a vezérlőktől. A tagváltozókba beolvasott adatokon végrehajtott tevékenységek után, a kapott eredményeket a képernyőre akarjuk vinni. Az **adatcsere** a vezérlő és a hozzá rendelt tagváltozó közt Value kategória esetén ***UpdateData*** függvényhívással valósítható meg.
- Az ablak **megszüntetése** (a ***DestroyWindow*** és a destruktor feladata).

Alapértelmezésben a párbeszédablak tartalmaz egy OK és egy Cancel gombot, melyekhez az ***OnOK*** és ***OnCancel*** kezelőfüggvények tartoznak. OK gomb választása esetén az OnOK kezelő beolvassa a vezérlőtől az adatokat a változókba, tehát a változók értékei az ablak bezárása során módosulnak. **Az OnOK metódus tartalmaz egy UpdateData(TRUE) hívást.** Cancel választása esetén az OnCancel nem módosítja a változók adatait, tehát az ablakba felvett értékek és beállítások nem kerülnek át az adattagokba. A CDialog osztály két függvénye felülírható a CMyDlg utódosztályban az **IDOK** és **IDCANCEL** azonosítók **BN_CLICKED** üzenetéhez rendelt kezelőfüggvénnyel. Ne felejtsük el a felülírt függvények végén az ősosztály azonos nevű metódusait meghívni (CDialog::OnOK() / CDialog::OnCancel())!

Párbeszédalapú alkalmazás (dialog based application) esetén **az alkalmazásosztály *InitInstance()* függvénye a következő szerkezetű**:

```
// virtual
 BOOL CNonDocViewApp::InitInstance()
{
CNonDocViewDlg dlg;
 m_pMainWnd=&dlg;
 int nResponse=dlg.DoModal();
 if ( nResponse == IDOK )   {....}
 else if ( nResponse == IDCANCEL ) {...}
return FALSE;                     // Nem indítja a Run-t
}
```

A ***CDialog::DoModal*** tagfüggvény visszatérési értéke int típusú, értékéből megtudhatjuk, melyik gombot (OK/CANCEL) választotta a felhasználó, s ennek megfelelően az elágazásba beírt kód végrehajtódik.

Párbeszédalapú alkalmazásnál az alkalmazásosztály InitInstance tagfüggvényének visszatérési értéke FALSE, emiatt nem indul el az alkalmazás Run metódusa, tehát terminál az alkalmazás.

Megjegyzés:

Enter hatására párbeszédablakban az alapértelmezett gombra a button clicked metódus hívódik meg. Mivel alapértelmezésben az OK gomb a default, Enter hatására többnyire az OnOK() hajtódik végre, míg Esc hatására valamint a bezárás ikon választása esetén az OnCancel hívódik meg. Emiatt, ha az OK és Cancel gombokat letöröljük az ablakról, akkor is megmarad az azonosítójuk az osztályvarázsló ObjectIDs ablakában, hogy a hozzájuk rendelt üzenetkezelő felülírható legyen.

Ha nem akarjuk, hogy Enter hatására bezáródjon az ablakunk, választhatunk más alapértelmezett gombot, vagy írjuk felül az OnOK-t egy üres metódussal (ne hívja meg az ősosztály OnOK-t)! Ekkor az ablakunk az Esc és a bezáró ikon hatására zárul be. Ha szeretnénk OK gombot is, vegyünk föl egy új nyomógombot (Azonosítója ne IDOK legyen!), melynek felirata OK, és a BN_CLICKED esemény kezelője ne a saját OnOK-t hívja, hanem a CDialog::OnOK() tagfüggvényét!

Az Enter hatására bezáródó ablak problémája többnyire szerkesztődoboz esetén vetődik fel. A szöveg begépelése után ösztönösen Entert ütünk, s így bezáródik az alkalmazásunk. Ha nincs szükségünk az OK gombra, ez esetben a probléma feloldható, ha az OK letörlése után szövegdobozunkra a tulajdonságablak Styles fülében a Multiline stílust jelöljük ki.

Megjegyzés:

A ***CDialog::EndDialog(int nResult)*** tagfüggvény segítségével mi is bezárhatjuk párbeszédablakunkat, miközben beállíthatjuk a DoModal visszatérési értékét.

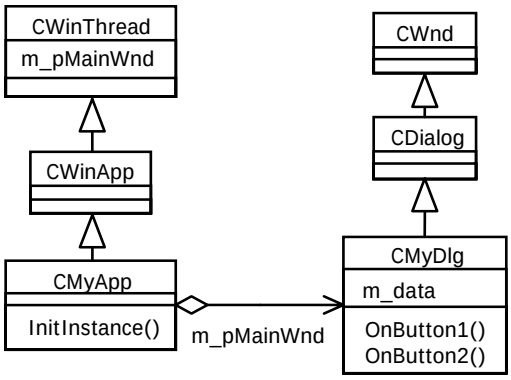
A DoModal visszatérési értéke -1, ha nem jött létre a párbeszédablak, vagy IDABORT, valamilyen más hiba esetén.

Ha egy párbeszédablak **modal**, akkor ahhoz, hogy az alkalmazásban egy másik ablakot választhassunk, be kell zárnunk a modal ablakot. Az előredefiniált párbeszédablak (az üzenetablak) alapértelmezésben ilyen. Az ***AfxMessageBox()*** függvény harmadik paramétere **MB_APPMODAL**. Ez a paraméter állítható **MB_SYSTEMMODAL**-ra is, ami azt jelenti, hogy a párbeszédablak **system modal**, tehát egyetlen más alkalmazásból sem választható ablak ennek lezárása nélkül.

Tipikusan modálisak a figyelmeztető vagy hibaüzenetet küldő ablakok. A Tip of the Day ablak is alkalmazásszintű modális üzenet. Az egér leütését, mozgatását ilyen üzenet esetén is feldolgozza a rendszer, ha a Tip of the Day ablak mögé az alkalmazásra kattintunk, egy halk figyelmeztető hangot hallunk.

3.3.6. A párbeszédalapú alkalmazás szerkezete

Ha az alkalmazásvarázslóval Dialog based alkalmazást generálunk, a 3.14. ábrán látható osztályok alkotják alkalmazásunk vázát. (Az About ablak, mint elhagyható elem nem szerepel az osztálydiagramon.)



3.14. ábra A Dialog based alkalmazás szerkezete

Az alkalmazásosztály a CWinApp osztályból származik, mely a CWinThread utódosztálya. A CMyApp osztály örökli a CWnd* típusú m_pMainWnd adattagot, mely a főablakra mutat. A főablak párbeszédalapú alkalmazás esetén a CDialog osztályból származó CMyDlg osztály objektuma. Mivel a CDialog osztály a CWnd utódosztálya, így az m_pMainWnd, bár CWnd* típusú, valójában utódobjektumra mutat. Tehát a mutatónak szükség esetén explicit típuskonverzió után küldhetők a CMyDlg metódusai, és érhető el m_data adattagja.

A főablak az alkalmazásosztály InitInstance tagfüggvényében jön létre, melynek kódját az előző szakasz `CNonDocViewApp::InitInstance()` kódrészlete ismerteti.

Mivel párbeszédalapú alkalmazás esetén az InitInstance FALSE értékkel tér vissza, így ebben az esetben nem hajtódik végre az alkalmazásobjektum Run metódusa.

3.3.7. Új szerkesztett párbeszédablak

1. lépés: Az új erőforrás létrehozása

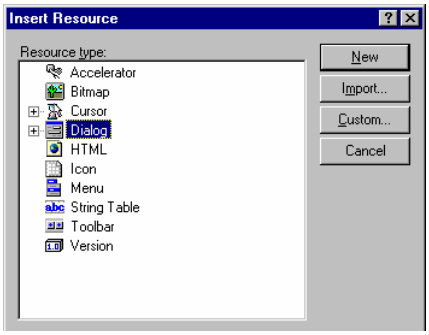
Workspace

ResourceView jobb egérgomb

Insert

Resource type: Dialog

New (3.15. ábra)



3.15. ábra Új erőforrás létrehozása

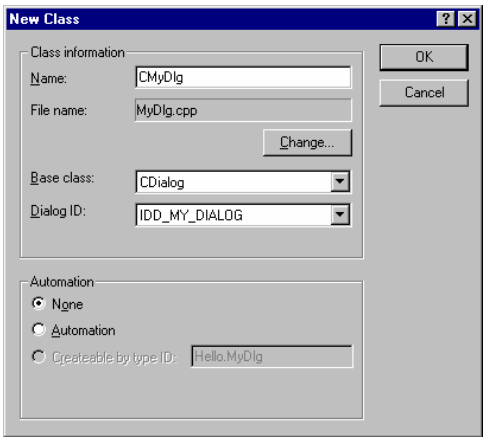
Természetesen ezzel a módszerrel hozhatunk létre más erőforrásokat is.

Módosítsuk "beszédes" nevűre az azonosítót! (Properties, ID: IDD_MY_DLG)

2. lépés: A párbeszédablak felületének szerkesztése

3. lépés: Új osztály csatolása az erőforráshoz

Ha valamely vezérlőhöz üzenetkezelő metódust vagy változót szeretnénk csatolni (vagy egyszerűen csak Ctrl duplakattintás a párbeszédablakon, vagy az osztályvarázsló megnyitásakor (Ctrl W) hatására), megjelenik egy üzenetablak a következő szöveggel: IDD_MY_DIALOG egy új erőforrás, valószínűleg új osztályt szeretnénk csatolni hozzá. Választhatunk új osztály, vagy egy létező osztály csatolása között. A 3.16. ábra mutatja az új osztály létrehozása párbeszédablakot. A Name mezőt kitöltve, és OK / OK-t választva az osztályvarázsló létrehozza az osztály forrásfájljait.



3.16. ábra Új osztály csatolása az új párbeszéd erőforráshoz

4. lépés: Objektum létrehozása

El kell döntenünk, hogy a párbeszédobjektum egy metódus lokális változója, vagy egy osztály adattagja lesz-e. Amelyik fájlban az objektumot létrehozzuk (CMyDlg dlg), oda be kell szerkesztenünk a MyDlg.h fejlécfájlt!

Adattag esetén a kezdőértékadások helye a tartalmazó osztály konstruktora, de az értékek módosíthatók a futás során. Az új párbeszédablak életciklusa a 3.3.5. fejezetben leírtaknak megfelelően:

- Adattagok értékeinek beállítása. `// dlg.m_int=5;`
- DoModal (A beállított értékek megjelenítése.)
  - Közben esetleg adatcsere az UpdateData függvénnyel.
- OnOK (A szerkesztett értékeket beolvassa.)
- Az adattagok értékeinek tesztelése `// if (dlg.m_int == 5){...}`


A 3. gyakorlat 1. szakaszában egy párbeszédalapú alkalmazás fejlesztésének lépéseit követhetjük végig. A 3.3. szakasz további feladatokat tartalmaz a tanultakkal kapcsolatban, melyekhez megoldási ötleteket találunk a 3.4. szakaszban.

3.4. ActiveX vezérlő csatolása

Az **ActiveX** komponensek hordozható szoftverdarabkák, melyeket magunk is készíthetünk, letölthetjük őket az Internetről, vagy megvásárolhatjuk őket. Az ActiveX az **OLE** (**Object Linking and Embedding** = objektumcsatolás és -

beágyazás) technológia kiterjesztése az Internetre multimédia szolgáltatásokkal. (Bennett, 1997, 234.old) A szerkezet alapját ma a **COM** (**Component Object Model**), a továbbfejlesztéseként létrejött **DCOM** (**Distributed COM**), és mára már a **COM+** technológia képezi.

A csatolt ActiveX vezérlőket úgy használhatjuk, mint a standard Visual C++ vezérlőket.

Az .ocx fájl egy speciális DLL. Mint a nevében is benne van dynamic link library, **dinamikusan szerkesztődik alkalmazásunkba**. Nem választhatunk, ahogy az MFC-nél, hogy statikusan, vagy dinamikusan szerkesszük be, tehát telepítéskor az alkalmazással együtt telepítenünk kell.

3.4.1. A file nyilvántartásba vétele

A regisztráció feladata, tárolni az alkalmazások által leggyakrabban használt információkat. Pl. a leggyakrabban használatos fájlokat, a felhasználó által kiválasztott opciókat... Korábban ezt a feladatot az .ini fájlok látták el, melyek könnyen törölhetőek, módosíthatóak voltak, és alkalmazásonként egyet létre kellett hozni. Most ezeket az adatokat a registry adatbázisa tárolja. (Lásd 7. fejezet!) Míg az .ini fájlok text fájlok voltak, addig a registry binárisan tárolja az adatokat, melyek a RegEdit eszközzel, vagy a programból a CWinApp tagfüggvényeivel érhetünk el.

ActiveX vezérlők esetén azért célszerű regisztrálni a vezérlőt, hogy bármely, így az általunk fejlesztett alkalmazásból is könnyen elérhessük.

Ha saját.ocx fájllal dolgozunk, azt célszerű vagy a Windows System32 alkönyvtárába, vagy a projekt alkönyvtárába elhelyezni a könnyebb elérhetőség miatt.

A nyilvántartásba vételt megtehetjük közvetlenül is, ha a Start gomb segítségével futtatjuk a **RegSvr32**-t saját.ocx paraméterrel. A sikeres regisztráció megtörténtét egy üzenetablak jelzi.

Vagy a Visual C++ alkalmazásban, amibe be szeretnénk szerkeszteni a vezérlőt:

Tools menüpont

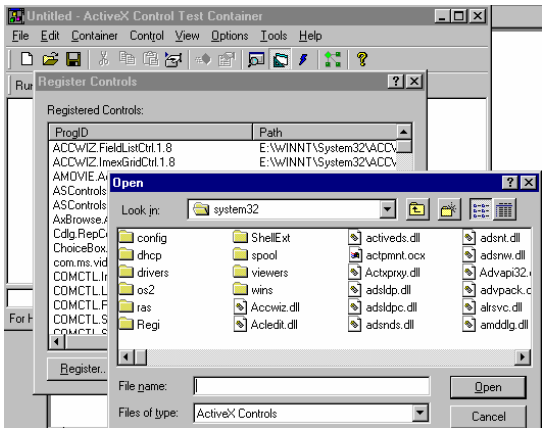
ActiveX Control Test Container TstCon32.exe

File menü

Register Controls

Register gomb (3.17. ábra)

A file-t az ismert ablakban megnyitva a regisztráció végrehajtódik. A regisztrált vezérlők a Register Conrtols ablakban megtalálhatók. Ez az ablak már a telepítéskor is tartalmaz jelentős számú vezérlőt.



3.17. ábra ActiveX vezérlő nyilvántartásba vétele

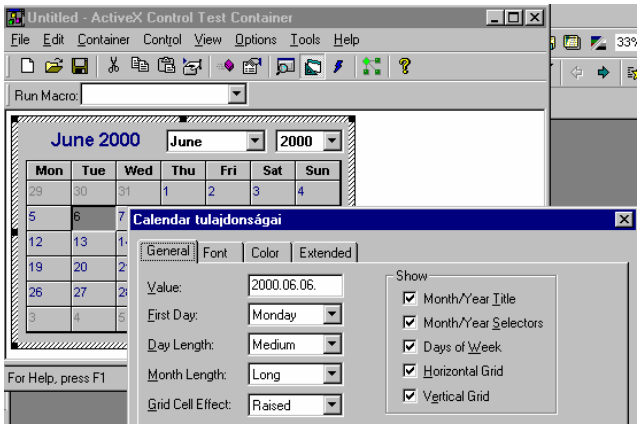
3.4.2. Regisztrált vezérlők tesztelése

Az ActiveX Control Test Container, mint az a nevéből is kiderül, a vezérlők tesztelésére írt alkalmazás.

Edit menü

Insert New Control

Calendar Control **OK**

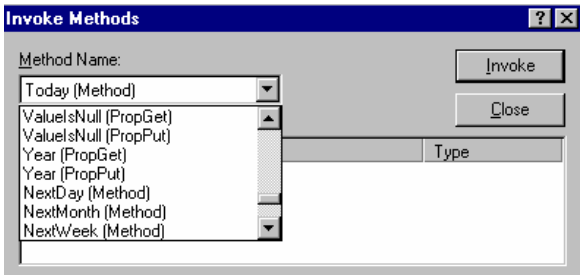


3.18. ábra ActiveX vezérlő tesztelése

A Properties eszközgomb vagy az Edit menü Properties menüpont hatására állíthatjuk a tulajdonságokat. Pl. a hét első napjaként a vasárnapot (First Day:

Sunday) alkalmaz. A háttérben a vezérlő az alkalmaz gomb hatására mutatja a változásokat. Átállíthatók a betű- és háttérszínek, a betűtípusok és méretek...

Az Invoke Methods eszközgomb (Controls menü Invoke Methods) mutatja a meghívható metódusokat. A tulajdonságlap által kijelzett tulajdonságok (PropGet) az állítható tulajdonságok (PropPut), míg a futtatható metódusok (Method) zárójeles megjegyzést kaptak. Próbáljuk ki, hogy átállítjuk a dátumot, majd a Today (Method) választásra az Invoke gombot lenyomjuk! Vagy többször is végrehajthatjuk a NextDay, NextWeek, NextMonth, NextYear, PrevDay...metódusokat. (3.19. ábra)



3.19. ábra ActiveX metódusok tesztelése

A Control menü Draw MetaFile segítségével megnézhető, hogy milyen lesz a vezérlőnk futás közben.

3.4.3. ActiveX vezérlő használata az alkalmazásban

Ahhoz, hogy ActiveX vezérlőt használni tudjunk, elő kell készítenünk az alkalmazásunkat rá.

AppWizard

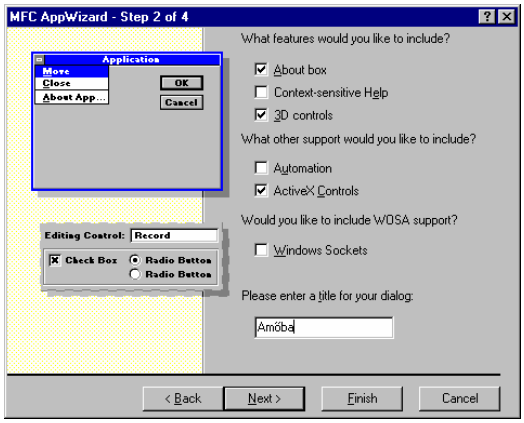
Dialog based application

Step 2 in 4

ActiveX Controls jelölőnégyzet kiválasztva (alapértelmezés)
(3.20. ábra)

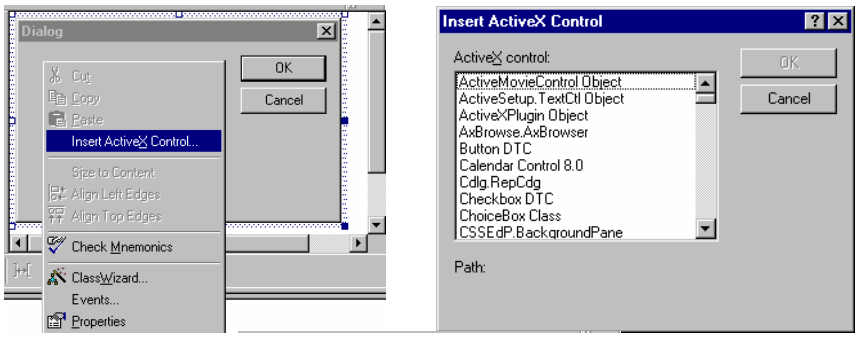
Megjegyzés:

Ha nem jelöltük be az ActiveX Controls jelölőnégyzetet az alkalmazásvarázslóban, utólag az alkalmazásosztály InitInstance függvényéből meg kell hívnunk az ***AfxEnableControlContainer()*** függvényt, és az StdAfx.h-ba be kell szerkesztenünk az Afxdisp.h fejlécfájlt. Ezután tudunk ActiveX vezérlőt beilleszteni az alkalmazásba.



3.20. ábra Az alkalmazásvarázslóban engedélyezzük az ActiveX vezérlő használatát

Ezután az erőforrás-szerkesztő bármely párbeszédfelületén jobb egérrel kattintva a legördülő menüből kiválasztjuk az Insert ActiveX Control menüpontot. (3.21. ábra)



3.21. ábra ActiveX vezérlő beszúrása

Az Insert ActiveX Control párbeszédablakban kiválaszthatjuk a kívánt vezérlőt.

A vezérlő a felületen ugyanúgy szerkeszthető, mint a Visual C++ vezérlői. Tulajdonságai beállíthatók, ha azonban kezelni akarjuk a vezérlőt, hozzá változót akarunk csatolni, akkor szükségünk lesz az osztályaira.

A vezérlőn Ctrl duplakattintásra vagy az

Osztályvarázsló

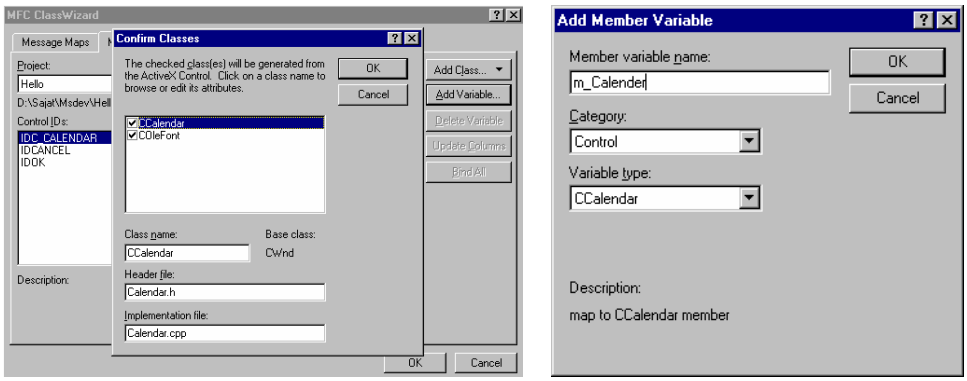
Member Variables fül

Osztály neve Kiválasztva a párbeszéd erőforrás osztálya

Control IDs: A vezérlő azonosítója

Add Variable gomb

'A vezérlő nincs beillesztve a projektbe' figyelmeztető üzenetre az OK gombot leütve a 3.22. ábra párbeszédablaka nyílik ki. Ha bejelöltük a szükséges osztályokat, OK gomb hatására azonnal a már ismerős Add Member Variable ablakkal találkozunk. Megadva a változó nevét, megtörtént az ActiveX adattag felvétele az osztályba.



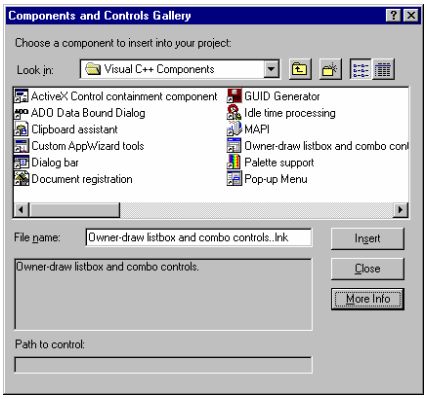
3.22. ábra ActiveX osztályok és változó generálása

Az ActiveX vezérlőhöz rendelt osztályok tagfüggvényei nem tesznek mást, mint hívják a vezérlő interfészében megadott függvényeket. Kódunkban az osztályok tagfüggvényeinek hívásával kezelhetjük a vezérlőobjektumot.

Megjegyzés:

ActiveX vezérlőket szúrhatunk be a Components and Controls Gallery segítségével is.

Projekt menüpont / Add to Projekt / Components And Controls



3.23. ábra Komponensgaléria

(Components and Controls Gallery)

A 4. gyakorlat dolgozza fel az ActiveX vezérlő csatolását és a vezérlő osztályból származtatott vezérlő esetét, mivel a megoldásokhoz az MFC áttekintése, illetve további, a 4. fejezetben tárgyalt elméleti ismeret szükséges.

3.5. Kérdések

1. Sorolja fel az alapvető vezérlőket, a hozzájuk tartozó MFC osztályokat, ismertesse a tulajdonságlap néhány tulajdonságát!
2. Milyen lehetőségeink vannak a vezérlők elrendezésére?
3. Hogyan csatolhatunk változókat a vezérlőkhöz, milyen kategóriák és típusok léteznek?
4. Mely függvények feladata a vezérlők és változók közti kapcsolattartás?
5. Milyen módszereket ismer a vezérlők kezelésére?
6. Ismertesse az üzenetablak hívási és megjelenítési lehetőségeit!
7. Ismertesse a párbeszédablak életciklusát, térjen ki az adattagok és a vezérlők közötti adatforgalomra!
8. Ismertesse a párbeszédalapú alkalmazásszerkezet osztálydiagramját!
9. Hogyan hozhatunk létre új szerkesztett párbeszédablakot alkalmazásunkba?
10. Milyen lépések egymás utáni végrehajtására van szükség ahhoz, hogy ActiveX vezérlőt csatolhassunk alkalmazásunkhoz?

3.6. Összefoglalás

- A **párbeszédfelület** az erőforrás-szerkesztővel szerkeszthető. **Vezérlő**ket húzhatunk a felületre, elrendezhetjük őket, beállíthatjuk tulajdonságaikat. Az osztályvarázsló segítségével üzenetkezelő metódusokat, adattagokat csatolhatunk hozzájuk. A változók Value vagy Control kategóriájúak lehetnek. Típusuk az adott vezérlőre jellemző listából választható. A vezérlő és a változó közti kapcsolatot a **DDX függvények** biztosítják. Futás közben az ***UpdateData*** metódus biztosítja az adatcserét közöttük. Az értékhatárok ellenőrzése a **DDV függvények** feladata.
- Az **előredefiniált párbeszédablak** vagy **üzenetablak** ablakosztály esetén a ***MessageBox***, más osztályok esetén az ***AfxMessageBox*** függvény segítségével hívható meg. Beállíthatjuk az ablak szövegét, címét, a rajta megjelenő gombokat, ikonokat.
- A **szerkesztett párbeszédablak** felületét, vezérlőit, a hozzájuk rendelt adattagokat és üzenetkezelő metódusokat a program írója határozhatja meg. Az adattagok értékét **DoModal** hívása előtt kell beállítanunk, így azok megjelennek a hozzájuk csatolt vezérlőkben. **OK gomb** választása esetén a vezérlőkből az adatok az adattagokba kerülnek, az ablak bezárása után értékük lekérdezhető. **Cancel gomb** választása esetén az ablak bezárásakor nincs adatcsere a vezérlők és az adattagok között.
- **Új erőforrás**t a Workspace / ResourceView / Resources / jobb egér / Insert s az erőforrás kiválasztásával szúrhatunk be. Az erőforrás szerkeszthető. Új párbeszédablakhoz az osztályvarázslóval osztályt rendelhetünk, majd felvehetjük a hozzárendelt osztály egy példányát. Az objektum a szerkesztett párbeszédablaknál leírtak szerint kezelhető.
- A Controls eszköztár vezérlőin kívül **ActiveX** vezérlőket is felhasználhatunk alkalmazásunk fejlesztésekor. A felmásolt .ocx fájlokat regisztrálni kell. A párbeszédfelületen jobb egér / **Insert ActiveX Control** menüpont hatására a kívánt vezérlő az ablakra kerül. Az osztályvarázsló létrehozza a kezeléséhez szükséges osztályokat. Úgy dolgozhatunk vele, mint a többi vezérlővel.

4. Az MFC felépítése

Ha alkalmazásaink fejlesztéséhez az MFC-t használjuk, akkor érdemes élnünk a Visual C++ fejlesztői felületének lehetőségeivel. Az MFC szorosan összefügg bármely anyagrésszel. Már eddig is éreztük ismeretének hiányát, azonban nem lehet róla beszélni addig, amíg alapvető tudással nem rendelkezünk néhány osztályáról. Ez a fejezet hivatkozik az eddig tanultakra, ismertet új osztályokat, és óhatatlanul megemlít olyan osztályokat is, amelyek a könyv további részében kerülnek feldolgozásra. Minél később foglalkozunk az MFC-vel, annál jobban megértjük szerkezetét s a benne szereplő osztályok felelősségeit. Viszont addig folyamatosan érezzük a hiányt, használunk valamit, és nem is igazán tudjuk, hogy mi az. Tehát csak kompromisszum árán ismerhetjük meg.

4.1. Az MFC áttekintése

Az **MFC** a **Microsoft Foundation Classes** rövidítése, a Visual C++ által használt **osztálykönyvtár**.

- Az MFC **magasabb szintű Windows absztrakció**t biztosít a programozás során, mint az API programozás. Ezáltal megkímél bennünket a gyakran előforduló részletkérdések végiggondolásától, programozásától. Ugyanakkor mi a kódban bármit önállóan is megvalósíthatunk.
- **Gyorsabban megtanulható** a hagyományos Windows programozásnál. Ez következik már az előzőekből, hisz nem kell azonnal a program futásának

összes részletével tisztában lennünk. A magasabb absztrakciós szint közelebb áll a hétköznapi tapasztalatokhoz.

- Az alkalmazás váza **gyorsabban fejleszthető**. Rendelkezésünkre bocsát alkalmazásszerkezeti struktúrákat, melyekben elkészített absztrakt osztályok absztrakt metódusait (OOP 5.4.6. szakasz) kell csak megvalósítanunk ahhoz, hogy a saját alkalmazásunk működjön. Természetesen az osztálykönyvtár osztályai a további fejlesztést is segítik, így gyorsítják.
- **Objektumorientált technikákat biztosít** a kód írásakor. A C++ a Visual C++ objektumorientált nyelve. Az OOP kötet (Lásd bevezető!) e technikák ismertetése céljából készült.

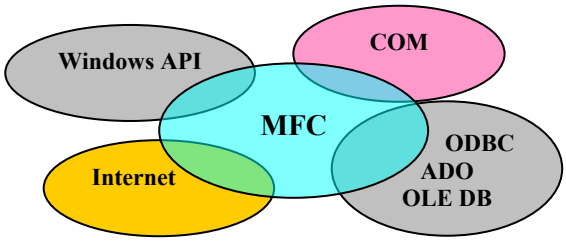
4.1.1. API: Application Programming Interface

Rutingyűjtemény, melyet a gép operációs rendszere alacsonyszintű szolgáltatások eléréséhez biztosít. Ez a Windows operációs rendszer interfésze.

A Win32 API támogatja a különböző Windows operációs rendszereken kívül több platform használatát is, pl. UNIX, Macintosh.

4.1.2. Az MFC mint osztálykönyvtár

Az MFC magja az a több mint 130 osztály, melyek kényelmes C++ tagfüggvényeket biztosítanak a Windows API jelentős részének eléréséhez. Jó példa erre a CWnd osztály, mely egységbezárja (encapsulates) (OOP 1.3.) "becsomagolja (wraps)" a HWND Windows ablakkezelőt, mint adattagot, és metódusain keresztül elérhetjük az ablakokat vezérlő Windows API-kat (4.8. szakasz). Az MFC osztályokon keresztül hozzáférünk az operációs rendszer egyes elemeihez, tartalmaznak COM technológiát magukba foglaló kódrészleteket. Kényelmesen kezelhetjük a különböző adatbázis-eléréseket, valamint Internetes kapcsolatokat. Az MFC ezen kívül tartalmaz alkalmazásszerkezeti és egyéb osztályokat is:



4.1. ábra Az MFC kapcsolatai

4.1.3. Az MFC mint framework

Az MFC tartalmaz egy másik Windows API réteget is. Ez egy állandóan dolgozó alkalmazás keretrendszer (framework), mely biztosítja a szokásos Windows interfészek – mint például eszközsorok, státusz sorok, nyomtatás, adatbázis-kezelés és ActiveX – támogatását.

A háttérben futtatja a WinMain függvényt, mely meghívja az alkalmazásosztály metódusait. A framework biztosítja az üzenetek továbbítását a kezelőfüggvényekhez, melyek lehetnek alapértelmezettek, vagy a programozó által megírtak (2.3, 2.6. szakasz).

Amikor az alkalmazásvarázslóval létrehozzuk az alkalmazás vázát (skeleton files), már egy a keretrendszer által biztosított működő programot készítünk. Az üzenetkezelők és a virtuális metódusok felülírásával a programozó csak a működését befolyásolja. A "Don't call us, we call you!" ("Ne hívj minket, mi hívunk téged!") elvnek megfelelően a framework hívja az MFC osztályok virtuális metódusait, melyek többnyire a futásidejű kötésnek (runtime binding) köszönhetően a programozó által fölülírt függvények (polimorfizmus). (OOP 5.3.) Ha például egy általunk létrehozott ablakosztályban (mely a CWnd utódosztálya) a ***PreCreateWindow()*** tagfüggvényt felülírjuk – hogy módosítsuk az ablak stílusát – a framework a mi PreCreateWindow függvényünket hívja az ablak létrehozása során.

4.2. Információk az MFC osztályokról

Az osztályok ismertetését a 6.0 verziótól az **MSDN** (Microsoft Developer Network) biztosítja, ez is jelképezi a Visual Studion belül az osztályok egységes kezelését. Régebbi verziók a telepítő CD-n tartalmazták a súgót.

Az MSDN a Start menüből is elindítható, de a Visual C++ alkalmazás Help menüjéből önálló alkalmazásként indul. Az MFC szerkezetét bemutató ábrát több úton is elérhetjük, itt következik az egyik:

Contents fül

Visual Studio dokumentation

Visual C++ dokumentation

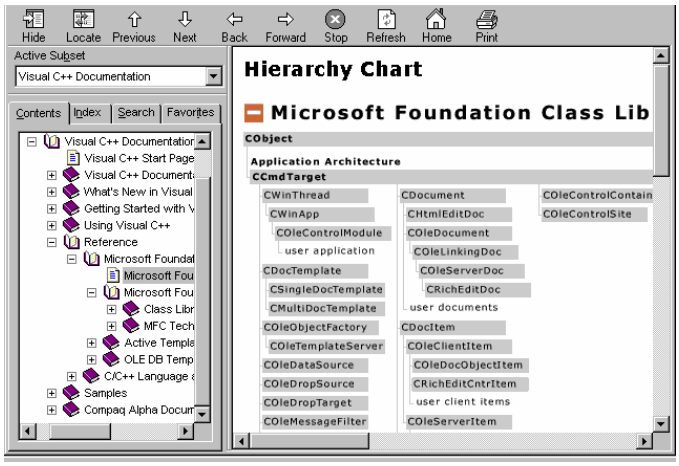
Reference

Microsoft Foundation Class Library and Templates könyv

Microsoft Foundation Class Library and Templates oldalán a

Microsoft Foundation Class Library (MFC) mappa

Hierarchy Chart mappa (4.2. ábra)



4.2. ábra Az MFC osztályszerkezete

E fejezet kiegészítéseként az olvasó figyelmébe ajánlom ugyanitt, a Hierarchy Chart helyett, az About the MFC Classes című ismertetését az MFC-nek.

Az osztályok leírását ABC-sorrendben tartalmazza a Class Library Reference (a Hierarchy Chart innét is elérhető):

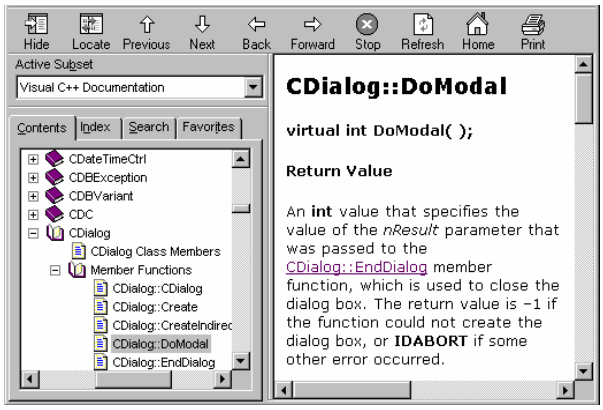
Reference

Microsoft Foundation Class Lybrary and Templates könyv

Microsoft Foundation Class Library

Class Library Reference (4.3. ábra)

Minden osztályról találunk egy általános leírást, benne az öröklési láncot az osztály őseiről. A **Member Functions** könyv alatt ábécérendben a tagfüggvények felsorolását láthatjuk. Ahogy az általánosan megszokott, nem szerepelnek benne az osztály által örökölt tagok. Azok leírását az ősosztályok valamelyikében kell keresnünk. A 4.3. ábra a CDialog osztály DoModal tagfüggvényét mutatja. Az oldalakon hiperlinkek segítségével juthatunk további információkhoz. Időnként előnyös lehet egy függvény keresésekor az ábécérend, máskor inkább a kategóriák szerinti csoportosítás a megfelelő. Azok az osztályok, melyek leírásában sok tagfüggvény található, biztosítják a tagfüggvények feladataik szerinti csoportosítását. Ezt az Osztálynév + Class Members megnyitásával érhetjük el (4.3. ábra). Nézzük meg például a CWnd osztály CWnd Class Members csoportjait!



4.3. ábra Az osztályok és tagfüggvényeik az MSDN-ben

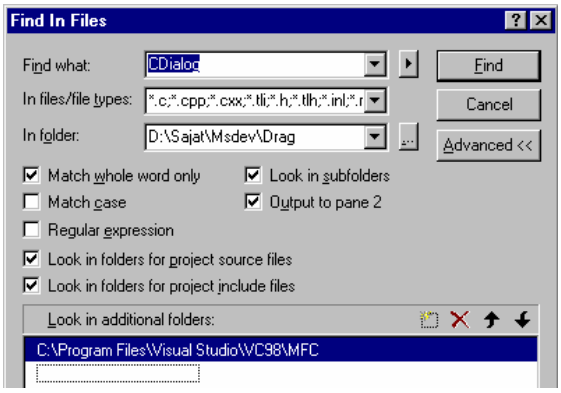
A súgó további használatáról már volt szó az 1.2.4. szakaszban.

Az MFC-vel együtt rendelkezésünkre áll az összes forráskód is.

- **A fejléc fájlok (.h): MFC \ Include könyvtárban**
- **Az implementációs fájlok (.cpp): MFC \Src (source) könyvtárban** találhatók.

A szükséges forráskódhoz legkönnyebben a Find in Files eszköz segítségével juthatunk.

A 4.4. ábrán látható ablakban beállíthatjuk, hogy csak teljes szóra keresünk-e, a projekt összes forrásfájljában, vagy include fájljában keresünk-e, illetve további foldereket állíthatunk be.



4.4. ábra A Find in Files ablak

Az osztályok forráskódja az osztály tagjait feladataik szerint csoportosítva tárolja. A csoportok az áttekinthetőséget növelik, valójában csak kommentek (megjegyzések). E csoportok száma és elnevezése osztályonként változhat. A csoportok határai nem élesek, bizonyos függvények tartozhatnának az egyik vagy a másik csoportba is. Vizsgáljuk meg pl. az afxwin.h osztályainak csoportjait! Használjuk a Find  parancsot a kereséshez!

Itt következik néhány gyakran előforduló csoport ismertetése:

//Constructors: C++ konstruktorok (OOP 4. fejezet) és egyéb inicializálást megvalósító tagfüggvények. Pl. CWnd::Create(). Általában public-ok.

//Attributes: Tulajdonságok (properties). Általában adattagok vagy Get / Set függvények, melyekkel private adattagok értékét kérdezhetjük le, vagy módosíthatjuk. (Egységbezárás, OOP 1.3. szakasz.) Általában public-ok. Nagy osztályoknál, pl. CDC vagy CWnd olyan sok tagja lehet, hogy több megjegyzéssel továbbosztották e csoportot.

//Operations: Valamilyen műveleteket tartalmaz. Általában nem konstans, nyilvános függvények. Többnyire ide tartoznak az átdefiniált operátorok (OOP 7. fejezet). Sok helyütt továbbosztódik ez a kategória is.

//Overridables: Virtuális függvények (OOP 5. fejezet), az utódosztályokban általában felülírjuk őket. Gyakran On-nal kezdődnek, hisz ide tartoznak az üzenetkezelő metódusok is. Általában protected vagy public elérésűek, és absztrakt függvények (pure virtal) is lehetnek.

//Implementation: A megvalósításhoz tarozó függvények. Ide tartoznak például a felülírt virtuális destruktorok vagy a PreTranslateMessage.

4.3. A CObject osztály

A CObject osztály a legtöbb MFC osztály őse. Úgynevezett alaposztály. Leírása az afx.h-ban található.

Szolgáltatásai:

- Támogatja a **szerializáció**t (**serialization**). A ***Serialize()*** tagfüggvénye segítségével, ha az objektum szerializálható (***IsSerializable()***) fájlba mentését, olvasását teszi lehetővé. A szerializációról az 6.2. fejezetben olvashatunk részletesebben.
- A futásidejű osztályinformációk elérését támogatja. A ***GetRuntimeClass()*** tagfüggvény az objektum CRuntimeClass (lásd később) struktúrájával tér vissza. Az ***IsKindOf()*** tagfüggvény igazat ad vissza, ha az objektum típusa a paraméterként megadott osztály vagy annak utódosztálya. (Nem támogatja a többszörös és a virtuális öröklést.)

- Támogatja a konténerosztályok (collections) használatát. Az összes, az objektumok összegyűjtésére használatos osztály a CObject-ből származik. (Lásd Hierarchy Chart, Arrays, Lists, Maps!)

Megjegyzés:

A CTypedPtr kezdetű konténerosztályok első paraméterében megadható ősosztály is a CObject utóda.

A **CRuntimeClass** osztálynak nincs ősosztálya. Feladata futásidőben információk tárolása az objektum osztályáról, ősosztályáról. Néhány adattagja:

m_lpszClassName: Az osztály neve. (ASCII, nullterminated)

m_nObjectSize: Az objektum mérete bájtokban. Ha tartalmaz mutató adattagot, akkor a mutatott memória mérete nincs benne.

m_wSchema: Az objektum sémaszáma. Értéke -1, ha az objektum nem támogatja a szerizalizációt. (Lásd szerializáció, 6.2. szakasz!)

Lehetőséget biztosít az ősosztály CRuntimeClass struktúrájának elérésére.

Ahhoz, hogy objektumunkra elérjük a futásidejű információkat, osztályának a CObject utódának kell lennie, és a deklarációnak tartalmaznia kell a **DECLARE_DYNAMIC**, **DECLARE_DYNCREATE**, **DECLARE_SERIAL** makróhívások valamelyikét, melyek engedélyezik objektumok dinamikus létrehozását, pl. lemezről olvasásnál a szerializáció alatt. Az implementációban pedig az **IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNCREATE**, **IMPLEMENT_SERIAL** makróhívások valamelyikét kell megadnunk.

A **RUNTIME_CLASS** makró egy mutatót ad vissza a CRuntimeClass objektumra.

```
CRuntimeClass* pClass = RUNTIME_CLASS( CObject );
```

A 4.2.6. (DragList) feladat megoldásában a GetListBoxFromPoint() függvényben láthatjuk, hogyan ellenőrizzük futás közben egy mutató osztályát. A RUNTIME_CLASS makró használatával a 6.1 és 6.2. gyakorlaton a Dokumentumsablon-objektum létrehozásánál 6.4.2. szakaszban az új nézetosztály létrehozásánál találkozunk.

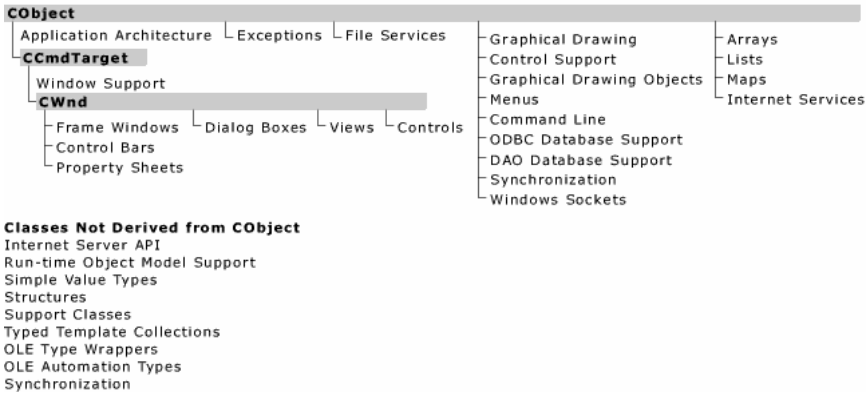
4.4. A CObject főbb utódosztályai

Az előző szakasz alapján, ha létrehozunk egy osztályt, melyben szerializációt szeretnénk megvalósítani, vagy a futásidejű információkat el akarjuk érni, akkor származtassuk azt a CObject-ből!

A CObject főbb utódosztályai (4.5. ábra) a:

4.4.1. Az alkalmazások főbb alaposztályai

- Parancsértelmezők (**CCmdTarget**): A felhasználó üzeneteit kezelik.
- Alkalmazásosztályok (**CWinApp**): Az alkalmazás vázát biztosítják.
- Dokumentumosztályok (**CDocument**): Dokument / View architektúra esetén az adatok tárolásáért felelősek.
- Ablakok (**CWnd**): A megjelenítésért felelősek. Képesek kezelni az alapvető Windows üzeneteket.



4.5. ábra A CObject utódosztályai (MSDN)

4.4.2. A kivételkezelést támogató osztályok

- A **CException** és utódosztályai is a CObject-ből származnak. Függvények visszatérési értékeként, vagy a visszatérés után is elérhető paraméterként adhatók át. Segítségükkel hibás végrehajtás esetén adatokat kaphatunk a bekövetkezett hibáról, így gondoskodhatunk kezeléséről (exeption handling), (OOP 9. fejezet).

4.4.3. A fájlkezelést támogató osztályok

- Ősosztályuk a **CFile** közvetlenül a CObject utódosztálya. Biztosítja a pufferezés nélküli, közvetlen, bináris lemezműveleteket. Virtuális függvényei lehetővé teszik a programozó számára a különböző fájl osztályok (utódosztályai) egységes kezelését. A polimorfizmusnak (OOP 5.3.2.) köszönhetően hasonlóan írhatunk a memóriába, mint a lemezre (**CMemFile**).

  - ***Open()***: Kivételkezelés segítségével biztonságosan nyithatjuk meg az első paraméterében megadott útvonalon elérhető fájlt, a második paraméterben megadott módon.
  - ***Close()***: A fájl lezárása után nem lehet írni és olvasni. A destruktor lezárja a fájlt, ha elfelejtenénk.

- ***Read()***:Beolvassa az első paraméterben megadott pufferbe a második paraméterben megadott számú bájtot. ***CStdioFile::ReadString()***: beolvassa az első paraméter által mutatott pufferbe a következő sort. Második paramétere eggyel több mint a maximális karakterek száma. Utolsóként egy '\0' karaktert helyez el.
- ***Write()***: Kiírja az első paraméterben megadott pufferből a második paraméterben megadott számú bájtot. ***CStdioFile::WriteString()*** kiírja a paraméter által mutatott sort a fájlba. A '\0' karakter nem kerül kiírásra.
- ***Flush()***: A pufferben maradó adatokat írja ki a fájlba. (Kiüríti a puffert.)

A **CArchive** osztály segítségével biztosítja a **szerializációt** az MFC objektumok számára. A CArchive lehetővé teszi, hogy az adatokat a szerkezetükkel együtt megőrizzük. A tárolás általában fájlban történik. A mentés és betöltés folyamatát szerializációnak nevezzük. (Részletek az 6.2. szakaszban.) A CArchive objektum létrehozása előtt, rendelkeznünk kell egy a műveletnek megfelelően megnyitott CFile objektummal. Ez a konstruktor paramétere lesz. CArchive segítségével nem csak egyszerű típusokat, hanem CObject-ből származó osztályok objektumait is tárolhatjuk, és kényelmesen írhatjuk olvashatjuk a háttértárról.

4.4.4. A konténerosztályok

- Collections. Az MFC-hez tartozó változtatható méretű tömbök (Array), dinamikus listák (List), és szótárak (Map) osztályait gyűjti össze.

  - **Egyszerű konténerosztályok**. Adott típusú elemeket tartalmazó konténerek. Természetesen ez nem azt jelenti, hogy csak egy adott típusú objektum tárolására alkalmasak, hiszen a **CObArray** CObject mutatókat tárol, de mivel ősosztály mutatója utódobjektumra is mutathat, így az MFC legtöbb osztályát tárolhatjuk ebbe a tömbbe. Természetesen amikor egy tömb elemére az utódosztály metódusát hívjuk, explicit típuskonverzióra van szükség. Hasonló módon kezelhető a **CPtrList** osztály.
  - **Egyszerű konténer sablonosztályok**. **CArray**, **CList**, **CMap**. A deklaráció mindhárom esetben a következő minta alapján történik:

    template< class TYPE, class ARG_TYPE> class CList: public CObject

    A **TYPE** a lista elemtípusát, az **ARG_TYPE** a függvények paramétertípusát adja meg. A CMap sablon négy paramétert tartalmaz, kettőt a kulcsra és kettőt a szótár elemeire vonatkozóan.

    Ha egy CMyClass típusú elemekből álló dinamikus listát akarunk készíteni, azt a következőképpen tehetjük:

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

A listára a CList tagfüggvényei CMyClass& argumentumokkal hívhatók meg. Pl.

```
CMyClass myObject;
myList.AddTail(myObject);
```

- További lehetőségeket rejtenek a CTypedPtr kezdetű osztályok, **CTypedPtrArray**, **CTypedPtrList** és a **CTypedPtrMap**. deklarációjuk:

  template< class BASE_CLASS, class TYPE > class CTypedPtrArray : public BASE_CLASS

  Láthatóan az első paraméter az ősosztályt adja, mely kötelezően CObArray, vagy CPtrArray típusú, míg a második paraméter a tömbelemek típusát határozza meg. (Értelemszerűen, a CTypedPtrMap három paraméterű.)

  Ha egy CMyClass típusú elemekből álló mutatótömböt szeretnénk létrehozni, azt a következőképpen tehetjük:

  ```
  CTypedPtrArray<CObArray, CMyCalss> myArray;
  ```

  A tömbre a CObArray metódusai hívhatók:

  ```
  CMyClass* pMyObject=new(CMyCLass);
  myArray.Add(pMyObject);
  ```

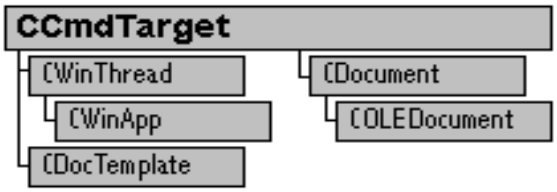
A konténerosztályok mindegyike bőségesen biztosít metódusokat a szokásos műveletek elvégzésére. **A sablonosztályokat az OOP könyv 8. fejezete részletesen tárgyalja.**

A **nem CObjectből származó osztályok** közül az előző szakaszban foglalkoztunk a **CRuntimeClass** osztállyal, mely a Run-time Object Model Support kategória tagja. Ebbe a kategóriába tartozik **CArchive** osztály is. Már találkoztunk a vezérlők és adattagok összekapcsolásánál a DoDataExchange paramétereként a **CDataExchange** osztállyal, mely a Support Classes kategória eleme, és gyakran használjuk a Simple Value Types kategória osztályának objektumait. Pl. **CString**, **CPoint**...

A kategóriák elemeit a Hierarchy Chart mutatja. (4.2. ábra)

4.5. Az alkalmazásszerkezeti osztályok

Az alkalmazás indításakor jönnek létre. Meghatározzák az alkalmazás szerkezetét, viselkedését. Megvalósítják az üzenetek feldolgozásához szükséges műveleteket. Biztosítják, az alkalmazás futását, nekünk csak az általuk meghívott virtuális függvényeket kell felüldefiniálni ahhoz, hogy a program az elvárt módon működjön.



4.6. ábra Az alkalmazásszerkezeti osztályok

4.5.1. A CCmdTarget osztály

- **A CObject utódosztálya**.
- **Az MFC üzenettérkép** (message map) **szerkezetének alaposztálya**.
- **Utódosztályai** (CWnd, CDialog, CWinApp, CDocument...) **hozzáférnek a parancsüzenetekhez**, vagyis bennük a parancsüzenetekhez kezelő metódusokat csatolhatunk. Emlékeztetőül: parancsüzenetek generálódnak, ha a felhasználó egy parancsot választ a menüsorból, az eszközsor gombjaiból, vagy leüt egy gyorsgombot (2.6.3. szakasz.).

4.5.2. A CWinThread osztály

Segítségével kényelmesen kezelhetjük az alkalmazás szálait. A CWinApp osztály ősosztálya. Tagfüggvényei lehetővé teszik, hogy létrehozzanak új szálakat (***CreateThread***), manipuláljanak szálakat (***PostThreadMessage***, ***SetThreadPriority***).

Tartalmazza az ***InitInstance, Run, ExitInstance, OnIdle*** függvényeket, melyeket eddig a **CWinApp** osztálynak tulajdonítottunk.

4.6. Többszálú alkalmazások

A Windows lehetővé teszi alkalmazások párhuzamos futtatását. Mivel a számítógépben általában egy processzor van, de mindenképp kevesebb mint a futó alkalmazások száma, ezért ezek az alkalmazások valójában nem párhuzamosan futnak, hanem 32-bites Windows úgy nevezett **preemptív időosztásos módszerrel megosztja** közöttük **a processzoridőt**. A Windows a processzoridőt **a szálak között** osztja meg. Az időosztás azt jelenti, hogy a futó programok szálai közül az első kap egy **időszelet**et (**time slice**) amíg futhat, majd a következő időszeletben egy következő szálé a processzor használatának lehetősége. A preemptív mód azt jelenti, hogy **az operációs rendszer feladata a processzor(ok) idejének felosztása a szálak között**. Ennek következtében, ha egy szál végtelen ciklusba vagy definiálatlan állapotba kerül, az a rendszerben futó egyéb szálak futására nem lesz

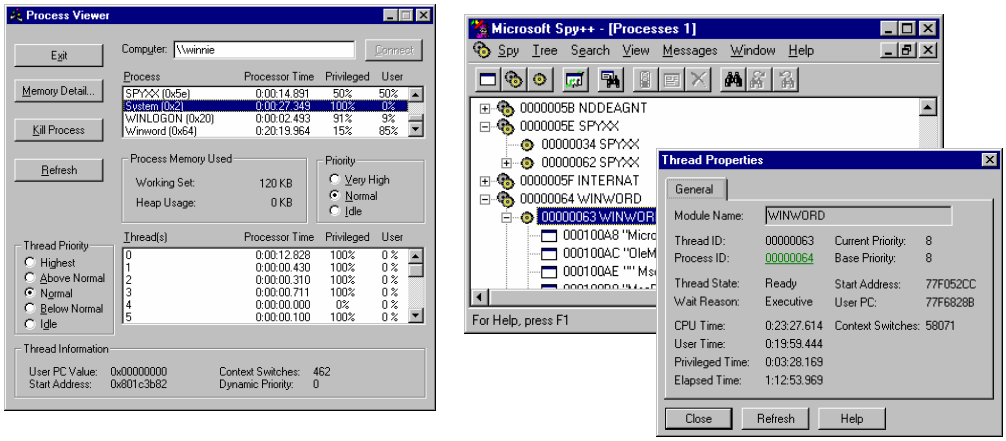
hatással. Az időszeletek kiosztásánál az operációs rendszer figyelembe veszi a szál prioritását.

Megjegyzés:

> A Windows 3.1 úgynevezett **kooperatív** többfeladatos működést valósít meg, ami azt jelenti, hogy az idő elosztása a programok együttműködésének (kooperációjának) eredménye. Lényege, hogy az éppen futó program feladata, hogy felszabadítsa a processzort, ha már nincsen rá szüksége. Ezért a végtelen ciklusba kerülő programok gyakran az egész rendszer összeomlását eredményezik. (Kis, 1999, 158. old.)

Kétféle programot különböztetünk meg, az alkalmazást és a processt (folyamatot). **Alkalmazásnak nevezzük a felhasználó által indított programokat**, míg **process minden a rendszerben futó program**, akár a felhasználó indította, akár az operációs rendszer. Tehát **az alkalmazások is folyamatok, vagy több processből álló rendszerek.**

Az alkalmazások (application) és processek megfigyelésére a legegyszerűbb a **Task Manager** használata (Ctrl+Shift+Esc). A Visual Studio a **Process Viewer** eszközt kínálja, melyben a szálak és a szálak prioritása is megfigyelhető – sőt módosítható – mutatja a process memória felhasználását is. A már megismert **Spy++** segítségével is megfigyelhetjük az éppen futó processeket, azok szálait, prioritásukat és a szálakhoz tartozó ablakokat (4.7. ábra).



4.7. ábra A Process Viewer és a Spy++ mutatja a processek szálait

Alkalmazásunkban előfordulhatnak olyan tevékenységek, melyek rendkívül időigényesek (részletes számítások, grafikus kép megjelenítése, lemezműveletek pl.

másolás...), s ezek teljes végrehajtása nem feltétlen szükséges ahhoz, hogy a felhasználó folytassa munkáját. Az is előfordulhat, hogy engedélyezni akarjuk a felhasználó közbelépését, pl. a hosszúra nyúló tevékenység leállítását. Ilyenkor célszerű ezt a műveletet egy másik szálban (thread) elhelyezni, mely párhuzamosan fut a programunk által használt elsődleges szállal.

Egy új szál elindítása viszonylag gyors művelet, és nem igényel sok memóriát. Az egy folyamaton (process) belüli szálaknak közös az adatszegmensük és a kódszegmensük, de saját veremszegmensük (stack) van. Így könnyen megoszthatjuk a szálak között az adatokat.

Alapvetően kétféle szálat különböztetünk meg:

- A **WorkerThread** (dolgozó szál): A háttérben dolgozik, nem kommunikál a felhasználóval, nem kezel user interface üzeneteket.
- **UserInterfaceThread**: Felhasználói felülettel rendelkező szál, mely lehetővé teszi a felhasználó beavatkozását a szál végrehajtásába.

Új dolgozó szál elindításához először létre kell hoznunk azt a függvényt amelyet a szál futtat. A függvény kötelezően LPVOID paraméterű és UINT visszatérési értékű:

```
UINT SzálFüggvény(LPVOID pParam)
{...
 return 0;
}
```

A pParam paraméter egy 32 bites mutató, tehát a látszólagos egy paraméterre vonatkozó kötöttség mögött nagyon sokféle változó címét átadhatjuk.

Megjegyzés:

Azok a Windows objektumok, melyeknek van handle adattagjuk, paraméterként csak a handle adattagot tudják átadni a szálfüggvényeknek. Pl. ablakok az m_hWnd adattagot. (Lásd 4.8.3. szakasz!)

Most már meghívhatjuk az ***AfxBeginThread*** függvényt az új szál elindításához:

CWinThread* AfxBeginThread(AFX_THREADPROC *pfnThreadProc*, **LPVOID** *pParam*, **int** *nPriority* = **THREAD_PRIORITY_NORMAL, UINT** *nStackSize* = **0, DWORD** *dwCreateFlags* = **0, LPSECURITY_ATTRIBUTES** *lpSecurityAttrs* = **NULL);**

```
//...
CWinThread* pMyThread;
pMyThread = AfxBeginThread( Szálfüggvény, &paraméter);
//...
```

A szál megállítása: a szálfüggvény visszatérésével vagy ***AfxEndThread***(UINT nExitCode) hívással történhet. Ez utóbbit beágyazott (a szálfüggvény által meghívott) függvényekből szoktuk alkalmazni.

Új felhasználói felülettel is rendelkező szálat az AfxBeginThread átdefiniált alakjával hozhatunk létre:

CWinThread* AfxBeginThread(CRuntimeClass* *pThreadClass*, **int** *nPriority* = **THREAD_PRIORITY_NORMAL, UINT** *nStackSize* = **0, DWORD** *dwCreateFlags* = **0, LPSECURITY_ATTRIBUTES** *lpSecurityAttrs* = **NULL);**

```
class CMyThread public CWinThread
{...}
//...
AfxBeginThread(RUNTIME_CLASS(CMyThread));
//...
```

A CMyThread osztályt legegyszerűbben az osztályvarázslóval készíthetjük el. Az InitInstance tagfüggvényében hozhatjuk létre a szálhoz a főablakot. Itt is igaz, hogy ha az InitInstance visszatérési értéke FALSE, nem indul el a Run().

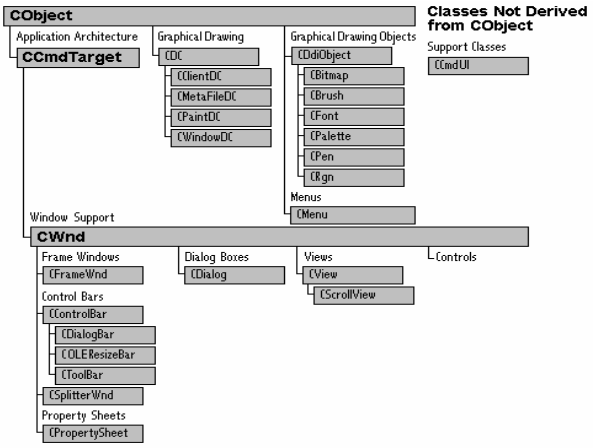
Többszálú alkalmazásoknál komoly problémát jelenthet a szálak működésének szinkronizálása, melyhez az MFC több szinkronizációs osztályt biztosít:

Pl. CEvent, CCriticalSection, CMutex, CSemafore, CSingleLock, CMultipleLock.

4.7. A megjelenítést támogató osztályok

A képernyőn Windows alkalmazások esetén a megjelenítés elsődleges eszközei az ablakok, melyek mindegyike a CWnd alaposztály utóda. Az ablakokon kívül további osztályok is segítik a kommunikációt a felhasználóval. (4.8. ábra)

A meghajtófüggetlen megjelenítést a **CDC** (Device Contents) osztályok és utódaik segítik. A konkrét rajzolást pedig a **CGdiObject** osztály utódosztályai. Mindegyikről részletesen lesz szó a 5. fejezetben. A **CCmdUI** osztállyal az 6.3. fejezetben találkozunk, a parancsüzeneteket küldő felhasználói interfészek megjelenítésekor hívott függvény paramétereként.



4.8. ábra A megjelenítést támogató osztályok

4.8. Az ablakok

A Windows sikerét, amint azt a neve is mutatja, a képernyőn látható ablakok biztosítják. Ha mi magunk megpróbálunk ablakfelületet készíteni, rá kell ébrednünk, hogy az egyik legnagyobb problémát az egymást eltakaró ablakok képeinek frissítése jelenti. Ezzel a problémával nem nagyon kell közvetlenül foglalkoznunk a kód írásakor, azonban a súgó többször utal rá, és a működését megértését is segíti, ha tudunk a megvalósításról.

Az egymás fölött elhelyezkedő ablakok sorrendjét az úgynevezett **z-sorrend**ből (z order) állapítja meg az operációs rendszer. Minden ablaknak van egy helye az ablakveremben. Ezt úgy tároljuk, mintha lenne egy 'z' koordinátája. A z-sorrend tetején levő ablak minden más ablakot eltakar, az alján levőt az összes többi ablak takarja.

Szülő - gyermekablak (**parent - child window**) kapcsolat:

- A gyermekablak mindig a szülőablak fölött van z-sorrendben.
- A gyermekablakot nem lehet a szülőablak területén kívülre mozgatni.
- A szülőablak műveletei; mozgatás, minimalizálás... automatikusan hatnak a gyermekablakra is.
- A Windows operációs rendszer a szülőablakon keresztül kezeli a gyermekablakot.

Szülő - gyermekablak kapcsolat van a Word és a benne megnyitott dokumentumok nézetei, a Visual C++ és a megnyitott forrásfájlok ablakai között.

Tulajdonos - tulajdon ablak (owner - owned window) kapcsolat:

- A tulajdon ablak mindig a tulajdonos fölött van z-sorrendben.
- A tulajdon ablak automatikusan megszűnik, ha a tulajdonosa megszűnik.
- A tulajdon ablak rejtve van, ha tulajdonosát minimalizáljuk, és megjelenik, ha tulajdonosát visszaállítjuk.

Tulajdonos - tulajdon ablak kapcsolat van az alkalmazás főablaka és az üzenetablakok között.

4.9. ábra Szülő - gyermek, tulajdonos - tulajdon ablak

Az **asztal** (**desktop**) takarja a teljes képernyőt. Az operációs rendszer számára ő az elsődleges ablak. Minden alkalmazásnak, ha használ ablakot, kell tartalmaznia egy főablakot. Erre mutat az m_pMainWnd mutató az alkalmazásosztályban. A főablak közvetlen gyermek ablaka az asztalnak. A főablakok az operációs rendszer feladatütemezésében jelennek meg.

4.8.1. A CWnd osztály

Az MFC ablakok alaposztálya. Definiálja az ablak alapfüggvényeit. Alapértelmezett válaszokat ad az ablaküzenetekre.

Néhány utódosztálya:

- CFrameWnd
- CControlBar

  Ősosztálya az eszközsornak (CToolBar), státuszsornak (CStatusBar)...
- CView

- CDialog
- CButton, CListBox, CScrollBar... az összes vezérlő.

A CWnd osztály több száz tagfüggvénye közül itt felsorolok néhányat, melyeket a könyvben is használunk azért, hogy képet kapjunk az osztályról. Nézzük meg a CWnd osztály tagfüggvényeit az MSDN-ben!

```
Create( )
PreCreateWindow()
EnableWindow()
SetFocus()
GetDlgItem()
UpdateData()
GetDC()
Invalidate()
ShowWindow()
MessageBox()
SetTimer()
KillTimer()
SendMessage()
PostMessage()
WindowProc()
DoDataExchange()
OnPaint()
OnLButtonDown()
```

4.8.2. Az időzítő

Amint azt az előző felsorolásból láthattuk, az ablakosztály felelőssége az időzítő kezelése. Az időzítőt a ***SetTimer*** tagfüggvény inicializálja.

SetTimer:

```
iInstallResult = SetTimer(1,500,NULL);
```

- **Visszatérési értéke**:

  Sikeres indítás esetén az időzítő azonosítója, egyébként 0.

- **Paraméterei**:
  - Az időzítő azonosítója. (Nem 0 UINT.)
  - A **WM_TIMER** esemény hívásásnak gyakorisága millisecundumban.
  - Függvényparaméter címe, mely a WM_TIMER esemény bekövetkezésekor hívódik. Ha értéke NULL, akkor az alkalmazás CWnd ablakának üzenetsorába kerül a kezelése. ON_WM_TIMER()

Az időzítő megszüntetése a CWnd osztály ***KillTimer*** tagfüggvényével történik.

KillTimer:

```
KillTimer(1);
```

- **Visszatérési értéke**:

  Logikai, 0, ha nem találja a paraméterben megadott azonosítójú időzítőt.

- **Paramétere**:

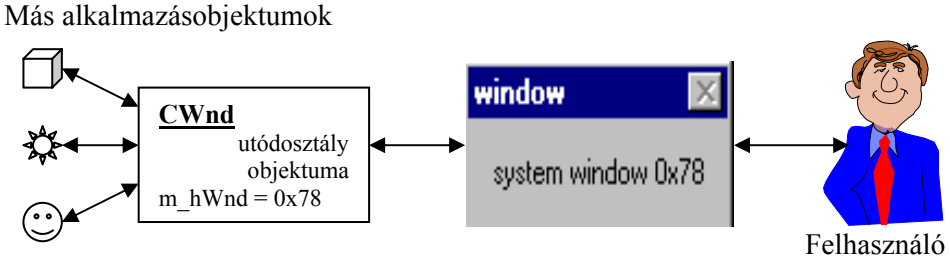
  Az időzítő azonosítója. (Nem 0, UINT.)

Az időzítő az osztályvarázsló segítségével az ablakosztály WM_TIMER üzenetéhez csatolt ***OnTimer*** metódust hívja meg a megadott időközönként. Több időzítő esetén az OnTimer(nIDEvent) azonosítója adja meg, melyik időzítő küldte az üzenetet. Az azonosítót tesztelve egyszerre több időzítőt is kezelhetünk.


A 5.3. gyakorlat és a 6.6.4. feladat leírásában találkozhatunk az időzítő használatával.

4.8.3. CWnd és HWND kapcsolata

A CWnd osztály objektuma nem azonos a Windows operációs rendszer objektumával, a rendszerablakkal. Az ablakobjektum a CWnd osztály (vagy egy utódosztályának) objektuma, melyet a konstruktorral hozunk létre. A Windows rendszerablak, egy tudatlan kezelője egy belső Windows adatstruktúrának, mely megfelel az ablaknak, és felhasználja a rendszer erőforrásait amikor megjelenik. A programozási nyelvek a Windows API-n keresztül érik el az operációs rendszerablakot. Az MFC a CWnd osztályban biztosítja a hozzáférést. **A rendszerablak azonosítója az ablakkezelő**, **a HWND** (window handle). Az ablakobjektum létrehozása után a CWnd osztály ***Create*** tagfüggvényének végrehajtása során jön létre a HWND. **Az ablakkezelőt az ablakosztály m_hWnd adattagjában tároljuk.** (4.10. ábra) Az m_hWnd egy mutató arra az adatstruktúrára, mely a rendszerablakot leírja.



4.10. ábra Az ablakobjektum és az ablakkezelő

Az ablak létrehozása két lépésben történik.

- ***CWnd::CWnd()***

  Nem hoz létre rendszerablakot.

- ***CWnd::Create()***

  Létrehozza a rendszerablakot. (2.1. gyakorlat)

  Kezelője a **CWnd::m_hWnd** adattagjába kerül.

  A kezelőtáblában (handle map) elhelyez egy (HWND, CWnd*) párt.

Az ablak megszüntetése is két lépésben történik.

- ***CWnd::DestroyWindow()***

  Törli a rendszerablakot és a kezelőtábla-bejegyzést, kinullázza az m_hWnd adattagot.

  Nem törli a C++ ablakobjektumot!

  Ha az ablak szülőablaka más ablakoknak, a gyermekablakokra automatikusan meghívódik a DestroyWindow.

- ***CWnd::~CWnd()***

  Ha a rendszerablak csatolva van (m_hWnd!=NULL), hívja a CWnd::DestroyWindow-t.

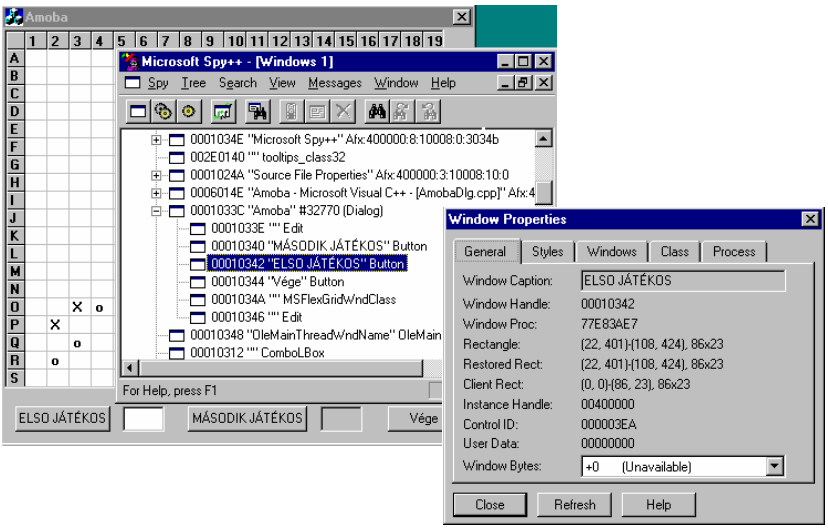
  Megszünteti a C++ ablakobjektumot. (QA, 1997.)

Az ablakok hierarchiája, tulajdonságai és az ablakkezelő futás alatt jól vizsgálható a **Spy++** eszközzel, mely önállóan is indítható, de a Visual C++ Tools menüjéből is. A megfelelő ablakot kiválasztva a jobb egérgomb menüjéből a Properties hatására jelenik meg a 4.11. ábra Window Properties ablaka.

Az ablak címsorát (Window Caption), az ablakkezelő értékét (Window Handle), az ablakeljárás címét (Window Proc), az ablak által meghatározott téglalap adatait (Rectangle), kliensterületének adatait (Client Rect), a hozzá tartozó vezérlő azonosítóját (Control ID)... mutatja a General fül.

A Window Properties Styles fülében megnézhetjük az adott ablak aktuálisan beállított stílusait. (Lásd 3.1. gyakorlat!)

A Window Properties Windows fül mutatja például gyermekablak esetén a szülőablak ablakkezelőjét, melyet a Spy++ főablakában a hierarchiában azonosíthatunk...



4.11. ábra Az amőba játék (4.1. gyakorlat) ablakai

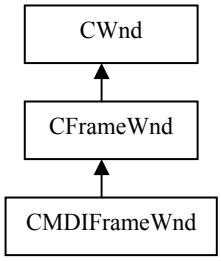
A kiválasztott ablak gyorsmenüjéből a Messages menüpontot választva a kinyíló ablakban megnézhetjük a kiválasztott ablaknak érkező üzeneteket. még egyszerűbb, ha a Spy menü / Messages /Window fülben a Finder Tool-t az ablak fölé visszük, a Messages fülben pedig kiválasztjuk az üzenetek szűrőjét (2.10, 2.11. ábra), OK után a keresett ablak keresett üzeneteit látjuk kilistázva (2.12. ábra).

4.8.4. A CFrameWnd osztály

Az ablakok keretének kezelését biztosítja. A Document/View alkalmazások **CMainFrame** osztályának közvetlen őse.

- Max / Min ikon
- Bezáró gomb
- Címsor
- Menüsor
- Görgetősávok
- Státuszsor
- Eszközsor

kezelését biztosítja.



4.12. ábra Keretablakok

A CFrameWnd egy SDI alkalmazás keretét adja, míg utóda, a **CMDIFrameWnd** az MDI alkalmazását. (Lásd 6. fejezet!)

A keretablak egyes elemei az erőforrás-szerkesztőben szerkeszthetőek. Természetesen futásidőben is módosíthatjuk tulajdonságait. (További részletek az 6.3. fejezetben.)

Az alkalmazásvarázsló által generált ablak címét létrehozásakor beállíthatjuk, de futás közben a ***SetWindowText*** tagfüggvénnyel bármikor módosíthatjuk. Vezérlő esetén ez a tagfüggvény módosítja a vezérlő szövegét. (Caption)

A keretablakosztály biztosítja a **CControlBar** objektumok számára a dokkolhatóságot. (Dokkolt eszközsort az ablak széléhez közelítve az eszközsor az ablak széléhez "ragad".) A ***DockControlBar()*** metódus segítségével beállíthatjuk az első paraméterben megadott controlbar dokkolását. További paraméterei alapértelmezettek. A második paraméter default értéke 0, ami a keretablak bármelyik oldalára engedélyezi a dokkolást azon oldalak közül, amit a keretablakosztály és a controlbar ***EnableDocking()*** metódusában egyaránt engedélyeztünk. A harmadik paramétere LPCRECT típusú és képernyő-koordinátákban adja meg, hogy a keretablak nem kliensterületén hova tapad a controlbar.

4.8.5. A CView osztály

A háttéradatok nézeteit biztosítja. Pl. Excelben a táblázatos és grafikus ábrázolást.

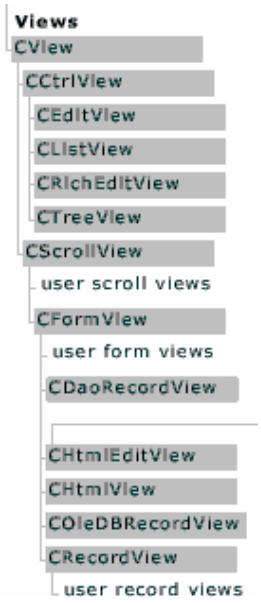
A CWnd utódosztálya.

Az MFC több utódosztályát is a rendelkezésünkre bocsátja. (4.13. ábra)

Absztrakt osztály (OOP 5.3.5. fejezet) alkalmazásainkban általunk létrehozott utódaival dolgozunk.

A Document/View szerkezet alaposztálya, mely az adatok megjelenítéséért felel. Kapcsolatban áll a keretablak- és a dokumentumosztállyal.

Az alkalmazásvarázsló utolsó lépésében, amikor felsorolja a létrehozott osztályokat és forrásfájljaikat a nézetosztály őse a kívánalmaknak megfelelően módosítható.



4.13. ábra A View osztályok

A 6.4. gyakorlat több nézettel dolgozik.

4.8.6. Drag & drop technika

A CView és utódosztályai kezelik a következő tagfüggvények segítségével:

- ***OnDragEnter***: Amikor az egér először belép az ablakba, akkor hívódik meg. Tájékoztat, mi történik, ha elengedjük az ablak fölött a vonszolt objektumot.
- ***OnDragOver***: Az egér ablak fölötti mozgatása közben hívódik.
- ***OnDragLeave***: Meghívódik, amikor az egér elhagyja az ablakot.
- ***OnDrop***: Akkor hívódik, amikor a felhasználó az ablak fölött elengedi az egér által vonszolt objektumot.

Más osztályok is rendelkeznek a drag & drop technika megvalósításához tagfüggvényekkel. Ilyen pl. a CWnd::DragAcceptFiles metódusa, melynek alkalmazásával az 6.1. gyakorlaton találkozhatunk, vagy a CDragListBox osztály, melyről a 4.8.8. szakaszban lesz szó.

A 4.2.6. feladat használja a drag & drop technikát.

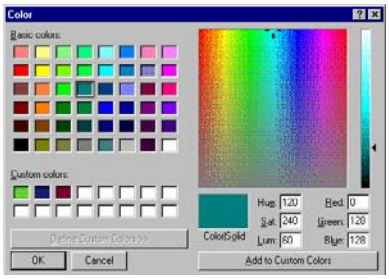
4.8.7. A CDialog osztály és utódai

Alaposztály. Alkalmazásaink fejlesztése során utódosztályaival dolgozunk. **Feladata a párbeszédablakok kezelése.** Létrehozhatjuk modális (modal), vagy nem modális (modeless) módon. Modális ablak esetén az ablak bezárásáig szünetel a többi ablak üzenetfeldolgozása.

Többnyire először az erőforrás-szerkesztővel elkészítjük a párbeszédfelületet, majd az osztályvarázslóval csatoljuk hozzá a CDialog utódosztályt az erőforráshoz.

Az MFC számos utódosztályt tartalmaz, melyekhez az erőforrások is biztosítottak. Itt csak néhányat említünk, melyek e könyvben előfordulnak.

- **CColorDialog**: **Színek kiválasztására szolgál**. A 5. gyakorlat 3. feladatában találkozunk vele. (4.14. ábra) A COLORREF típusú értékkel visszatérő ***GetColor()*** metódusával kérdezhetjük le az ablak bezárása után a kiválasztott színt.

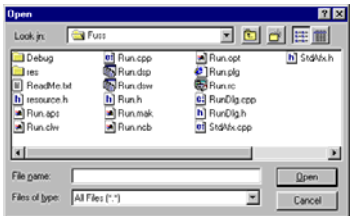


4.14. ábra A CColorDialog

A 5.3. gyakorlat leírásában szerepel.

- **CFindReplaceDialog**: **Keresés és csere esetén hívható**. Ezt az osztályt használja a Word és a Visual C++ keresője is. Metódusa pl. a ***FindNext*** vagy a ***ReplaceAll***.

- **CFileDialog**: Feladata, hogy az operációs rendszer fájlstruktúrájában támogassa a fájlok kiválasztását, mentését. Többnyire ezt használjuk az **Open és a Save As menüpontok választása esetén**. (4.15. ábra) Konstruktorának első paraméterében határozhatjuk meg, most épp megnyitni (TRUE), vagy menteni akarunk (FALSE). Második paramétere az alapértelmezett fájl kiterjesztés, ezt meghatározva, ha mentéskor nem adjuk meg a fájl neve után a kiterjesztést, azt automatikusan a fájlnév mögé fűzi, és megnyitáskor sem szükséges a kiterjesztést beírnunk. Harmadik paraméterében beállíthatjuk az alapértelmezésben kiválasztott fájlt is. Ez esetben, az ablak megnyitásakor a fájl könyvtára lesz az aktuális könyvtár. Az ötödik paraméter adja meg a szűrőt. A szűrő sztring szintaxisa a következő: Az első szakasz a szűrőablak szövege '|' jellel elválasztva a szűrőfeltétel. Újabb '|' jel az új sor előtt. Ha egy sorban több különböző típust is megengedünk, azokat ';' választja el egymástól. A szűrőt két '||' jel zárja. Pl. a következő konstruktorhívás egy kétsoros szűrőt tartalmaz:



4.15. ábra A CFileDialog

```
CFileDialog dlg(FALSE, "emf", "C:\\Haz.emf",
       OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
       "Meta files (*.emf; *.wmf)|*.emf;*.wmf |
        All files (*.*)|*.*||");
```

Hibás fájlnév kezelésére felülírhatjuk egy utódosztályban a virtuális ***OnFileNameOK()*** metódust.

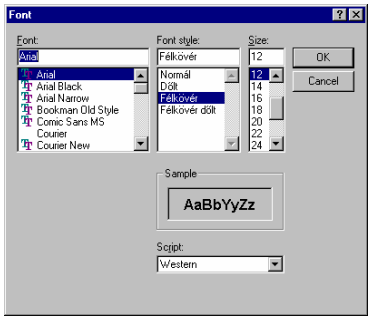
Az ablak bezárása után az CFileDialog objektumra meghívva a ***GetFileName()*** / ***GetPathName()*** metódusokat használhatjuk a kiválasztott fájlt.

```
if (dlg.DoModal())
{
       CopyMetaFile(h,dlg.GetFileName());
}
```

A fejezethez tartozó 4.2.5. és 4.2.11. feladatban valamint az 5.1. gyakorlat leírásában találkozunk használatával.

- **CPrintDialog**: Nyomtatáskor használatos. Többnyire **a File menü Print menüpontjának hatására nyílik meg**. Lehetővé teszi több telepített nyomtató esetén a nyomtató kiválasztását és egyéb nyomtatási beállításokat.

- **CFontDialog**: A betűk beállítására használatos. **Kiválaszthatjuk a betű típusát, méretét, stílusát**. A konstruktor első paraméterében átadott LOGFONT struktúrában állítja be a font tulajdonságait. (4.16. ábra.)



4.16. ábra A CFontDialog

A CFontDialog használatára az 6.3. gyakorlatban látunk példát.

A CDialog utódosztályai közül említést érdemel még a **CPropertySheet** és a **CPropertyPage** osztály. Segítségükkel többfülű alkalmazásokat készíthetünk kényelmesen. (Lásd kód!) Csak létre kell hoznunk a fülek erőforrásait (IDD_PROPPAGE_), majd hozzájuk rendelni a CPropertyPage-ből származó megfelelő nevű osztályokat (CPage1, CPage2...)! Ezután létre kell hozni a CPropertySheet és a CPageX objektumokat, majd felfűzni őket a CPropertySheet-re (**AddPage**)! A CPropertySheet objektum a továbbiakban úgy kezelhető, mint az eddigi párbeszédablakok.

```
{
 CPropertySheet sheet("Fülek");
 CPage1 page1;
 CPage2 page2;
 sheet.AddPage(&page1);
 sheet.AddPage(&page2);
 if (IDOK == sheet.DoModal())
        {
               //Adatok kiolvasása a 'Page' objektumokból.
        }
}
```

A PropertySheet ablak címe az objektum konstruktorának paraméterében, míg a fülek feliratai erőforrásaik Caption tulajdonságában állíthatók be. A ***CPropertySheet::SeTitle()*** metódus is a cím beállítását szolgálja.

A PropertySheet ablak három gombot tartalmaz, OK, Cancel (Mégse) és Apply (Alkalmaz) gombot. Ha az Apply gombot nem akarjuk használni, az ablak létrehozása előtt a következő kódsorral vehetjük le a felületről: .

```
sheet.m_psh.dwFlags |= PSH_NOAPPLYNOW;
```

 Használatára a 5.3. gyakorlaton látunk példát.

A CPropertySheet osztály ***SetWizardMode()*** tagfüggvénye segítségével varázslókat is készíthetünk. Ez esetben fülek nincsenek, és egymás után következnek a felfűzött lapok. Az OK, Cancel és Apply gombok pedig Back, Next és Finish gombokra módosulnak.

Megjegyzés:

A Finish gomb a **PSH_WIZARDHASFINISH** flagbeállítás mellett jelenik meg. Választása után a ***DoModal()*** **ID_WIZFINISH** értékkel tér vissza. További értékek a prsht.h-ban.

4.8.8. A vezérlők a CWnd utódai

4.8.8.1. Néhány további vezérlőosztály

Most csak néhány vezérlő osztályt említünk. A legtöbb vezérlő a CComboBox-hoz hasonlóan közvetlenül a CWnd osztályból származik, és nincs az MFC-ben utódosztálya. Ezeket nem mutatja a 4.17. ábra. A második fejezetben többel találkozhattunk közülük.

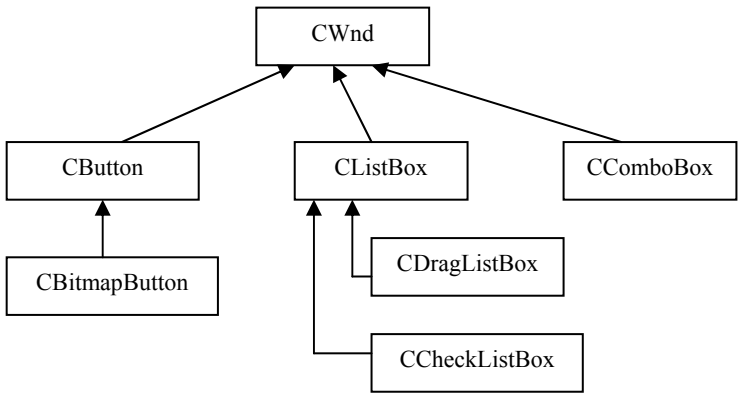
- **CEdit**: A szerkesztőmezőhöz tartozó CEdit osztály ismeri az alapvető szövegszerkesztési funkciókat. A szerkesztőmezőben az egérrel (vagy billentyűzettel) kijelölve egy szövegrészt, majd a **Ctrl C** gyorsgombot használva, az a vágólapra másolódik. A vágólapon található szöveget a **Ctrl V** gyorsgombbal a (fókuszban levő) szerkesztőmezőbe illeszthetjük. A kivágáshoz a **Ctrl X** gyorsgomb is használható. E műveleteket kódunkból is meghívhatjuk.
  - ***SetSel***: **void SetSel(int** *nStartChar***, int** *nEndChar***, BOOL** *bNoScroll* **= FALSE);** A kijelölést végzi. Első két paramétere a kijelölés első és utolsó karaktere. A (0,-1) paraméter a teljes szöveg kijelölését jelenti. Ha az első paraméter -1, az megszünteti az eddigi kijelölést. A ***GetSel***(**int&** *nStartChar***, int&** *nEndChar*) metódussal lekérdezhetjük az érvényben levő kijelölés elejét és végét.
  - ***Copy***: Ha van a kijelölt szöveg, azt a vágólapra másolja.
  - ***Paste***: Ha van a vágólapon szöveg, azt beilleszti.
  - ***Cut***: Ha van kijelölt szöveg, azt a vágólapra másolja, és kitörli.
  - ***Clear***: Ha van kijelölt szöveg, azt törli (anélkül, hogy a vágólapra másolná).

- ***Undo***: Az utolsó művelet előtti állapotot állítja vissza. A Clear művelet által törölt adat így visszaállítható.

➧ **CBitmapButton**: A CWnd-ból származó CButton osztály utóda.

- ***LoadBitmaps***: Inicializálja a paraméterekben megadott legalább egy, legfeljebb négy bittérképpel a gombot. A paraméterek erőforrás azonosítók, vagy fájlok.
- ***SizeToContent***: Átméretezi a gombot a bitmap méretének megfelelően.

A 3.3.6. feladatban ezt az osztályt használtuk föl a duplakattintás és a képet mutató nyomógomb megjelenítésére. A feladat megoldása részletezi a CBitmapButton osztály használatának feltételeit.



4.17. ábra Vezérlő osztályok

➧ **CListBox**: Feladata a listák megjelenítése, kezelése. Főbb tagfüggvényei:

- ***AddString***: A paraméterben megadott új elem fölvétele a lista végére.
- ***DeleteString(n)*** : Az n. adat törlése. A listában maradt utolsó elem indexével tér vissza.
- ***InsertString***: **int InsertString(int** *n***, LPCTSTR** *lpszItem* **)** A második paraméterben megadott elem beszúrása az n. helyre. n = –1 esetén a lista végére szúr be.
- ***GetCurSel***: Visszatér a kiválasztott elem indexével. A sorszámozás 0-val kezdődik.
- ***SetCurSel***: Beállítja a sorszáma alapján az aktuális listaelemet.
- ***Dir***: **int Dir(UINT** *attr***, LPCTSTR** *lpszWildCard* **)**: Az attribútumban megadott típusú (Lásd súgó!) a második paraméterben helyettesítő

karakterek segítségével meghatározott fájlok (pl. *.*) nevét fűzi föl a listára. Az utolsó felfűzött fájl indexével tér vissza.

- ***GetCount***: **int GetCount( ) const;** A listadoboz elemeinek számával tér vissza.
- ***GetItemHeight:*** **int GetItemHeight(int** *nIndex* **) const;** A listadoboz paraméterben megadott indexű elemének pixelekben mért magasságával tér vissza.
- ***FindString: int FindString(int n, LPCTSTR*** *lpszItem* **) const** Az n. elemtől keres, majd a végére érve folytatja az elejéről. Az n. elemig dolgozik. n = –1 esetén elölről keres. A második paraméterben megadottal kezdődő elem indexét adja vissza. Az aktuális elem a keresés során változatlan (const).
- ***SelectString***: **int SelectString(int** *n***, LPCTSTR** *lpszItem* **)** Az előzővel azonos, csak a megtalált elem lesz az aktuális. Ha a keresés sikertelen, az aktuális elem változatlan.

↳ **CDragListBox**: A ListBox vezérlőnkhöz egy Control típusú CDragListBox objektumot is rendelhetünk.

- Nem lehet rendezett. (Properties / Styles / Sort vagy **LBS_SORT**)
- Nem engedélyezheti több elem választását. (**LBS_MULTIPLESELECT**)

Töltsük föl az AddString-gel! Az elemek sorrendje drag&drop-pal állítható. (4.18. ábra)

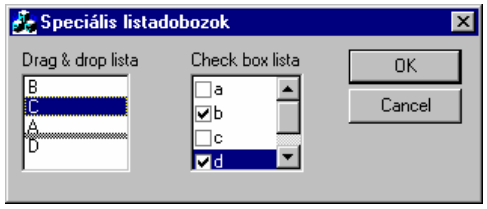
Megjegyzés:

Az LBS_ a List-Box Styles rövidítése.

↳ **CCheckListBox**: A ListBox vezérlőnkhöz egy Control típusú CCheckListBox objektumot is rendelhetünk.

- Properties / Styles: **Owner draw: fixed**
- Properties / Styles: **Has Strings** jelölőnégyzet kiválasztva.

Töltsük föl az AddString-gel! A kiválasztott elemek jelölve lesznek. (4.18. ábra)



4.18. ábra A CDragListBox és a CCheckListBox

↳ **CTreeCtrl**: Segítségével adatainkat fa szerkezetben ábrázolhatjuk.

- Properties / Styles: **Has buttons** jelölőnégyzet kiválasztása esetén a szokásos kis négyzetek is megjelennek a gyermekelemet tartalmazó tagok előtt.
- **HTREEITEM**: Az egyes elemek kezelője. Segítségével hivatkozhatunk az egyes elemekre. Minden elem kezelője más és más, de néhány speciális elemre azonosítókkal is hivatkozhatunk pl. **TVI_ROOT**=gyökérelem, **TVI_FIRST** / **TVI_LAST** = első / utolsó elem **TVI_SORT** = rendezetten. (TVI=TreeViewItem)
- ***InsertItem***: Az egyes elemek felfűzésére szolgál. Visszatérési értéke a felfűzött elem kezelője. A hInsertAfter paraméterében szokás megadni a felfűzés módját a fenti azonosítókkal.
- ***GetRootItem***: Visszaadja a fa gyökérelemét. Ha üres a fa, visszatérési értéke NULL.
- ***GetItemText***: Visszatér a paraméterében megadott kezelőjű elem szövegével.
- ***ItemHasChildren***: Igaz, ha a paraméterében megadott elemnek vannak gyermekei.
- ***GetChildItem / GetParentItem***: Visszatér a paraméter gyermekének / szülőjének kezelőjével.
- ***GetNextItem***: Visszatér a szint következő elemével. Ha nincs több elem, akkor NULL értéket ad vissza.
- ***GetCount***: A vezérlő elemeinek számát adja.
- ***DeleteItem / DeleteAllItems***: Elem / összes elem törlése.
- ***SelectItem / Select***: Kiválasztja a paraméterében megadott kezelőjű elemet. Paramétertől függően kirajzolja az elemet a beállított stílusnak

megfelelően, vagy úgy görgeti a fát, hogy a megadott elem lesz az első látható.

 Használatát a 6.6.7. feladatban láthatjuk.

4.8.8.2. Vezérlők futásidejű létrehozása

Mivel a vezérlők maguk is ablakok, így a 4.8.3. szakaszban tárgyaltaknak megfelelően két lépésben létrehozhatóak és megszüntethetőek. Azt kérdezhetnénk, nem egyszerűbb az erőforrás-szerkesztővel létrehozni, majd láthatatlanná tenni egy vezérlőt, aztán szükség esetén megjeleníteni? De igen, ha fordításidőben tudjuk a vezérlők számát, helyét, ha használható az erőforrás-szerkesztő...

Mi van abban az esetben, ha futáskor dől el, hány darab vezérlőre lesz szükség a feldolgozás során? Ilyenkor bizony hagyományos módon ki kell számolnunk a vezérlők méretét, helyét, meg kell határoznunk a stílusukat és a ***Create()*** függvény segítségével futásidőben is létrehozhatóak a vezérlők. Egy másik példát a 6. gyakorlat 4.1. szakaszában láthatunk, ahol Create segítségével hozunk létre listadobozt. A vezérlő osztályokban többnyire felülírták a CWnd osztály virtuális Create metódusát. Így létrehozásukkor az osztály saját Create metódusát hívjuk saját paraméterlistájával.

4.9. Az egyszerű érték típusok kategória osztályai

4.9.1. A CString osztály

A sztring változó hosszúságú karaktersorozatot takar. Szöveges adatok kezelését teszi egyszerűbbé. Nélküle a szövegeket, mint karaktertömböket kellene kezelni. Bár a sztring mérete dinamikusan változtatható, mégsem kell mutatóként bánni vele. Sztring konstansokat kettős idézőjelek közt adhatunk meg. Tartalmazhatnak escape szekvenciákat, például az új sort \n, tabulátort \t és a '\': \\ jellel adjuk meg.

A CString osztálynak nincs ősosztálya.

- #include <afx.h>
- Változó hosszúságú karaktersorozat.
- Első karaktere a 0. helyen.
- [ ]operátorral hivatkozhatunk egy karakterére.
- A + a konkatenáció, az egymás után fűzés művelete.

- Alkalmazhatjuk a const char* vagy a LPCTSTR (lásd később) függvényparaméter helyett.

Az osztály függvényei a szokásos sztringműveletek mindegyikéhez támogatást nyújtanak. A következő lista közülük emel ki néhányat:

```
GetLength( )
IsEmpty( )
GetAt( )
SetAt( )
Mid( )
Left( )
Right( )
Delete()
MakeUpper( )
TrimLeft( )
```

Ha sztringet úgy szeretnénk formázni, ahogy azt a printf függvénnyel tettük a C-ben, használjuk a ***Format*** függvényt! Pl:

```
CString str;  str.Format("%d szöveg", szám);
```

A sztrigben a keresést a ***Find*** és a ***FindOneOf*** függvények támogatják.

Visszatérési értékük a keresett karakter indexe a sztringben. Ha nincs találat, -1-gyel térnek vissza. Karaktert vagy sztringet is kereshetünk. A Findban második paraméterben megadhatjuk az indexet, ahonnét a keresést kezdjük. A FindOneOf a paraméterben megadott karaktersorozat bármelyikének megtalálásakor visszatér.

A ***SpanIncluding*** / ***SpanExcluding*** A paraméterben megadott karaktersorozattal megegyező / nem megegyező első elemeivel tér vissza a sztringnek. A művelet az első a paraméterben nem megadott / megadott elem esetén befejeződik. PL:

```
CString string("cababsoab");
CString result = string.SpanIncluding("abc");
```

A result tartalma: "cabab".


A 6.6.7. feladat OnInitialUpdate függvényében a CString osztály több metódusát is használjuk.

Megjegyzés:

Amennyiben UpdateData() hívás esetén szám van a szerkesztőmezőhöz rendelve, és a szerkesztőmező üres, vagy betűt is tartalmaz, akkor float típus esetén "Please enter a number!", int típus esetén "Please enter an integer!" üzenetet kapjuk. Ha magyar nyelvű üzenettel helyettesítjük, vagy egyszerűen 0-nak vesszük a

számot ilyen esetben, ezt a következő kóddal tehetjük meg az UpdateData() hívása előtt:

```
//Az Elad (3.1. gyakorlat) alkalmazás súlybevitel
//szövegmezőjéről van szó. Beállítja, hogy üres vagy
//betűt tartalmazó súly esetén magyarul üzenjen.
CString suly;
GetDlgItem(IDC_SULY_EDIT)->GetWindowText(suly);
if (suly.IsEmpty() ||
!suly.SpanExcluding("0123456789").IsEmpty())
{
      MessageBox("Kérem számot adjon meg!");
      GetDlgItem(IDC_SULY_EDIT)->SetWindowText("0");
}
```

Gyakran találkozunk az **_T("...")** makróval, melynek feladata a sztring karaktereinek Unicode-ra konvertálása, ha a #define _UNICODE fordítási direktíva definiálva lett programunkban. Előnye, hogy a konstans sztringeket változtatás nélkül használhatjuk hagyományos és unikódos alkalmazások esetén. Sok függvény Unicode paramétert vár. Pl. **wchar_t** (wide character).

Megjegyzés:

A **Unicode** lehetővé teszi számunkra a világ összes ábécéjének használatát.

Az **ASCII** kód egy 8 bites ISO szabvány. A különböző nemzetek karaktereit úgynevezett kódlapok használatával érhetjük el. Ezek használatakor a felső 128 karakter az adott terület speciális karaktereit tartalmazza. A magyar betűk a 852-es kódlapon találhatóak. Használata során a többnyelvű szövegek kiírása okoz problémákat.

A Unicode 1987-ben a XEROX kutatási eredményeként jött létre. A karaktereket 16 biten tárolja. Nevében az Uni három szó kezdetére utal: Unique (egyedi), Universal (univerzális), Uniform (egységes).

A Unicode részei:

- Alfabetikus karakterek.
- Műszaki piktogramszimbólumok.
- Kínai, japán, koreai írásjegyek (≈ 27000 Han karakter).
- A felhasználó által generálható karakterek.
- Konvertálást segítő karakterek.

Függvények paramétereiként használatos sztringmutatók:

- **LPCSTR**: 32 bites mutató egy konstans sztringre, const char*. (LongPointerConstSTRing)
- **LPSTR**: 32 bites mutató egy sztringre, char*.
- **LPCTSTR**: 32 bites mutató egy konstans sztringre, mely Unicode és **DBSC** portábilis. (DBSC: Double-Byte Character Set) Megegyezik a const char*-gal, ha nem használunk Unicode-ot.
- **LPTSTR**: 32 bites mutató egy sztringre, mely Unicode és DBSC portábilis. Megegyezik a char*-al ha nem használunk Unicode-ot.

A CString és a sztringmutatók kölcsönösen helyettesíthetik egymást, ha használjuk az LPCTSTR(CString) és a CString(LPCTSTR) konverziós operátorokat.

A legtöbb függvényben CString paraméter helyett LPTSTR áll, mert így híváskor paraméterként a TCHAR tömb mutató, a literális konstans ("AAA") és CString objektum is explicit konverzió nélkül használható.

4.9.2. A CTime és a CTimeSpan osztály

Az egyszerű változó típusok kategóriájában található, az idő kezelésére alkalmas osztályok. A CTime objektumok az abszolút idő, míg a CTimeSpan objektum a relatív idő tárolására szolgálnak. Két CTime objektum közti idő különbségét egy CTimeSpan objektum adja.

- Ősosztályuk nincs.
- Több konstruktoruk is van, hogy kényelmessé tegyék a létrehozást. (Lásd súgó!)

```
CTime(int nYear, int nMonth, int nDay,
                int nHour, int nMin, int nSec, int nDST = -1);
CTime time(2000, 10, 5, 12, 0,0);
```

  Hibás adat estén debug üzemmódban hibaüzenet, release módban korrekció. Pl. 2000, 22...-t átalakítja 2001, 10. hónapra.

- A CTime objektum tartalmának lekérdezését is sok tagfüggvény támogatja. A Get függvények nem változtatják meg a CTime objektum adatait.

```
int GetYear() const;
int GetMonth() const;
int GetDay() const;
int GetHour() const;
int GetMinute() const;
int GetSecond() const;
int GetDayOfWeek() const;
// 1=Sunday, vasárnap, 2=Monday, hétfő, .., 7=Saturday, szombat.
```

- Időadatot a kiíráshoz CString típusúvá alakíthatunk a Format tagfüggvény segítségével.

```
CTime time(2000,11,10,3,10,0);
 CString st;
 st = time.Format("%Y év, %m hó");
 %Y négyjegyű évszám, %y kétjegyű évszám
 %m hónap
 %d nap
 %H óra 24 órás kijelzésben, %I 12 órás kijelzésben
 %M perc
 %S másodperc
 %p 12 órás kijelzésnél délelőtt AM délután PM.
```

- ***CTime::GetCurrentTime()*** az aktuális rendszeridőt adja vissza. Osztályszintű metódus (static) (OOP 1.12. fejezet).

```
CString st = CTime::GetCurrentTime().Format("%Y,%m");
```

- A CTime +, - operátorok használatánál a CTimeSpan használata elkerülhetetlen. A CTimeSpan másodpercekben tárolja az időt 4 byte helyen. Megengedett maximális értéke kb. ± 68 év.

 Az idő kijelzésével a státuszsorban a 6.6.4. feladat foglalkozik.

4.10. Az osztályok tárolása

Alapvetően két file-ban tárolódnak:

- **Interface file**: Ez általában a header file (.**h**).

  Ebben valósulnak meg a deklarációk és egyéb más információk, melyek az osztály használatához szükségesek.

- **Implementation file**: Ez általában a .**cpp** file.

  Ez tartalmazza a definíciókat, tehát pl. a tagfüggvények kódját.

  Tartalmazhat néhány kisebb osztályt is deklarációval és implementációval együtt. Pl. About.

4.11. Globális függvények

Jellemzőjük, hogy minden függvényből elérhetőek. Többnyire az Afx betűkkel kezdődnek.

Megjegyzés:

Afx (Active Framework Extension) a globális függvények szokásos kezdőbetűi. A globális változók **afx** kezdetűek.

A makrókat csupa nagybetűvel írjuk.

- ***AfxAbort()***: Feltétel nélkül leállítja az alkalmazás futását. A program terminál.

- ***AfxBeginThread()***: A függvény lehetővé teszi programunk számára több szál elindítását, majd meghívja a globális ***::CreateThread*** függvényt, amely elindítja a szálat.

  - **Visszatérési értéke**: egy **CWinThread*** mutató.

  - **Paraméterei**:

    Az első paraméter azt a függvényt határozza meg, amellyel a szál a végrehajtást kezdi. A függvény természetesen hívhat további függvényeket.

    A szálnak a második paraméterben megadott pParam mutatón keresztül adhatunk át információt.

    A harmadik paraméter a szál prioritását határozza meg. Minél magasabb a prioritás, annál gyakrabban kapja meg a szál a CPU-tól a vezérlést. E paraméter **THREAD_PRORITY_NORMAL** értékkel alapértelmezett, és minden további paraméter is az. (OOP1.4.2. szakasz)

- ***AfxEndThread()***: Ha nem akarjuk megvárni, hogy az AfxBeginThread első paraméterében meghatározott függvény visszatérjen, az AfxEndThread függvény segítségével állíthatjuk meg a szálat.

- ***AfxGetThread()***: Az aktuálisan végrehajtott szál mutatójával tér vissza.

- ***AfxGetApp()***: Visszaad egy CWinApp típusú mutatót az alkalmazás CMyApp objektumára.

- ***AfxGetAppName()***: Az alkalmazás nevét tartalmazó sztringre mutató mutatót ad vissza.

- ***AfxGetInstanceHandle()***: Az aktuális alkalmazáspéldány mutatóját adja vissza. (.exe, .dll)

- ***AfxGetMainWnd()***: A főablakra mutató mutatóval tér vissza.

- ***AfxMessageBox()***: Már találkoztunk vele, üzenetablakot készít, a nem CWnd-ből származó osztályokban használjuk.
- ***DDX_***...: A globális függvények közé tartoznak a vezérlők és a hozzájuk rendelt változók közötti kapcsolatot meghatározó függvények. Az adatbázis-kezelésnél a kapcsolattartásra használatos függvények is globálisak.

Az API függvények mindegyike globális függvényként érhető el. Álljon itt egy példa erre:

- ***GetTickCount***: A gép bekapcsolása óta eltelt idővel tér vissza milliszekundumban. Mivel a visszatérési típusa DWORD, ami egy 32 bites változó, így 39,7 naponként elölről kezdi a számlálást. (NT vagy Windows 2000 szerver esetén a bekapcsolás óta eltelt időt a registry 8 byte-os HKEY_PERFORMANCE_DATA adatából kaphatjuk meg, illetve a time globális függvény segítségével kérdezhetjük le.)

4.12. Kérdések

1. Ismertesse az MFC szerepét, felépítését!
2. Mit biztosít a CObject osztály utódai számára?
3. Ismertesse az alkalmazásszerkezeti osztályokat!
4. Milyen típusait különböztetjük meg a szálaknak? Részletezze a különbséget! Ismertesse, melyik szálat hogyan hozhatjuk létre!
5. Milyen szempontok szerint csoportosíthatjuk az ablakokat?
6. Ismertesse a CWnd osztály feladatát, néhány tagfüggvényét! Mikor hozzuk létre, és hogyan érhetjük el az operációs rendszer ablakobjektumát?
7. Hogyan használhatunk időzítőt programunkban?
8. Ismertesse néhány utódosztályát a CWnd osztálynak!
9. Mely osztályok biztosítják többfülű alkalmazás létrehozását? Ismertesse a kódolás menetét!
10. Beszéljen a CListBox és utódainak kezeléséről!
11. Ismertesse a CString osztályt és a rajta végezhető műveleteket!
12. Milyen osztályt biztosít az MFC az idő kezelésére?

4.13. Összefoglalás

- Az **MFC (Microsoft Foundation Classes)** egy osztálykönyvtár. Magasabb szintű Windows absztrakciót tesz lehetővé, így biztosítja a gyorsabb fejlesztés lehetőségét. Objektumorientált technikákat biztosít az alkalmazás fejlesztéséhez.
- Az MFC biztosítja az osztályok összes **forráskód**ját. Az eligazodást a deklarációkban és implementációkban egységes megjegyzések segítik.
- A **CObject osztály** a legtöbb MFC osztály őse. Utódosztályai számára lehetővé teszi a **szerializáció** és a **futásidejű információk** elérését. Támogatja a **konténer osztályok** használatát.
- Az **alkalmazásszerkezeti osztályok** az alkalmazás indításakor jönnek létre. A **CCmdTarget** osztály utódai, melyek az osztályok **üzenetkezelési mechanizmus**át biztosítják. **A CWinThread osztály a CWinApp osztály közvetlen őse**. **Biztosítja szálak létrehozását, kezelését**. Tartalmazza az **InitInstance**, a **Run,** az **ExitInstance és az OnIdle** metódusokat.
- Az ablakok egymás fölötti elhelyezkedését z-sorrendnek hívjuk. Az ablakok között szülő-gyermek, tulajdonos-tulajdon kapcsolat lehet.
- **Az ablakok alaposztálya a CWnd**. Tartalmazza az ablakok kezeléséhez szükséges tagfüggvényeket. Az **m _hWnd adattagjá**n keresztül elérhetjük az operációs rendszer által létrehozott **rendszerablak**ot. **Ablakot két lépésben hozunk létre**. A konstruktor létrehozza az ablakobjektumot, míg a Create a rendszerablakot. **Megszüntetni is két lépésben lehet**, a DestroyWindow a rendszerablakot, míg a destruktor a C++ ablakobjektumot szünteti meg.
- **Időzítőt** a CWnd osztályokban **hozhatunk létre a SetTimer tagfüggvénnyel**. **Megszüntetni a KillTimer metódussal lehet**. A **WM_TIMER** üzenetet többnyire az **OnTimer** tagfüggvénnyel kezeljük.
- A CWnd főbb utódosztálya **a CFrameWnd, mely a keret, címsor, menüsor, gördítősávok, státuszsor és az eszközsor kezeléséért felelős**. A **CView** osztály az **adatok megjelenítésé**t biztosítja, és kezeli a **drag & drop technika** függvényeit. A **CDialog** osztály előre elkészített utódosztályaival a **párbeszédfelület felhasználóbarátságá**t támogatja. A többi vezérlővel együtt a CListBox és utódai is a CWnd utódosztályai.
- A **CString osztály** nem utódja a CObject-nek. A **sztringek kezelésé**hez biztosít kényelmes felületet a programozó számára. A **CTime** osztállyal az **időadatok**at adhatjuk meg és kérdezhetjük le. **GetCurrentTime** statikus tagfüggvénye az **aktuális rendszeridő** elérését teszi lehetővé.

5. Megjelenítés a grafikus felületen

A **karakteres alkalmazás** (**character-mode application**) olyan alkalmazás, mely nem gondoskodik saját grafikus felületről. **Az egyes karakterek képeit karaktergenerátor segítségével határozza meg.** A különböző kódtáblák különböző karaktereket tárolnak, megjelenítéskor az aktuális kódtábla megfelelő karakterét látjuk a képernyőn. A karakterek lehetnek akár kis ábrácskák is.

A Windows alkalmazások **grafikus alkalmazás**ok. A megjelenítéshez a **GUI**-t (**Graphical User Interface**) használják. **A kijelzés** itt nem karaktergenerátor segítségével történik, hanem **képpontok, raszterpontok segítségével, vektor és TrueType technikával**. A grafikus alkalmazások gyors elterjedését felhasználó-barátságuk indokolja. Egy kép felülete jóval szemléletesebb, mintha csak írt szöveget látunk. Megfelelnek az "Amit látsz, azt kapod!" elvnek (**WYSIWYG** = **What You See Is What You Get**).

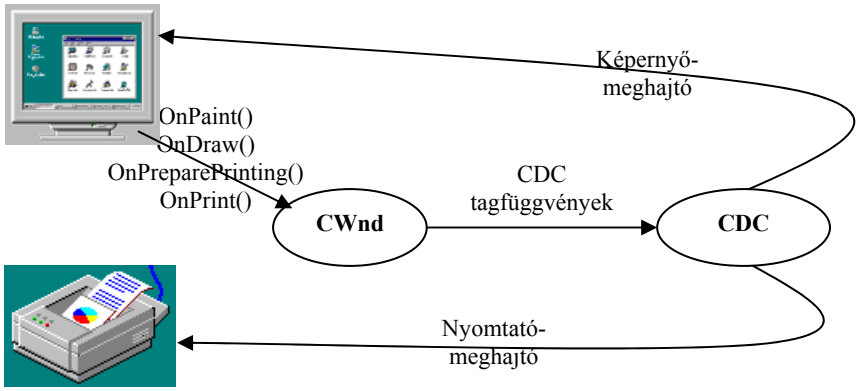
5.1. A DC

A DC (**device context**) leírja az **eszköz környezet** elemeit**. A GDI számára azokat az információkat tartalmazza, melyek az éppen használatos környezetben történő megjelenítéshez szükségesek.** Rögzít egy sor rajzeszközt, melyeket a rajzoláshoz használhatunk és a rajzolás módját. Segítségével nem kell minden egyes vonal kirajzolásakor megadnunk a rajzoló színt, a vonalvastagságot, a stílust.... Tárolja az aktuális pozíciót, az aktuális háttér és betűszínt, valamint az aktuális fontot. Így szöveg kiírásakor, csak a szöveget kell megadnunk.

A **DC** meghatározza:

- A grafikus objektumokat (pl. Pen, Brush, Bitmap).
- Azok jellemzőit.
- A grafikus megjelenítés típusát (Raster, Vector, Pseudo).
- A kiviteli eszközt (képernyő, nyomtató).

A CDC osztály a CObject osztályból származik. **Két rendszer meghajtó kezelő adattagja van, az m_hDC és az m_hAttribDC. Ezek alapértelmezésben megegyeznek, de mutathatnak különböző device contextre is.** Pl. nyomtatásnézet (Print Preview) esetén a kivitel a képernyőre kerül, de a jellemzőket a nyomtatómeghajtóból olvassa a CDC.



5.1. ábra A megjelenítés folyamata

A CDC a kimeneti hívások többségét az m_hDC-nek küldi el pl. ***SetTextColor()***, a beviteli hívások többségét pedig az m_hAttribDC-ből olvassa ki pl. ***GetTextColor()***. Van néhány olyan függvény is, melynek hívása mindkét kezelőre fontos lehet, ezekből két különböző metódus készült pl. ***GetTextMetrics()*** (m_hAttribDC)– ***GetOutputTextMetrics()*** (m_hDC), ***GetCharWidth()*** (m_AttribDC) – ***GetOutputCharWidth()*** (m_hDC).

5.1.1. A CDC objektumai

A CDC osztályhoz CGdiObject objektumokat rendelhetünk hozzá:

- **CPen**: A rajzolt vonalak színét, vastagságát és mintáját tartalmazza. Alapértelmezése: egy pixel széles, fekete, szolid toll.

- **CBrush**: Az alakzatok kitöltő színét és mintázatát adja. Alapértelmezése: fehér, szolid ecset.
- **CFont**: A szöveges karakterek formáját határozza meg. Alapértelmezés: a meghajtó rendszerfontja.
- **CBitmap**: Egy bittérkép, mely pl. ábrák mozgatásakor vagy alakzatok kitöltésekor használatos mintát tartalmaz. Alapértelmezésben üres.
- **CPalette**: Megadja, hogy milyen színek érhetőek el az alkalmazásból. Alapértelmezés: a rendszerpaletta.
- **CRgn**: Egy sokszög vagy ellipszis, mely a kivitelt, bevitelt határolja. Lehet a teljes ablak, beleértve a kereteket is, lehet a kliensterület, de lehet az érvényesítésre váró terület, attól függően, hogy a DC-t milyen céllal hozták létre.

5.1.2. Az objektumok kiválasztása a DC-be

A GDI objektum létrehozása után a ***SelectObject()*** metódussal választható ki a CDC új objektuma. **Használat után állítsuk vissza az eredeti állapotot!**

```
CPen newPen;
newPen.CreatePen(PS_SOLID, 10, RGB(255, 0,0);
CPen* pOriginalPen;
pOriginalPen = dc.SelectObject(&newPen);
...
dc.SelectObject(pOriginalPen);
newPen.DeleteObject();
```

A Windows biztosít néhány GDI objektumot készen. (Továbbiak a súgóban.)

BLACK_BRUSH	**DKGRAY_BRUSH**	
GRAY_BRUSH	**LTGRAY_BRUSH**	
NULL_BRUSH	**WHITE_BRUSH**	
BLACK_PEN	**NULL_PEN**	**WHITE_PEN**
ANSI_FIXED_FONT	**DEVICE_DEFAULT_FONT**	

SYSTEM_FONT: A rendszerfont. Alapértelmezésben a Windows ezt használja menük, vezérlők és más szövegek megjelenítésére. Szélessége proporcionális.

DEFAULT_PALETTE: Alapértelmezett paletta.

DC_BRUSH: Windows 98 és Windows 2000 esetén használható. Színe a ***SetDCBrushColor()*** metódussal módosítható.

DC_PEN: Windows 98 és Windows 2000 esetén használható. Színe a ***SetDCPenColor()*** metódussal módosítható.

Ezek az objektumok a ***CDC::SelectStockObject()*** paramétereként kiválaszthatóak. A ***CGdiObject::CreateStockObject()*** paramétereként hozzárendelhetjük őket a megfelelő GDI objektumhoz is. A ***GetStockObject()*** API függvény a paraméterében megadott azonosítójú rajzolóobjektum kezelőjét (**HGDIOBJ**) adja vissza.

A rajzoló függvény vége előtt állítsuk vissza az eredeti állapotot! A lokális objektum a függvény végén felszabadul. Nem szerencsés megoldás, ha a DC olyan GDI objektumra mutat, ami már nem is létezik!

5.1.3. Hogyan férünk hozzá a DC-hez?

Konstruktorral, pl. a **CClientDC dc(this)** definíció segítségével készíthetünk az aktuális ablak kliensterületéhez egy DC objektumot. A CWnd osztály ***GetDC(),*** illetve ***GetWindowDC()*** metódusait is használhatjuk, melyek egy CDC* mutatóval térnek vissza. A GetDC() a kliensterületre mutató DC-t ad, míg a GetWindowDC() a teljes ablakra vonatkozót. Csak akkor használjuk, ha valamilyen különleges esetben a nem kliens területre is rajzolni akarunk, pl. a keretre, a menüsorra vagy az eszközsorba.

```
CDC* pDC=GetDC();
```

A festés befejezése után a 'Get' függvények használata esetén kötelező a ***ReleaseDC()*** (CWnd::ReleaseDC(CDC* pDC) hívás, mert az egyszerre megnyitható DC-k száma korlátozott, így más alkalmazások esetleg nem tudják megjeleníteni adataikat.

Ha a háttérben, a memóriában megjelenítésre váró adatokat tárolunk, szükségünk lehet egy a megjelenítő eszközzel kompatibilis DC-re. Ezt a ***CDC::CreateCompatibleDC()*** függvénnyel hozhatjuk létre. Hasonló célokat szolgál bitmap létrehozásához a ***CBitmap::CreateCompatibleBitmap()*** metódus.

```
memDC.CreateCompatibleDC(&dc)
bitmap.CreateCompatibleBitmap(&dc, width, height);
```

5.1.4. A CDC osztály utódosztályai

- **CClientDC**: Az ablak kliensterületét adja vissza. Konstruktora hívja a GetDC, destruktora a ReleaseDC metódust.
- **CPaintDC**: A WM_PAINT üzenet kezelésekor használatos. Konstruktora hívja a ***CWnd::BeginPaint()***, destruktora a ***CWnd::EndPaint()*** függvényt.
- **CWindowDC**: A teljes ablakterületre rajzolhatunk segítségével. Konstruktora hívja a GetWindowDC(), destruktora a ReleaseDC() metódusokat.
- **CMetaFileDC**. Egy Windows metafájl (**.wmf** kiterjesztés), vagy továbbfejlesztett **metafájl** (enhanced metafile **.emf** kiterjesztés) a GDI-nek szóló utasítások sorát tartalmazza arról, hogyan jelenítsünk meg egy ábrát vagy

szöveget. GDI utasítások ideiglenes tárolására és lejátszására (ismételt végrehajtására) használható. A metafájl a képet a meghajtótól függetlenül tárolja. Általában metafájlból lassabban rajzolunk mint bitmapből, tehát, ha gyors rajzra van szükség, akkor használjunk bitmapet! A bitmappel szemben viszont a metafájl biztosítja a meghajtófüggetlen rajzolást. Jó megoldás lehet, ha a metafájlban tárolt rajzot az előkészítő tevékenységek során az aktuális meghajtóval kompatibilis bitmapre rajzoljuk, majd a bittérképet – amikor szükség van a képre – gyorsan megjelenítjük.

32 bites környezetben a továbbfejlesztett metafájl használata ajánlott, a wmf csak 16 bitessel kompatibilis környezetben fut.

- ***Create() / CreateEnhanced()*** tagfüggvényének paramétereként adhatjuk meg a fájl nevét. (NULL esetén a memóriában tárolja az adatokat.)
- ***Rajzolás a metafájlba:*** Ezután a DC-nek küldhetünk rajzoló utasításokat, mint LineTo, Rectangle... melyek a metafájlban tárolódnak.
- A ***Close() / CloseEnhanced()*** függvénnyel zárhatjuk a fájlt. A Close tagfüggvény visszaad egy kezelőt (**HMETAFILE** / **HENHMETAFILE**).
- A ***CopyMetaFile() / CopyEnhMetaFile()*** segítségével az alkalmazás végén ideiglenes fájlból, vagy a memóriából adott fájlba menthetjük a képet. Paramétere a kezelő és az új fájl neve.
- A ***DeleteMetaFile() / DeleteEnhMetafile()*** segítségével törölhetjük az ideiglenes metafájlt.
- A ***GetMetaFile() / GetEnhMetaFile()*** megnyit egy metafájlt, és visszaad egy kezelőt ehhez a fájlhoz.
- A ***CDC::PlayMetaFile()*** / ***PlayEnhMetaFile()*** tagfüggvénye segítségével rajzolhatjuk ki az elmentett képet.
- A ***SetAttribDC()***: A CMetaFileDC m_hAttribDC adattagja alapértelmezésben 0, a ***SetAttribDC()***-vel adhatunk értéket neki.


Az 5.1. gyakorlat 5.1.2. és 5.1.3. szakaszában találunk példát az alkalmazásra.

5.2. A GDI (Graphics Device Interface)

A grafikus meghajtó interfész a grafikus függvények hívásához meghajtófüggetlen felületet biztosít. Ez azt jelenti, hogy a programozó egységesen kezelheti a különböző meghajtók felületét. Azonos módon írhatunk például a képernyőre, vagy a különböző típusú nyomtatókra. Továbbítja a grafikus

függvényeket a megfelelő meghajtó eszköznek (device driver). A meghajtóeszköz feladata, hogy meghívja a hardver specifikus kimeneti függvényeket. Az ablakosztályok grafikus függvényei a CDC osztály (Device Context) segítségével írnak a meghajtókra.

5.2.1. A GDI objektumok

A GDI objektumok példányai a CGdiObject osztálynak vagy utódosztályainak. (CBitmap, CBrush, CPen, CFont, CPalette, CRgn.) Az ablakokhoz hasonlóan minden **CGdiObject** objektum egy C++ objektum (az alkalmazásunk memóriaterületén) és egy system GDI object (az operációs rendszer memóriaterületén) együttesét jelenti. A C++ objektum tartalmaz egy **m_hObject** nevű tagváltozót, mely rámutat a system object struktúrára. Az operációs rendszer objektumai megoszthatóak, vagyis több alkalmazás is használhatja őket.

Az objektumok létrehozása történhet egy vagy két lépésben. Két lépés esetén először a C++ objektum jön létre konstruktora segítségével, második lépésben a Create fogja feltölteni az m_hObject mutatót. Pl:

```
CPen pen;
pen.CreatePen(PS_SOLID, 10, RGB(0,0,0));
// folytonos, 10 pixel vastag, fekete vonalat húz.
```

Egy lépésben is létrehozhatjuk pl:

```
CPen pen(PS_DOT,1,RGB(0,255, 0)
//Pontozott, 1 pixel vastag, zöld vonalat húz.
```

5.2.2. CPen

A toll inicializálásához a **BOOL CreatePen(int** *nPenStyle*, **int** *nWidth*, **COLORREF** *crColor* **);** tagfüggvényt használhatjuk. Első paramétere a toll stílusa, második a vonalvastagság pixelben, harmadik a szín.

A toll stílusai:

- **PS_SOLID**: folytonos vonal
- **PS_DASH**: szaggatott vonal
- **PS_DOT**: pontozott vonal
- **PS_DASDOT**: vonal, pont
- **PS_DASHDOTDOT**: vonal, pont, pont
- **PS_NULL**: üres toll
- **PS_INSIDEFRAME**: kitöltő szín, a toll színét a kitöltő színhez hasonlóan állítja elő. (Lásd színek, tiszta szín, 5.2.7.szakasz!)

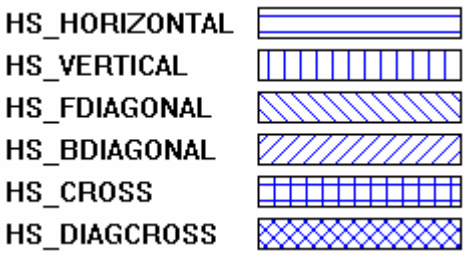
Üres toll választása esetén olyan alakzatokat rajzolhatunk, amelyeknek nincs határvonaluk. Ha ez esetben vonalat rajzolunk, az láthatatlan lesz.

További stílusok leírását a súgóban a CPen konstruktoránál találhatjuk meg.

5.2.3. CBrush

Az új ecset (**CBrush**) létrehozásához a következő lehetőségek állnak rendelkezésünkre:

- ***CreateSolidBrush()***: A paraméterben megadott színű ecsetet készít.
- ***CreateHatchBrush()***: Az első paraméterben megadott mintájú, a második paraméterben megadott színű ecsetet hoz létre. Mintái:



- ***CreatePatternBrush()***: A paraméterben megadott bitmap mintázatával hozza létre az ecsetet.

Megjegyzés:

A **CreateHatchBrush** metódus második paraméterében megadott szín a minta vonalainak színét jelenti (tiszta szín 5.2.7. szakasz). A minta hátterének színét a ***CDC::SetBkColor()*** metódussal állíthatjuk be.

5.2.4. CBitmap

- ***CreateBitmap()***: A paraméterben megadott méretű színválasztású, pontonként megadott értékű bitmapet hoz létre. Ritkán használjuk, helyette inkább a ***CreateCompatibleBitmap()*** (Lásd később!) vagy a ***LoadBitmap()*** használatosak.
- ***LoadBitmap():*** Inicializálja a bitmapet a paraméterében megadott erőforrás-azonosító vagy erőforrás fájlnév segítségével.

  A fájlból történő feltöltéshez jól használható a ***LoadImage()*** nevű API függvény, mely egyaránt alkalmas ikonok, kurzorok, animált kurzorok és bittérképek betöltésére. A szélesség és magasság paraméterének beállításával a fájlban tárolt képet adott méretűre konvertálja. NULL paraméterek esetén a kép

eredeti méretében töltődik be. Gyakorlati használatával a 3.3.3. feladat megoldásánál találkozunk.

A bittérkép megjelenítésére rendkívül sok lehetőség kínálkozik. A különböző raszterműveleti kódok segítségével a célterületen található képpont és a forrás bitmap képpontjából állíthatjuk elő a képet. Ezek közül a BitBlt függvény ismertetése a 5.3.2. szakaszban található.

5.2.5. CFont

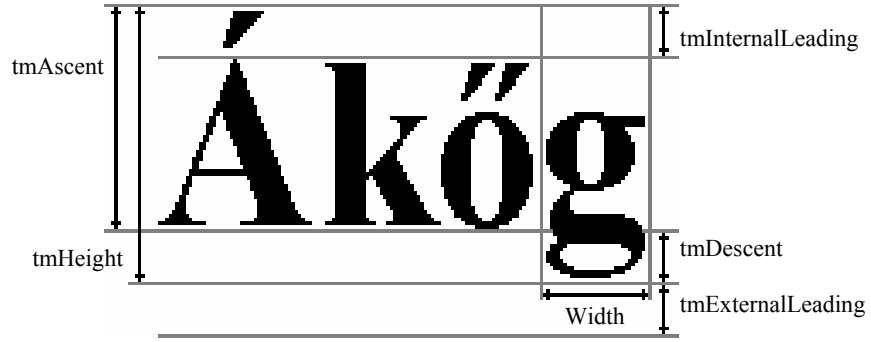
A **CFont** objektumot a ***CreateFont()***, a ***CreateFontIndirect()*** vagy a ***CreatePointFont()*** segítségével hozhatunk létre.

```
CFont font;
font.CreatePointFont(100, "Arial");      //(Size, FaceName)
```

A **LOGFONT** egy logikai font struktúra. Egy adott font adatait a ***CFont::GetLogFont()*** tagfüggvénnyel kérdezhetjük le. (További információk a súgóban.) A rendszerben létező fizikai font adatait a **TEXTMETRIC** struktúrában tároljuk. Az aktuális méretekre vonatkozó információkat a ***CDC::GetTextMetrics()*** tagfüggvénnyel kaphatjuk meg.

```
LOGFONT logfont;
font.GetLogFont(&logfont);
...
TEXTMETRIC tm;
pDC->GetTextMetrics(&tm);
lineHeight = tm.tmHeight + tm.tmExternalLeading;
```

A font adatainak jelentését a TEXTMETRIC struktúrában a 5.2. ábra mutatja.



5.2. ábra A font adatai

A különböző típusú fontok lehetnek proporcionálisak (**proportional font**), melyeknek cellaszélességük karakterenként változik, vagy fix szélességűek (**fixed pitch**), mint a Curier New.

Ez egy proporcionális font (Arial).
Előnye a kisebb helyfoglalás.

```
Ez egy fix szélességű font (Curier New).
Minden font mérete azonos.
```

Emiatt a TEXTMETRIC struktúra a szélességet két értékkel adja meg. A **tmAveCharWidth** az átlagos karakterszélesség, míg a **tmMaxCharWidth**-ből a maximális karakterszélességhez juthatunk hozzá. A **tmInternalLeading** a nagybetűk ékezeteinek helyét tartalmazza, míg a **tmExternalLeading** a sorközt biztosítja. A **tmHeight** a legnagyobb betű magassága, így ha hozzáadjuk a tmExternalLeading értékét, egy teljes sor magassága lesz az eredmény, amint azt az előző kódrészlet mutatja. Minden adat logikai egységekben adott, tehát függ a dc mapping mode-jától. (Lásd később!)

 A 6. gyakorlat 6.3. szakaszában találunk példát a font használatára.

5.2.6. CRgn

Egy sokszög, ellipszis vagy ezek kombinációjával létrehozott tartomány. A régió létrehozása is két lépésben történik. A konstruktor elkészíti az objektumot, a Create függvények pedig feltöltik az **m_hObject** adattagot. A ***CreateRectRgn(x1, y1, x2, y2)*** egy téglalap alakú, a ***CreateEllipticRgn(x1, y1, x2, y2)*** ellipszis formájú, míg a ***CreateRoundRectRgn(x1, y1, x2, y2, x3, y3)*** lekerekített sarkú, téglalap formájú régiót hoz létre. Az x3, y3 paraméterek a lekerekítés szélességét és magasságát adják. A ***CreatePolygonRgn(lpPoints, nCount, fillingMode)*** függvénnyel sokszög alakú régiót készíthetünk. Az lpPoints paraméter egy pontokból álló tömbre mutat. Az nCount a pontok száma a tömbben.

Régiót úgy is létrehozhatunk, hogy más régiók és halmazműveletek segítségével határozzuk meg a tartományt. A ***CRgn::CombineRgn(pRgn1, pRgn2, Mode)*** első két paramétere a műveletben résztvevő két régiót, a harmadik paraméter a műveletet határozza meg.

A Mode értékei:

- RGN_AND metszet
- RGN_OR unió
- RGN_DIFF különbség
- RGN_XOR kizáró vagy = unió \ metszet

A CRgn osztály **BOOL EqualRgn(CRgn*** *pRgn* **) const;** metódusa segítségével összehasonlíthatunk két régiót. Ha a régió, amire meghívtuk a függvényt, megegyezik a paraméterben megadott régióval, nem nulla a visszatérési érték,

egyébként nulla. A **BOOL PtInRegion(int** *x*, **int** *y* / **POINT** *point* **) const;** függvény pedig megadja, hogy a paraméter valamelyik formája által meghatározott pont eleme-e a régiónak. Visszatérési értéke nem nulla, ha eleme, egyébként nulla.

Megjegyzés:

A régió mérete maximum 32 767 logikai egység, vagy 64K memória lehet!

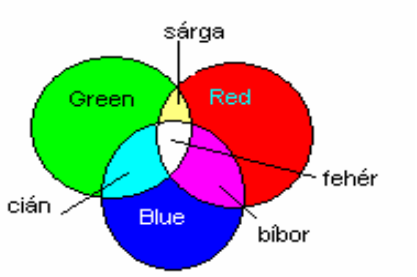
5.2.7. A színek

A színek beállítása az ún. **RGB**(red, green, blue) értékük segítségével történik. A (0,0,0) = fekete, a (255, 255, 255) = fehér szín. A **COLORREF** típust az RGB számhármassal állíthatjuk elő.

```
pDC->SetBkColor(RGB(0,255,0));
pDC->SetTextColor(255, 0, 0);
```

Az RGB színmodell:

Szín	**RGB érték**
piros	255, 0, 0
zöld	0, 255, 0
kék	0, 0, 255
cián	0, 255, 255
bíbor	255, 0, 255
sárga	255, 255, 0
fehér	255, 255, 255
fekete	0, 0, 0
sötét vörös	128, 0, 0
sötét zöld	0, 128, 0
sötét kék	0, 0, 128
sötét cián	0, 128, 128
sötét bíbor	128, 0, 128
sötét sárga	128, 128, 0
sötét szürke	128, 128, 128
világos szürke	192, 192, 192



5.3. ábra Az RGB színkeverés

A COLORREF 32 bitje 8 bitenként

Nem használt	Blue	Green	Red

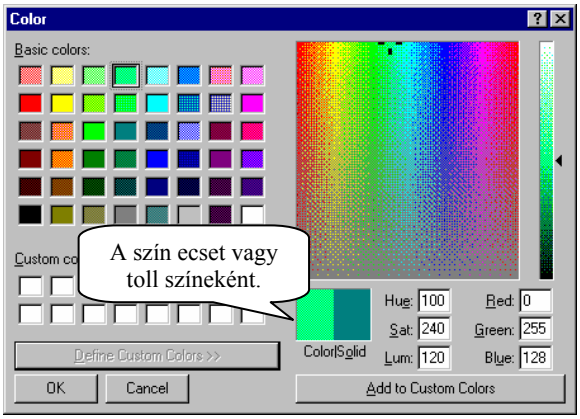
A párbeszédfelületek és vezérlők színe.

A COLORREF a színt a 0x00bbggrr hexadecimális formában tárolja. Értékét a 65536*blue + 256*green+ red értékből kapjuk.

A COLORREF színből az alapszíneket a ***GetRValue***(COLORREF) , ***GetGValue***(COLORREF) és ***GetBValue***(COLORREF) makrók segítségével kaphatjuk vissza.

A színek megjelenítését a videókártya lehetőségei és az aktuális paletta beállításai is befolyásolják.

Tiszta szín (pure color) **a videokártya által előállított szín.** Nem pixelek keveréke. Tollhoz csak tiszta színt rendelhetünk hozzá. Ha nincs az adott színnek megfelelő tiszta szín, akkor a vonal a legközelebbi videokártya színnel rajzolódik ki. Ez alól egyetlen kivétel van, ha a toll PS_INSIDEFRAME stílusú. Ekkor a színt egy speciális eljárással, az úgynevezett egyengetéssel (dithering) állítják elő a különböző színekből. (Young, 1998. II. 83.old.)



5.4. ábra A ColorDialog a Define Custom Colors része

A ColorDialog a Define Custom Colors részben mutatja a kiválasztott szín kevert színként vagy tiszta színként történő megjelenítését.

A háttér színének kirajzolását megakadályozhatjuk a ***CDC::SetBkMode()*** függvényének **TRANSPARENT** paraméteres hívásával. A háttérmód alapértelmezésben **OPAQUE**, ami a háttér aktuális háttérszínnel történő kitöltését jelenti.

5.2.8. Grafikus objektum törlése

Ne feledjük objektumainkat a ***CGdiObject::DeleteObject()*** függvényével törölni! A DeleteObject() nem törli a C++ objektumot, csak kinullázza az m_hObject adattagot. Ha egy PatternBrush-t törlünk, nem töröljük a hozzá kapcsolódó bitmap objektumot!

Ha a rendszerobjektum mutatóját meg akarjuk őrizni, és később újra felhasználni, használjuk a ***Detach()*** / ***Attach()*** függvénypárt! A Detach() visszatérési értéke az m_hObject system object mutató, mely az Attach() segítségével újra a C++ objektumhoz csatolható.

```
HGDIOBJ hPen;
hPen = pen.Detach();
...
pen.Attach(hPen);
```

Megjegyzés:

Az rendszer automatikusan törli az erőforrásait, ha a folyamat (process), amely létrehozta őket terminál. Ennek ellenére célszerű törölni az erőforrás objektumokat, ne foglaljanak a folyamat futása alatt sem fölöslegesen memóriát.

5.3. Rajzolás a képernyőre

5.3.1. A WM_PAINT üzenet és kezelése

Az üzenetet a Windows küldi el, ha újra kell festeni az ablak kliensfelületét. Csak akkor küldi el**, ha nincs más üzenet az alkalmazás üzenetsorában**. (Pihenőidős tevékenység, IdleTime process.) A nem kliens terület (menüsor, címsor) újrafestése a **WM_NCPAINT** üzenet feladata.

A WM_PAINT üzenetet a ***CWnd::OnPaint()*** metódusa kezeli, melyet az MFC biztosít számunkra.

A CView::OnPaint() feladata az ***OnDraw()*** metódus meghívása az aktuális dc-vel.

```
void CView::OnPaint()
{
CPaintDC dc(this);
OnPrepareDC(&dc);
OnDraw(&dc);
}
```

A CPaintDC csak akkor használható, ha már egy WM_PAINT üzenetre felelünk. Konstruktora meghívja a ***CWnd::BeginPaint()***, destruktora a ***CWnd::EndPaint()*** tagfüggvényt.

Az újrafestés üzenet hívásához jeleznünk kell az operációs rendszer számára, hogy az ablakot frissíteni kell. Ezt többnyire az ***Invalidate(BOOL bErase=TRUE)*** függvény hívásával tehetjük. Hatására a kliensterület hozzáadódik a frissítésre váró régióhoz. A bErase paraméter megadja, hogy a hátteret is frissítsük-e. Ha a kliensterület egy részét szeretnénk csak frissíteni, vagy a frissítését visszavonni, az

InvalidateRect() / ***ValidateRect()***, valamint az ***InvalidateRgn()*** / ***ValidateRgn()*** metódusokat használhatjuk. Paraméterük a bErase-en kívül a frissítendő téglalap vagy régió mutatója, mely hozzáadódik / kivonódik a frissítésre váró régióból.

Az érvénytelen terület (**invalid region**) meghatározza, hogy a meghajtófelület mely részét rajzoljuk újra. Célja, hogy optimalizáljuk a kódot, ne frissítsük mindig az egész kliensterületet!

Az OnPaint() függvényhívást az ***UpdateWindow()*** és a ***RedrawWindow()*** metódusokkal közvetlenül is kikényszeríthetjük.

5.3.2. A CDC osztály tagfüggvényei

A teljesség igénye nélkül, csak hogy érzékeljük lehetőségeinket, nézzük meg a CDC osztály néhány tagfüggvényét! A nevük alapján kitalálhatjuk mi a feladtuk.

GetBkColor( )	SetBkColor( )
GetTextColor( )	SetTextColor( )
TextOut( )	GetTextMetrics( )
SelectObject( )	SetPixel( )
Rectangle( )	Ellipse( )
FillRect( )	FrameRect( )
DrawIcon( )	GetCurrentBitmap( )
GetCurrentPosition( )	SetBkMode( )

Zárt alakzatok belsejének kitöltéséhez jól használható a FloodFill() metódus.

BOOL FloodFill(int *x*, **int** *y*, **COLORREF** *crColor* **);**

Kitölti az aktuális ecsettel az (x,y) pontot tartalmazó, crColor színnel határolt területet. Visszatérési értéke 0, ha nem sikerült befejezni a műveletet pl. mert nem zárt az adott színnel határolt terület, vagy ha a pont a megadott színű.

Említést érdemel még a bitmapek kirajzolását lehetővé tevő

BOOL CDC::BitBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **CDC*** *pSrcDC*, **int** *xSrc*, **int** *ySrc*, **DWORD** *dwRop* **);** tagfüggvény. Első négy koordinátája a célterület téglalapjának adatai, ezt a forrást tartalmazó DC mutatója követi. A hatodik és ahetedik adat a forrás bitmap bal felső sarkának koordinátái, míg az utolsó a megjelenítés operációja. A kirajzolás során a célterületen található kép és a forrás bitmap pontjaira ad meg műveleteket (és, vagy, kizáró vagy műveletek a bitmapen vagy inverzén). A dwRop értékeit keressük meg a súgóban! Az **SRCCOPY** érték egyszerűen kimásolja a forrás bitmapet a célterületre. A függvény és társai szinte korlátlan lehetőségeket biztosítanak számítógépes grafikák készítéséhez.

A CDC osztály további, a megjelenítést támogató függvényeit, valamint a fenti függvények paramétereit keressük meg az MSDN-ben!

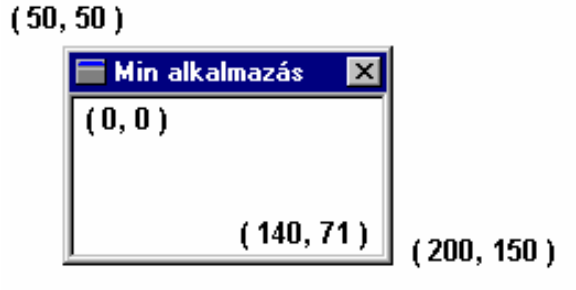
A függvényhívások során gyakran használjuk a következő osztályokat:

- **CPoint**: egy pont két koordinátája határozza meg. Tagváltozói az **x** és az **y**.
- **CRect**: egy téglalap felső, bal, alsó és jobb koordinátáit tárolja a top, left, bottom és right nevű tagváltozókban.
- **CSize**: a méret megadásakor használatos. Tagváltozói a szélesség és magasság adatokat szolgáltatják cx és cy néven.

Ahhoz, hogy adatokat írjunk ki a képernyőre, szükségünk lehet az aktuális értékek lekérdezésére. Ebben segít bennünket a CWnd osztály ***GetWindowRect()*** metódusa, mely a paraméterében megadott **LPRECT** mutató segítségével adja vissza – képernyőkoordinátákban – az aktuális ablak által meghatározott téglalapot. A ***CWnd::GetClientRect()*** tagfüggvény segítségével a kliensterület méretét kérdezhetjük le, mert a kliensterület téglalapjának koordinátáit adja vissza klienskoordinátákban. Ez azt jelenti, hogy a top, left adatok értelemszerűen 0 értékűek, a bottom, right adatok pedig gyakorlatilag a magasság és szélesség méretei (0, 0, cy, cx). A ***CWnd::ClientToScreen()*** függvény a paraméterében megadott pont vagy téglalap klienskoordinátáit átalakítja képernyő-koordinátákká. A CRect osztály **PtInRect()** tagfüggvényével kérdezhetjük le, hogy egy adott pont a téglalapban megadott területen van-e.

```
CRect rect, clientRect;
GetWindowRect(&rect);
GetClientRect(&clientRect);
```

Kódrészlet esetén, ha a **rect(top = 50, bottom = 150, left = 50, right = 200), clientRect(top = 0, bottom = 71, left = 0, right = 140)** értékeket kapjuk, azok a 5.5. ábrán jelölt pontok adatai. Az ablakon kívüli adatok képernyő-koordinátákban, míg az ablak belsejében levők klienskoordinátákban értendők.



5.5. ábra Az ablak méretadatai

A CWnd osztály ***MapWindowPoints()*** tagfüggvénye (**void CWnd::MapWindowPoints(CWnd*** *pwndTo*, **LPRECT** *lpRect* **) const;**) a második paraméterben megadott téglalap koordinátáit alakítja át az első paraméterben megadott ablak koordinátáira. Ha első paramétere NULL, a téglalapot képernyő koordinátákban kapjuk meg. A következő kódrészlet megmutatja, hogyan lehet méretezni egy **CListBox** listBox listadobozt a benne felsorolt elemek számától függően. A kód a listadoboz szülőablakának (this) egyik metódusában található.

```
CRect rect;
listBox.GetClientRect(&rect);
listBox.MapWindowPoints(this, &rect);
rect.bottom =
       rect.top+listBox.GetCount()*(listBox.GetItemHeight(0)+1);
```

Gyakran használjuk CRect objektumok esetén a ***Width()*** és ***Height()*** tagfüggvényeket, melyek a téglalap szélességét és magasságát adják vissza.

A ***GetSystemMetrics()*** API függvény adja vissza a paraméterében meghatározott rendszerinformációkat. Néhány paramétere:

SM_CMOUSEBUTTONS: esetén a visszatérési érték az egérgombok számát adja.

SM_CXBORDER,
SM_CYBORDER: az ablakkeret szélessége és magassága pixelekben.

SM_CXCURSOR,
SM_CYCURSOR: a kurzor szélessége és magassága.

SM_CYCAPTION: a szokásos címsor magassága pixelekben.

SM_CXMAXIMIZED,
SM_CYMAXIMIZED: a monitoron megjeleníthető maximalizált ablak méretei pixelekben.

SM_SHOWSOUNDS: visszaadott értéke nem 0, ha vizuálisan kell megjeleníteni az audió (hang) információkat.

További paraméterek a súgóban érhetőek el.

5.3.3. Logikai és fizikai megjelenítés

A megjelenítéskor koordinátákat használunk a pozicionáláshoz. Ahhoz, hogy mit jelentenek ezek a koordináták, meg kell adnunk a koordinátarendszer kezdőpontját, a tengelyek irányát, és az egységeket a koordinátarendszerben.

A GDI kétféle egységet használ a távolságok megadására. Az egyik a logikai egység, mely az eszköz felbontásától független távolságegységet jelent, a másik a fizikai egység, az eszközfelbontástól függ. Mivel leggyakrabban a képernyőn jelenítünk meg, ezt képpontnak pixelnek hívjuk.

A GDI három koordinátarendszer használ párhuzamosan. Az eszközkapcsolat koordinátarendszerét, melynek egységei fizikai egységek és az origó az eszközkapcsolat meghatározásától függ.

A rajzoló terület logikai koordinátarendszerét az ablakok (window) **adják, míg a fizikai megjelenítést a viewport végzi fizikai koordinátarendszerével.**

A fizikai koordinátarendszer mozgatható az eszközkapcsolat koordinátarendszeréhez képest. A logikai koordinátarendszer kezdőpontja a fizikai koordinátarendszerben definiálható. A logikai koordinátarendszer tehát a fizikaihoz képest mozgatható. Ha elmozgatjuk a fizikai koordinátarendszert, vele együtt mozog a logikai koordinátarendszer is. (Benkő, 1995, 285. old.)

A CDC biztosítja a függvényeket, melyek átalakítják a logikai adatokat az aktuális leképezési módnak (**mapping mode**) megfelelő fizikai értékekké.

Az origó (origin) a (0,0) koordinátájú pont helyét a ***CDC::SetWindowOrg()*** / ***SetViewportOrg()*** függvényekkel állíthatjuk be, s a ***CDC::GetWindowOrg()*** / ***GetViewportOrg()*** függvényekkel kérdezhetjük le. Az origó alapértelmezésben az ablak vagy képernyő bal felső sarka.

A leképezési mód meghatározza a logikai egység és a fizikai egység közti kapcsolatot és a tengelyeken a pozitív irányt. Az ablak méretét a ***SetWindowExt()*** CDC metódussal adhatjuk meg, visszaadott értéke CSize típusú, és az ablak előző méretét tartalmazza. Az aktuális méret a GetWindowExt() függvénnyel kérdezhető le. Ugyanez Viewport-ra: ***CDC::SetViewportExt()*** / ***GetViewportExt()***.

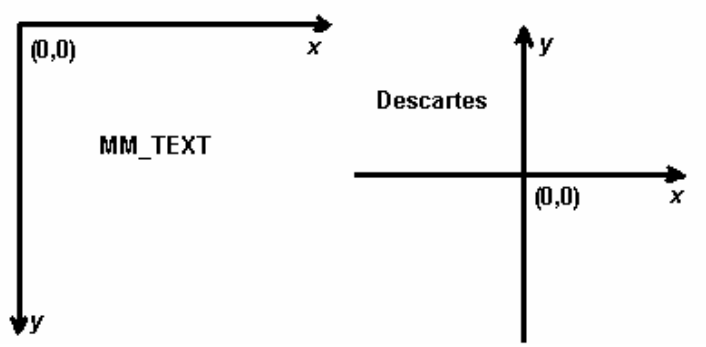
Megjegyzés:

> A rendelkezésünkre álló eszközökről és adataikról a ***CDC::GetDeviceCaps***() függvény szolgál információkkal. Paraméterei és visszatérési értékei a súgóban megtalálhatóak.

Az MM_TEXT az alapértelmezett leképezési mód. A legtöbb nyelv hagyományosan ilyen módon írja szövegeit balról jobbra, majd felülről lefelé. MM_TEXT esetén az x tengely jobbra növekszik, és az y tengely lefelé. Egy logikai egységet egy eszközpontra konvertál.

Más leképezési mód a ***CDC::SetMapMode()*** metódussal állítható be, és a ***CDC::GetMapMode()*** metódussal kérdezhető le.

Ha például egy hagyományos Descartes koordinátarendszerben kívánunk rajzolni, a képernyő közepére kell helyeznünk az origót, az x tengely jobbra növekedjen (ez az alapértelmezés is), az y tengely pedig fölfelé (ezt az irányt meg kell fordítanunk)!



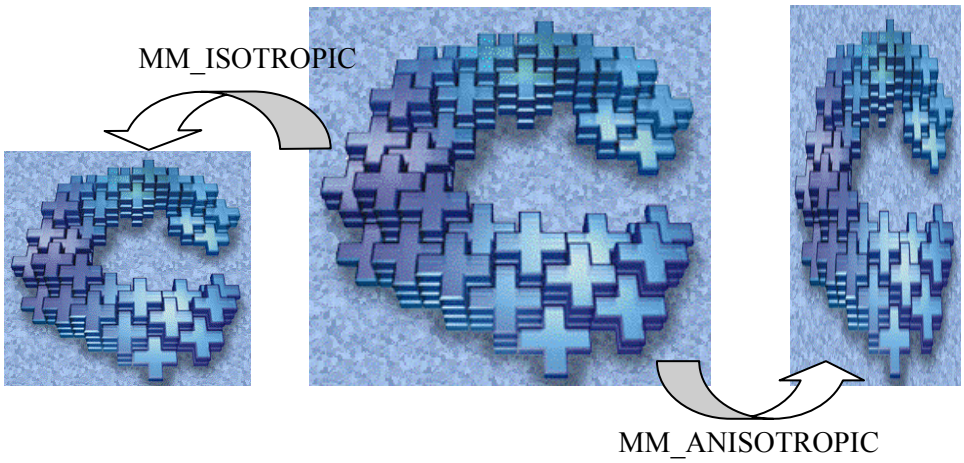
5.6. ábra A leképezési mód átalakítása

```
pDC->SetMapMode(MM_ISOTROPIC); // Magyarázat később.

CRect rect;
GetClientRect(&rect); // Az ablakméret lekérdezése.
pDC->SetViwportExt( rect.Width(), -rect.Height() )
// Az y tengely irányának megváltoztatása.
pDC->SetViewportOrg( rect.Width() / 2, rect.Height() / 2 );
//Az origó középre helyezése.
```

Az alapértelmezetten kívül rendelkezésünkre állnak még további leképezési módok is, melyek használata olyankor előnyös, ha lényeges, hogy egy adott ábra bármilyen eszközön azonos méretű legyen. Mindegyikre jellemző a pozitív x irány jobbra és pozitív y irány fölfelé.

- **MM_LOENGLISH**: egy logikai egységet 0.01 hüvelykre (inch) konvertál.
- **MM_HIENGLISH**: egy logikai egység 0.001 hüvelyk.
- **MM_LOMETRIC**: egy logikai egység 0,1 mm.
- **MM_HIMETRIC**: egy logikai egység 0,001 mm.
- **MM_TWIPS**: egy logikai egység 1/1440 hüvelyk

Mi magunk is készíthetünk saját leképezési módot. (Lásd az előző kód!) Ez kétféle lehet, **MM_ISOTROPIC** és **MM_ANISOTROPIC**. Az előző biztosítja a kép x és y koordináta egységének 1:1 arányát, tehát az ábra arányai megmaradnak, míg az anisotropic esetben torzíthatjuk a képet, az x és y koordináták mérete egymástól függetlenül változtatható.



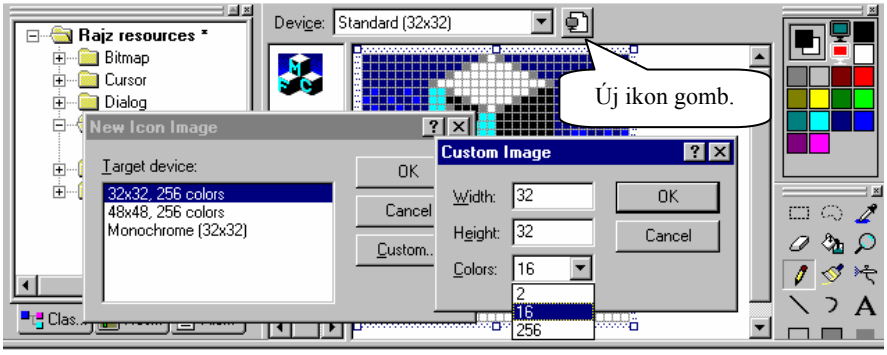
5.7. ábra Az általunk definiált leképezési módok

A ***CDC::LPtoDP()*** függvényei CPoint, CRect vagy CSize objektumokat konvertálnak át logikai koordinátákról az éppen beállított leképezési módnak megfelelő eszközkoordinátákra. A ***DPtoLP()*** függvények az ellenkezőjét teszik.

Ha eszköz független programot akarunk készíteni, akkor feladatainkat egy logikai koordinátarendszerben kell megfogalmaznunk, s a megvalósítást szabad csak az aktuális eszközre bíznunk.

5.3.4. Az ikonok

Az ikonok speciális bittérképek. Egy ikonfájl egy vagy több képet tartalmazhat, pl. 16x16 pixeles 16 színű, 32x32 pixeles 16 színű vagy 2 színű (monokróm) képet. De általunk beállított méretű és 256 színű ikonok is készíthetők az ikonszerkesztő eszközgombjának segítségével.



5.8. ábra Az ikonszerkesztő

Ha azt akarjuk, hogy az adott meghajtókörnyezetben megjelenjen az alkalmazás ikonja, el kell készítenünk a megfelelőt. Ezzel a problémával már találkoztunk az 1.1. gyakorlat, a Hello ikonjának elkészítése során.

Az ikonszerkesztő egy a bittérképeknél nem használatos lehetőséget kínál ikonjaink megrajzolásához. A kiválasztható színek közt két képernyő formájú jelet is találunk, az egyik sötétzöld, míg a másik narancsszínű. A zölddel festett területe az ikonnak a háttér színét, míg a narancsra festett a háttér inverz színét veszi fel. Így elérhetjük, hogy ikonjaink szinte tetszőleges háttér esetén jól láthatóak legyenek a képernyőn.

Az ikonokat .ico kiterjesztésű fájlokban tárolhatjuk.

5.4. Az egérkurzor a képernyőn

Különböző tevékenységek alatt az egérkurzor alakjának megváltoztatásával jelezzük, hol tartunk a folyamatban.

A legegyszerűbb a homokórakurzor létrehozása. Ezt hosszantartó tevékenységek alatt szokás megjeleníteni. A homokóra kurzor a ***BeginWaitCursor()*** metódus hatására jelenik meg, és az ***EndWaitCursor()*** metódus hatására vált vissza az eredeti kurzorformára. Mivel a CCmdTarget osztály tagfüggvénye, minden utódosztályából hívható.

```
BeginWaitCursor();
//Tevékenység elvégzése.
EndWaitCursor();
```

Homokórakurzort a **CWaitCursor** osztály objektuma segítségével is létrehozhatunk. Csak fel kell vennünk egy ilyen lokális objektumot a függvénynek azon a pontján, ahol a hosszantartó tevékenység elkezdődik. Az objektum konstruktora hívja a BeginWaitCursor, destruktora pedig az EndWaitCursor metódust.

5.4.1. Tetszőleges alakú kurzor

- A kurzor alakjának megváltoztatásához szükségünk lesz egy kezelőre (handle).
  ```
  HCURSOR m_hCross;
  ```
- Az m_hCursor a ***CWinApp::LoadStandardCursor()*** segítségével kaphatja a standard kurzorazonosítók értékét, vagy a ***CWinApp::LoadCursor()*** függvénnyel általunk készített, és a Cursor erőforrások közé felvett kurzor értékét veheti fel.
  ```
  m_hCursor = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
  ```
  vagy:
  ```
  m_hCursor = AfxGetApp->LoadCursor(IDC_MYCURSOR);
  ```

- A kurzor is egy Windows erőforrás, használata után fel kell szabadítani az általa használt memóriaterületet a ***DestroyCursor()*** függvénnyel. Paramétere a kurzor kezelője. DestroyCursor hívásakor a kurzor nem lehet használatban! Ne használjuk a rendszer által megosztott kurzorok felszabadítására!

- Az esemény kezelőfüggvényéből a ***SetCursor***(m_hCursor) függvényhívással választhatjuk ki az általunk kívánt kurzort. Visszaadott értéke az előzőekben használt kurzor kezelője.

- Az alkalmazásunk beállítja a kurzort minden alkalommal, amikor az egy ablak fölé ér. A rendszer visszaállítja a kurzort az egérüzenetek kezelése alkalmával, pl. minden egérmozgatáskor. Mindezt az adott ablak **WM_SETCURSOR** üzenetéhez rendelt **OnSetCursor()** metódus végzi. (Lásd 5.4.4. szakasz!)

- Az ablakhoz tartozó egérkurzort az ablak létrehozása előtt, a **CREATESTRUCT** struktúra **lpszClass** adattagjában állíthatjuk be. Az lpszClass **WNDCLASS** struktúra típusú és **HCURSOR hCursor** adattagja az osztály kurzorkezelőjét adja meg. Az osztálynév beállítását legegyszerűbben az ***AfxRegisterWndClass()*** függvény segítségével végezhetjük el. (Gyakorlati alkalmazása a 6.6.4. feladat megoldásában található.)

Standard kurzorazonosítók:

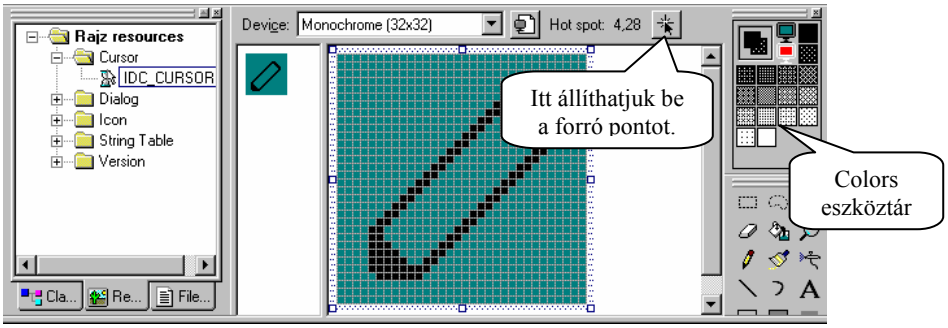
IDC_ARROW	Nyíl	
IDC_CROSS	Célkereszt	
IDC_IBEAM	I alakú	
IDC_SIZEALL	Négy irányú nyíl	
IDC_SIZENS	É-D irányú nyíl	
IDC_SIZENWSE	ÉNY-DK irányú nyíl	
IDC_SIZENESW	ÉK-DNY irányú nyíl	
IDC_SIZEWE	Ny-K irányú nyíl	
IDC_UPARROW	Felfelé nyíl	

Az erőforrás-szerkesztőbe importálhatunk más fájlban tárolt kurzorokat is.

5.4.2. Általunk rajzolt kurzor

Új erőforrásként kurzort is felvehetünk az erőforrás-szerkesztőben. Ugyanúgy szerkeszthetjük, mint bármelyik képet. Amire figyelni érdemes, az az ikonokhoz hasonlóan lehetővé teszi a háttérszínhez való alkalmazkodást. A Colors eszköztárban a zöld monitort választva az adott terület átlátszó lesz, tehát mutatja a

kurzor mögötti képet, míg a piros monitor a háttérben jól látszó színt választ a kép adott részének kifestésekor.



5.9. ábra Új kurzor az erőforrás-szerkesztőben

A kurzor a jó láthatóság miatt nem egy pixel méretű, így képének leírásakor fontos szerep jut a forró pontnak (**hot spot**), mely megadja, hogy a kurzor épp melyik pontra mutat.

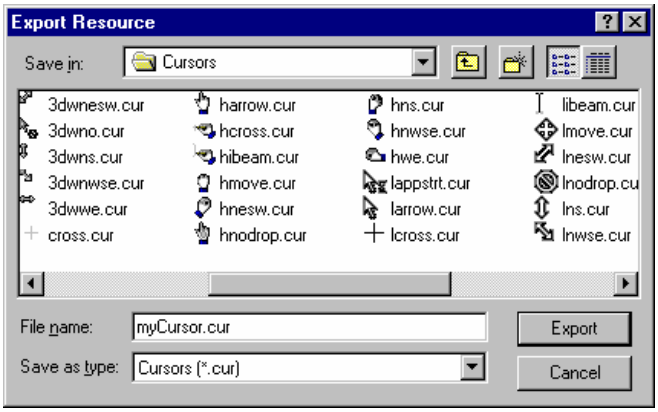
Megjegyzés:

A hot spot valójában forró foltot jelent, s megadja azt a területet, amely reagál, ha az egeret fölé visszük. Pl. hypertext szövegeknél e terület fölött szokott a kurzor kéz alakúra változni. A mi beállításunk azt adja meg, hogy a rendszer a kurzorkép mely pontjának koordinátái alapján dönti el, hogy a kurzor az adott terület fölött van-e.

Az ily módon megrajzolt kurzor a ***LoadCursor()*** függvény segítségével tölthető be.

Ha új kurzort készítünk, a Visual C++ környezet a projekt res alkönyvtárába menti el cursor1**.cur** néven. Ha az első mentés előtt átnevezzük az azonosítóját, az automatikusan a fájl nevét is átírja. Később a tulajdonságlapon adhatunk új nevet kurzorunknak.

A kurzorkép természetesen tetszőleges helyre a projekten kívül is exportálható. Az azonosítóján jobb egérgombbal kattintva, a felbukkanó menüből az Export menüpontot választva .cur kiterjesztésű fájlba menthetjük kurzorunkat.



5.10. ábra A kurzor exportálása

5.4.3. Animált kurzor

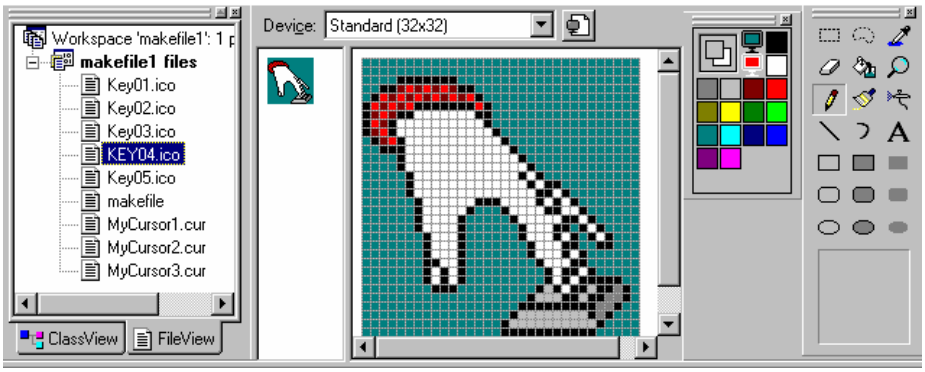
Kurzort a ***LoadCursorFromFile()*** függvény segítségével is betölthetünk. A függvény paramétere a fájl elérési útja. **Animált kurzor** fájljának kiterjesztése **.ani**, animált kurzor betöltéséhez tehát a függvénynek egy ilyen fájl elérési útját kell megadnunk.

```
SetCursor(LoadCursorFromFile("E:\\WINNT\\Cursors\\banana.ani"));
```

Animált kurzor csak olyan monitorok esetén alkalmazható, melyek támogatják a DIB (device-independent bitmap) megjelenítést. (VGA nem támogatja a színes animált kurzorokat!)

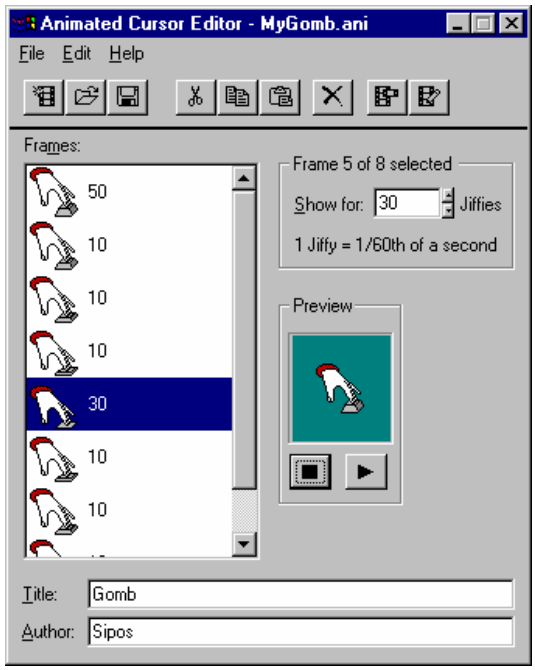
Saját animált kurzort készíthetünk az MSDN SDK-ban biztosított **AniEdit program segítségével**. (MSDN / Index fül / AniEdit kulcsszó.)

A leírás alapján: A File menü Open Workspace parancsával nyissuk meg a make fájlt! A Visual C++ elkészíti a projektet. Állítsuk a Project / Settings-ben (a General és a Debug fülben egyaránt) az exe fájl nevét AniEdit.exe-re! Ezen kívül adjuk meg a Project / Settings / Debug fül / Remote executable path and filename szövegmezőben a telepítésnek megfelelő útvonalat a Visual C++ bin alkönyvtárához! Készítsük el az AniEdit.exe fájlt a Build menü segítségével!



5.11. ábra Ikonok készítése az animált kurzorhoz

- A Visual C++ környezet erőforrás-szerkesztőjében készítsük el a képkockákat! (File / New / Files / Cursor File vagy ikon.) (5.11. ábra)



5.12. ábra Animált kurzor készítése az AniEdit eszköz segítségével

- Az AniEdit.exe File menü / Import Frame menüpontja segítségével fűzzük fel őket a végrehajtás sorrendjében! A Jiffies ablakban beállíthatjuk, hogy az egyes képkockák mennyi ideig (1Jiffy = 1/60s) szerepeljenek a lejátszásban. A

Preview segítségével kipróbálhatjuk az animált kurzor képét, s a File menüben elmenthetjük az alkalmazásoknál megszokott módon. (5.12. ábra)

Ha az imagedit SDK eszközt is elkészítjük és az AniEdit.exe-vel egy alkönyvtárba másoljuk, lehetőségünk lesz tetszőleges betöltött animált kurzor képeinek módosítására is. Természetesen új képkockák befűzése, vagy a sorrend megváltoztatása is módunkban áll.

Megjegyzés:

A kurzor és az ikon erőforrások hasonlóak, és sok esetben felcserélhetően használhatóak. Csak arra kell figyelni, hogy a kép formátumát a meghajtó támogassa. Pl. VGA monitor csak monokróm kurzort tud megjeleníteni.

5.4.4. A kurzort kezelő függvények

Az eddigiekben a kurzor betöltésével (LoadCursor) és kiválasztásával (SetCursor) foglalkoztunk. A kurzor kezeléséhez más API függvények is rendelkezésünkre állnak:

- ***GetCursor()***: visszatér az aktuális kurzor kezelőjével.
- ***GetCursorPos(LPPOINT lpPoint)***: a paraméterben átadott pontba beolvassa a kurzor aktuális koordinátáit képernyő-koordinátákban.
- ***SetCursorPos(int x, int y)***: beállítja a kurzort a paraméterben megadott koordinátákra.
- ***int ShowCursor(BOOL bShow)***: visszaad egy megjelenítés számlálót. Ha TRUE-val hívjuk, eggyel növeli, ha FALSE-szal, eggyel csökkenti az értékét. 0, vagy nagyobb értéknél látszik a kurzor. Kezdőértéke 0, ha nincs egér telepítve: értéke -1. Természetesen ki- és bekapcsolgatásra ugyanúgy lehetőségünk van, mintha egy logikai érték határozná meg a kurzor megjelenítését. Arra azonban ügyeljünk, hogy többször egymás után kikapcsolva, bekapcsolnunk is ugyanannyiszor kell, ha látni akarjuk a kurzort!

A CWnd osztály kurzorkezelő függvényei:

- ***CWnd::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)***: feladata a **WM_SETCURSOR** üzenet kezelése. Ablakokban felülírható. Ha ősosztályának azonos nevű tagfüggvényét meghívjuk (alapértelmezett megvalósítás), az visszaállítja az ősosztályban megadott kurzort.

  Paraméterei a kurzort tartalmazó ablak mutatója, a kurzor helyét meghatározó kód (pl. **HTCLIENT** a kliensterületen) és az egérüzenet azonosítója.

  Ha a második paramétere **HTERROR,** és egérgomb lenyomás az üzenet, akkor a ***MessageBeep*** függvényt hívja.

- ***CWnd::CreateCaret(CBitmap* pBitmap),*** ahol a bitmap magadja a kurzor képét. ***CWnd::CreateSolidCaret(int nWidth, int nHeight).*** Ha (0,0) paraméterrel hívjuk, akkor a rendszer által definiált ablakkeret szélesség, magasság értékét veszi fel. Téglalap (vonal) alakú kurzort definiál.
- ***CWnd::SetCaretPos()*** / ***GetCaretPos()*** beállítja a kurzor pozíciót / visszatér a kurzorpozícióval.

  A létrehozott kurzor kezdetben nem látható.
- ***CWnd::ShowCaret()*** / ***HideCaret()***: láthatóvá teszi a kurzort / elrejti a kurzort.

Szövegkurzor készítése ablakban:

```
CClientDC clientDC;
TEXTMETRIC tm;
ClientDC.GetTextMetrics(&tm);
CreateSolidCaret(tm.tmAveCharWidth / 3,
                            tm.tmHeight+tm.tmExternalLeading);
SetCaretPos(0,0);
ShowCaret();
```

Képkurzor a szerkesztőmezőben:

```
CBitmap* pBitmap = new CBitmap;
pBitmap->LoadBitmap(IDB_BITMAP);
m_EditCtrl.CreateCaret(pBitmap);
m_EditCtrl.ShowCaret();
```

Megjegyzés:

Ablakhoz alapértelmezett kurzor hozzárendelésére láthatunk példát a 6.6.4. feladat megoldásánál.

5.4.5. Az egér mozgásának korlátozása

A ***ClipCursor()*** függvény az egér mozgását a paraméterben megadott téglalapba zárja. A téglalap adatai képernyő-koordinátákban értendők. A ClipCursor(NULL) megszünteti a bezárást.

```
CRect rect;
GetClientRect(&rect);
ClientToScreen(&rect);
ClipCursor(&rect);
// További műveletek a kliensterületen
ClipCursor(NULL);
```

5.5. Kérdések

1. Ismertesse a CGdiObject osztályt, példányainak létrehozását és megszüntetését!
2. Mi a feladata a CPen, a CBrush, a CFont és a CRgn objektumoknak?

3. Hogyan kaphatunk COLORREF típusú változót? Mi az RGB színmodell?
4. Mi a DC feladata, hogyan változtathatjuk meg a rajzolás vagy a háttér színét?
5. A metódusok kódjának írásakor hogyan kaphatjuk meg az aktuális DC mutatóját?
6. Milyen függvényekkel kérdezhetjük le az ablakok méretét?
7. Melyik esemény hatására jelennek meg adatok a képernyőn? Mikor keletkezik ez az esemény?
8. Ismertesse a CDC osztály rajzoláshoz használt tagfüggvényeit s a leggyakrabban használt paraméter típusokat!
9. Hogyan hozhatunk létre homokórakurzort?
10. Hogyan választhatunk a már rendelkezésünkre álló kurzorok közül?
11. Hogyan hozhatunk létre saját kurzort?
12. Melyik függvénnyel tölthetjük be az animált kurzort? Hogyan készíthetünk animált kurzort?
13. Ismertesse a kurzort kezelő függvényeket!
14. Hogyan szoríthatjuk az egér mozgását egy téglalap alakú területre?

5.6. Összefoglalás

- **A grafikus meghajtó interfész a grafikus függvények hívásához meghajtófüggetlen felületet biztosít**. Azonos módon írhatunk például a képernyőre vagy a különböző típusú nyomtatókra.
- A GDI objektumok példányai a CGdiObject osztálynak vagy utódosztályainak. CBitmap, CBrush, CPen, CFont, CPalette, CRgn. Minden **CGdiObject** objektum tartalmaz egy **m_hObject** nevű tagváltozót, melyet az objektum Create tagfüggvénye tölt fel a system object struktúra mutatójával. Az objektumok elkészítésekor a **Create függvények paramétereiben határozzuk meg az objektumok tulajdonságait.** (Pl. a toll vastagságát, az ecset mintázatát vagy font méretét.)
- A színeket a függvények argumentumaiban COLORREF típusként kell megadnunk. **A COLORREF előállítható az RGB(red,green,blue) függvénnyel**, ahol mindhárom színt egy 0 és 255 közötti értékkel adhatjuk meg.
- A **DC** (**device context**) egy leíró. A GDI számára azokat az információkat tartalmazza, melyek az éppen használatos környezetben történő megjelenítéshez szükségesek. **A CDC osztálynak két rendszermeghajtó kezelő adattagja van, az m_hDC és az m_hAttribDC.** A CDC a kimeneti hívások többségét az m_hDC-nek küldi el, a beviteli hívások többségét pedig az m_hAttribDC-ből

olvassa ki. A **CClientDC dc(this) konstruktorral, a *GetDC() vagy a GetWindowDC()*** függvényekkel kérdezhetjük le.

- **A CDC egy konténer osztály. Elemei CGdiObject objektumok.** A GDI objektum létrehozása után a ***SelectObject()*** metódussal választható ki a CDC új objektuma. A Windows biztosít néhány GDI objektumot készen (stock object). Pl. **GRAY_BRUSH, BLACK_PEN, SYSTEM_FONT, DEFAULT_PALETTE.** Ezek az objektumok a ***CDC::SelectStockObject()*** paramétereként választhatóak ki.
- **A WM_PAINT üzenetet a rendszer küldi el**, ha újra kell festeni az ablak kliens felületét. Az újrafestés pihenőidős tevékenység (IdleTime process). Az üzenetet a ***CWnd::OnPaint()*** metódusa kezeli. Az OnPaint() függvényhívást az ***UpdateWindow()*** és a ***RedrawWindow()*** metódusokkal közvetlenül is kikényszeríthetjük.
- A CDC osztály tagfüggvényei segítségével rajzolhatunk, írhatunk a képernyőre, nyomtatóra. Ilyen metódusok pl. a TextOut( ), SetPixel(), Rectangle(), FillRect(), DrawIcon(),GetCurrentPosition(). A függvények paraméterezésekor gyakran használjuk a **CPoint, CRect, CSize** osztályokat.
- Az ablakok méretadatait a ***CWnd::GetWindowRect(), CWnd::GetClientRect(), CWnd::ClientToScreen()*** tagfüggvényekkel kérdezhetjük le. A rendszeradatokat a ***GetSystemMetrics()*** API függvény adja vissza.
- **A leképezési mód meghatározza a logikai egység és a fizikai egység közti kapcsolatot és a tengelyeken a pozitív irányt**. **Az MM_TEXT az alapértelmezett leképezési mód.** Más leképezési mód a ***CDC::SetMapMode()*** metódussal állítható be, és a ***CDC::GetMapMode()*** metódussal kérdezhető le.
- A homokórakurzor a ***BeginWaitCursor()*** metódus hatására jelenik meg, és az ***EndWaitCursor()*** metódus hatására vált vissza az eredeti kurzorformára. Tetszőleges alakú kurzort is megjeleníthetünk a ***SetCursor***() API függvény segítségével. A Windows kurzorok standard kurzorazonosítókkal érhetők el. Mi is készíthetünk kurzort az erőforrás-szerkesztő segítségével. A kurzorokat .cur kiterjesztésű fájlokban tároljuk.
- **Animált kurzor** fájljának kiterjesztése **.ani. Saját animált kurzort készíthetünk az** MSDN SDK-ban biztosított **AniEdit program segítségével**.
- A kurzor kezeléséhez további API függvények használhatók. A CWnd osztály ***OnSetCursor()*** metódusa kezeli a **WM_SETCURSOR** üzenetet. A ***CWnd::CreateCaret(CBitmap* pBitmap)*** segítségével képből készíthetünk kurzort, míg a ***CWnd::CreateSolidCaret()*** tetszőleges méretű, téglalap alakú kurzort hoz létre. A ***CWnd::HideCaret()*** elrejti, a ***ShowCaret()*** láthatóvá teszi a kurzort.
- A ***ClipCursor()*** függvény az egér mozgását a paraméterben megadott téglalapba zárja.

6. SDI, MDI alkalmazások

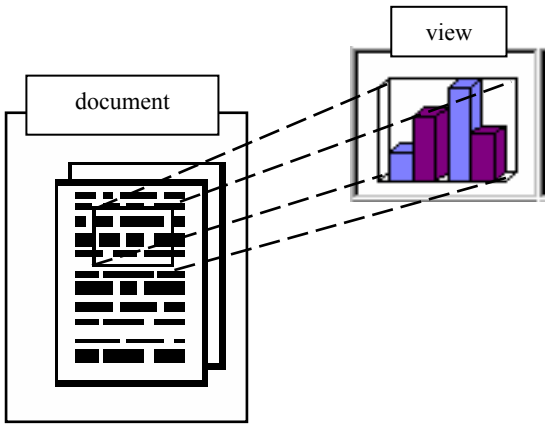
Az eddigiek során, egyszerűen a jobb áttekinthetőség kedvéért párbeszédalapú alkalmazásokkal dolgoztunk, de már többször említettük, főként az MFC-vel foglalkozó fejezetben, a dokumentumalapú alkalmazások néhány osztályát.

Párbeszédalapú alkalmazások esetén a CMyDialog osztály objektuma a főablak, ő végzi az eseménykezelés irányítását, tárolja az adatokat, megjeleníti azokat és vezérli a vezérlőkön keresztül a programot. A CMyApp osztály elindítja az alkalmazást, biztosítja a fő szálat. Az alkalmazásvarázsló által generált "csontváz" (skeleton files) e két fő osztályt tartalmazza. (Nem feltétlen szükséges a CAboutDlg osztály generálása, s ez az osztály nem játszik fontos szerepet alkalmazásunk szerkezetében.) Ezen kívül az általunk létrehozott osztályokkal dolgozunk, esetleg a vezérlőkhöz hozzárendelt, az osztályvarázsló által generált úgynevezett **wrapper** (csomagoló) **osztály**okkal, melyek szerepét és kapcsolatát a többi osztállyal látjuk, hisz mi magunk hoztuk őket létre.

Leggyakrabban használt **alkalmazásaink** viszont **jelentős számú adattal rendelkeznek, melyeket különböző nézetablakokon szeretnénk megjeleníteni**. Több funkciót kell megvalósítanunk, ezek mindegyikét nem tehetjük gombokra, **legördülő menükre van szükségünk** a tárolásukhoz. **A felhasználók munkáját kényelmessé teszi az eszköztárak használata, s megszoktuk, hogy a státuszsor is tájékoztat bennünket az alkalmazás aktuális állapotáról.** Az ilyen összetettebb rendszerek kezeléséhez kínálja a Visual C++ a Document / View architektúrát.

6.1. A Document / View architektúra

A Document / View architektúra olyan szerkezet, mely **két alaposztály, a dokumentumosztály és a nézetosztály együttműködésére épül. A View osztály felel az adatok megjelenítéséért, míg a dokumentumosztály az adatok tárolásáért.** Azért választottuk ketté az adatokat a megjelenítéstől, mert ugyanazokat az adatokat többféleképp is meg akarjuk mutatni, lehetővé akarjuk tenni módosításukat. Az elsődleges szerep a nézetosztályé, a **WYSIWYG** (What You See Is What You Get) **elv**nek megfelelően. A dokumentumosztály a háttérben működik, például biztosítja az adatok tárolásán kívül azok beolvasását, illetve mentését is.



6.1. ábra A dokumentumobjektum egy része látható

A nem dokumentum / nézet alapú alkalmazásoknál nincs szükség e két funkció szétválasztására. Párbeszédalapú (dialog based) alkalmazások esetén az adatok tárolását és megjelenítését is párbeszédablakok végzik. Például egy tömörítő vagy egy fájlkereső program esetén jobban megfelel a megvalósításnak egy párbeszédfelület.

6.1.1. SDI, MDI alkalmazások

A Dokument / View architektúra két eleme az **SDI** (**Single Document Interface**) és az **MDI** (**Multiple Document Interface**).

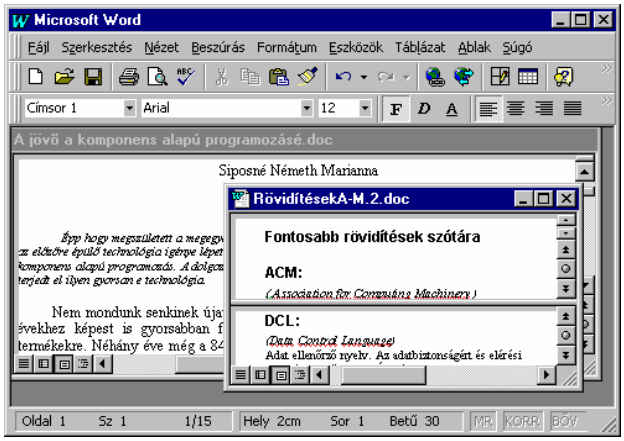
Az SDI alkalmazások egyszerre csak egy dokumentumot tudnak megnyitni. Lehetőségünk van az alkalmazásban különböző típusú dokumentumok kezelésére, de egyszerre csak egyet tudunk megnyitni közülük. SDI alkalmazás például a kellékek eszköztár Paint vagy PaintBrush alkalmazása. (6.2. ábra) Ha új ábrát szeretnénk megnyitni vagy létrehozni, az alkalmazás előbb bezárja a régit, s csak

utána nyitja meg az újat. Egyszerre több ábrában csak az alkalmazás több példányának párhuzamos futtatásával lehet dolgozni.

6.2. ábra Egy SDI alkalmazás, a Paint

Az SDI alkalmazás ugyanannak a dokumentumnak több nézetét is kínálhatja, tehát több nézetablakot is kezelhet, az 'egyetlen' (single) jelző a dokumentumra, tehát az adatokra vonatkozik.

Az MDI alkalmazás, mint a nevéből kiderül, több dokumentum párhuzamos kezelésére készült. E dokumentumok lehetnek azonos típusúak, de lehetnek különbözőek is. Például az Office Word alkalmazása egyszerre több különböző dokumentumot képes kezelni. (6.3. ábra)



6.3. ábra Egy MDI alkalmazás, a Microsoft Word

Az 6.3. ábra mutatja, hogy egyszerre két dokumentum (A jövő a komponens alapú programozásé és a RövidítésekA-M.2) lett megnyitva, ebből a Rövidítéseknek két nézetét is láthatjuk. MDI alkalmazás esetén az osztott nézetek közös keretben, míg az önálló nézetek önálló gyermek-keretablakokban találhatóak.

6.1.2. A Document / View architektúra osztályai

- Az **alkalmazásosztály** (application class): a **CWinApp** utóda.
- A **dokumentumsablon-osztály** (document template class): a **CDocTemplate** utóda.
- A **keretablakosztály** (frame window class): a **CFrameWnd** utóda.
- A **nézetosztály** (view class): A **CView** utóda.
- A **dokumentumosztály** (document class): A **CDocument** utóda.

Az alkalmazás osztályai többnyire nem közvetlen utódosztályai a fent említett MFC osztályoknak.

6.1.2.1. Az alkalmazásosztály

Az alkalmazásvarázsló által létrehozott CMyApp osztály a **CWinApp** osztály utóda. **Feladata az alkalmazás inicializálása, a fő szál elindítása, az alkalmazás üzenetkezelésének biztosítása és Document / View alkalmazás esetén a dokumentumsablon-szerkezet felépítése**.

Egy SDI alkalmazás, a ***CMyApp::InitInstance()*** metódusában a következő kódsorokat találjuk:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CMyDoc),
        RUNTIME_CLASS(CMainFrame),      // main SDI frame window
        RUNTIME_CLASS(CMyView));
AddDocTemplate(pDocTemplate);
```

A dokumentumsablon (pDocTemlate) hozza létre a dokumentumosztály objektumát, a fő keretablak erőforrásai (**IDR_MAINFRAME**) segítségével a keretablakosztály objektumot és a nézetosztály objektumot. Az ***AddDocTemplate()*** segítségével fűzi fel őket a dokumentumsablonok listájára.

Vegyük észre! A RUNTIME_CLASS makró a dokumentumsablon konstruktorának az általam írt osztályokról információkat ad át. A CRuntimeClass információk segítségével az MFC-ben megírt és már régen lefordított függvények az általam most létrehozott osztályok objektumait készítik el.

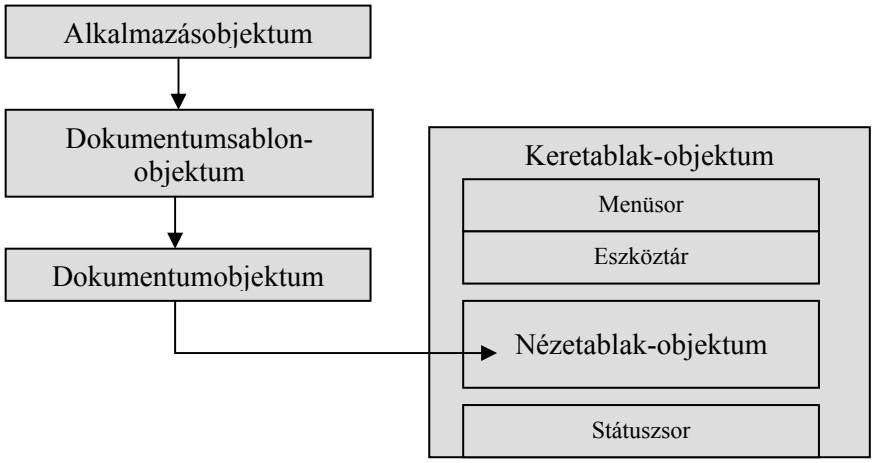
6.1.2.2. A dokumentumsablon-osztály

A dokumentumsablon összefogja egy dokumentum megnyitásához szükséges információkat: az erőforrásokat, a dokumentum osztályát, a keretablak osztályt és a dokumentumhoz tartozó alapértelmezett nézetosztályt.

Irányítja a dokumentumok, nézetek (viewk) és keretablakok létrehozását. Ha több különböző típusú dokumentumot tud kezelni az alkalmazás, azok mindegyikéhez külön dokumentumsablon tartozik.

Ősosztálya a **CDocTemplate**, SDI alkalmazás esetén **CSingleDocTemplate** típusú (lásd fenti kód), MDI alkalmazás esetén **CMultiDocTemplate** típusú.

Az alkalmazásobjektum hozza létre és irányítja az összes, az alkalmazást támogató dokumentumsablont.



6.4. ábra Az SDI alkalmazás objektumai és kapcsolatuk

6.1.2.3. A keretablakosztály

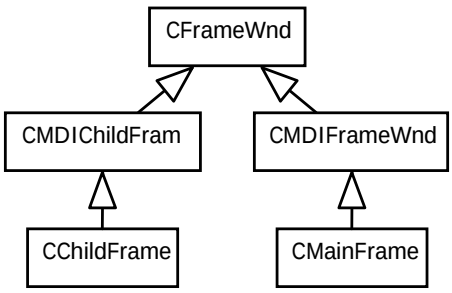
A keretablak feladata az alkalmazás keretének kezelése: címsor, menüsor, eszköztár, státuszsor, gördítősávok. Az 6.3. fejezetben részletesen foglalkozunk ezekkel az interfészekkel.

A fő keretablakosztály neve az alkalmazásvarázsló által létrehozott alkalmazásokban a **CMainFrame**. Ez SDI alkalmazás esetén közvetlenül a **CFrameWnd** osztályból származik, míg MDI alkalmazás esetén a **CMDIFrameWnd** osztályból.

SDI alkalmazás esetén az egyetlen dokumentum nézetei körül találjuk a keretablakot. Ez a keretablak több különböző nézetablakot tartalmazhat, mint gyermekablakot.

MDI alkalmazásoknál az alkalmazásnak van egy fő keretablaka (CMainFrame), és minden nézet rendelkezik egy saját keretablakkal. A dokumentumok keretablakai a **CMDIChildWnd** utódosztályai. Az alkalmazás menüsorát, eszközsorát, státuszsorát a fő keretablak kezeli, de a dokumentumokhoz tartozó keretablakok is tartalmazhatnak ilyen elemeket. Mozaikszerű elrendezés

esetén például a dokumentumok keretablakai önálló címsorral rendelkeznek (6.3. ábra), míg ha a teljes ablak állapotot választjuk, a fő keretablak címsora mutatja az aktuális dokumentum nevét.



6.5. ábra Az MDI alkalmazás keretablakai

A keretablakban az aktuális nézetobjektumot a ***GetActiveView()*** metódus segítségével érhetjük el. A függvény egy CView* típusú mutatóval tér vissza, melyet szükség esetén konvertálni (cast) kell a megfelelő osztályra. Az aktuális dokumentumot a ***GetActiveDocument()*** metódus segítségével kaphatjuk meg, mely CDocument típusú mutatót ad vissza.

6.1.2.4. A nézetosztály

Feladataa dokumentumosztályban tárolt adatok megjelenítése.

Ősosztálya a **CView** vagy utódosztályai:

- **CScrollView**: görgethető ablak.
- **CFormView**: a párbeszédfelülethez hasonló.
- **CHtmlView**: Web böngésző lehetőségek Document/View architektúrában.
- **CEditView**: szöveges adatok megjelenítéséhez.
- **CRecordView**: ODBC form alapú adatelérésekhez.
- **CDAORecodView**: DAO form alapú adatelérésekhez.
- **COleDBRecordView**: OLE DB adatforrások megjelenítésére.
- **CTreeView**: fa struktúrájú megjelenítéshez.
- **CListView**: adatok listázásához.

A CView osztályok megjelenítését az *OnDraw()* metódus végzi. (WM_PAINT hatására meghívódik.) A CView::OnDraw metódus absztrakt (OOP 5.3.5. fejezet). ***virtual void OnDraw(CDC* pDC) = 0*;** Tehát a CView absztrakt osztály, nem

példányosítható. Ha view objektumot akarunk létrehozni, az utódosztályban felül kell írnunk az OnDraw metódust. Ezt az alkalmazásvarázsló előkészíti nekünk. A mi feladatunk a metódus értelmes kóddal történő feltöltése.

Az újrafestéshez az ***Invalidate()*** metódusokkal (5.3.1. szakasz) állíthatjuk be a frissítésre váró tartományt, melyet pihenőidőben a rendszer újrarajzol. Az újrafestést az ***UpdateWindow()*** hívással kényszeríthetjük ki, mely egy **WM_PAINT** üzenetet küld a szál üzenetsorába, ha van újrafestésre váró terület.

Az OnDraw csak az aktuális nézetet rajzolja újra. Az összes nézet frissítéséhez a ***CDocument::UpdateAllViews(NULL)*** metódusát kell meghívnunk, mely az adott dokumentumhoz tartozó nézeteket rajzolja újra. Mindegyikre meghívja az ***OnUpdate()*** metódust.

A View osztály metódusaiban a ***GetDocument()*** tagfüggvény hívásával kaphatunk a dokumentumosztályra (CDocument* típusú) mutatót, a ***GetParentFrame()*** pedig a keretablakosztály (CFrameWnd* típusú) mutatóját adja vissza.

A CView osztály biztosítja az alkalmazások számára a **drag & drop** technikát, amint azt a 4.8.6. szakasz részletezte.

Megjegyzés:

A CView osztály több osztálynak felajánlja, hogy lássák adatait. Ilyen a Document / View architektúra osztályai közül a CDocument, a CDocTemplate, a CPreviewView, a CFrameWnd, a CMDIFrameWnd, a CMDIChildWnd és a CSplitterWnd. A friend kapcsolat (OOP 6. fejezet) deklarációja a CView osztályleírásában, az afxwin.h-ban található.

6.1.2.5. A dokumentumosztály

A Document / View architektúra alaposztálya. Az alkalmazásban **az adatok tárolásáért felelős osztály.** A fájlokból az adatokat az alkalmazás számára kényelmesen kezelhető formában olvassa be, és kínálja föl megjelenítésre, módosításra.

A CDocument osztály ősosztálya a CCmdTarget. Ez azt jelenti, hogy **a dokumentumosztályban parancsüzeneteket kezelhetünk**.

A CObject utódosztályaként biztosítja, hogy **az Input / Output műveleteket szerializációval végezhetjük**. (Lásd 6.2. fejezet!)

Annak, hogy a CObject utódosztálya, további következménye, hogy **futásidejű információk elérését biztosítja** (4.3. szakasz). Ezt már kihasználtuk a dokumentumsablon létrehozásakor az alkalmazásosztályban (6.1.2.1. szakasz **RUNTIME_CLASS** makró). A futásidejű információknak köszönhető, hogy az

MFC-ben (jó ideje) megírt dokumentumsablon az általunk most létrehozott osztályok objektumait készíti el az alkalmazásunkban.

Az adatok kezelésére rendelkezésünkre áll a:

- ***DeleteContents()*** metódus, mely **az összes adattag tartalmát törli**. Az általunk felvett adattagok törlése érdekében felül kell írnunk a metódust! Használata SDI alkalmazásoknál különösen fontos, mivel az új dokumentum megnyitásakor ugyanazt a dokumentumobjektumot használjuk az adatok tárolására. A framework hívja mielőtt újrahasznosítaná a dokumentumot. A DeleteContents() függvény biztosítja, hogy üres dokumentumba olvassuk be az adatokat. (Dinamikus tagok helyének felszabadítása.)
- A ***SetModifiedFlag()*** metódus alapértelmezett paramétere a TRUE, ekkor **beállítja a módosítást jelző flaget**, melynek köszönhetően a dokumentum zárása előtt visszakérdez, mentse-e az adatokat. FALSE paraméter esetén törli a módosítás flag beállítását.
- Az ***IsModified()*** metódus nem nulla értékkel tér vissza, ha a dokumentum az utolsó mentés óta módosítva lett.

A dokumentumosztály egy **CPtrList** típusú (OOP 8. fejezet) m_viewList tagváltozóban tárolja a dokumentumhoz tartozó nézeteket. Ez egy típus nélküli mutatókat tartalmazó lista. A nézetlista kezeléséhez kényelmes függvények állnak rendelkezésünkre.

- Az ***AddView()*** segítségével nézetablakot csatolhatunk dokumentumunkhoz. A függvény a nézetosztály dokumentumosztályra mutató mutatóját is beállítja a dokumentumra.
- A ***RemoveView()*** kiveszi a nézetablakot a dokumentum listájából. A nézetablak dokumentum mutatóját NULL-ra állítja.
- A ***GetFirstViewPosition()*** visszaad egy **POSITION** típusú értéket, mellyel a GetNextView paramétereként megkaphatjuk a nézetlista első elemét.
- A ***GetNextView()*** **visszaad a paraméterében átadott POSITION típusú változóval meghatározott CView* mutatót a nézetlistáról és a POSITION értékét a következő nézetre állítja.** Ha a paraméter a GetFirstViewPosition() által adott érték, akkor a CView* mutató az első nézetablakra mutat. A függvény által beállított POSITION újra paraméterül adható a további nézetablakok eléréséhez. Ha a visszaadott nézetablak az utolsó a listán, a POSITION értéke NULL lesz. NULL paraméterrel ne hívjuk a GetNextView() metódust!

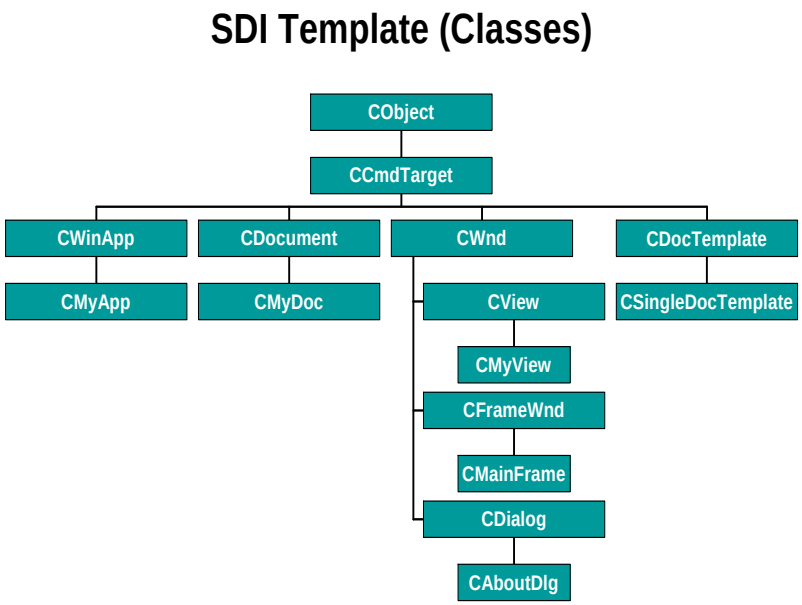
A következő kódrészlet végigmegy a nézetlistán és frissíti azt:

```
POSITION pos = GetFirstViewPosition();
CView* pView;
while (pos != NULL)
{
 pView = GetNextView(pos);
 pView->UpdateWindow();
}
```

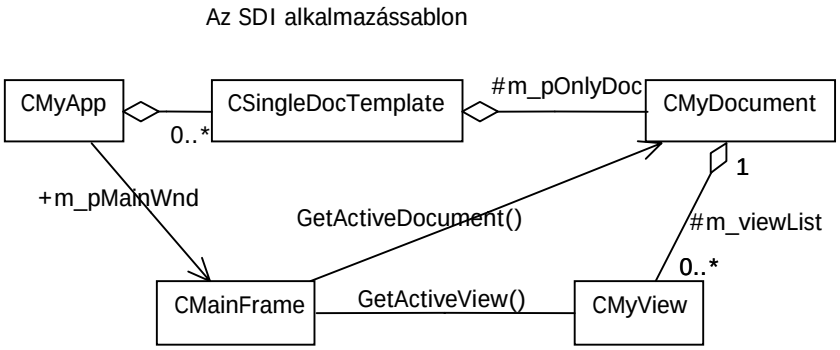
Rendelkezésünkre áll egy a CView osztálynál már említett függvény a fenti kód egyszerűbb megvalósítására:

- Az ***UpdateAllViews(NULL)*** frissíti a dokumentumosztály összes nézetét. Első paramétere a nézet (mutatója), amely módosította a dokumentumot, mert annak képét nem kell frissíteni. Ha NULL paraméterrel hívjuk, akkor minden nézetet frissít. További paraméterei a módosításról tartalmaznak információt, és alapértelmezettek.

6.1.2.6. Az SDI alkalmazássablon osztályai



6.6. ábra Az SDI alkalmazássablon osztályhierarchia diagramja



6.7. ábra Az SDI alkalmazássablon osztályai közti kapcsolatok

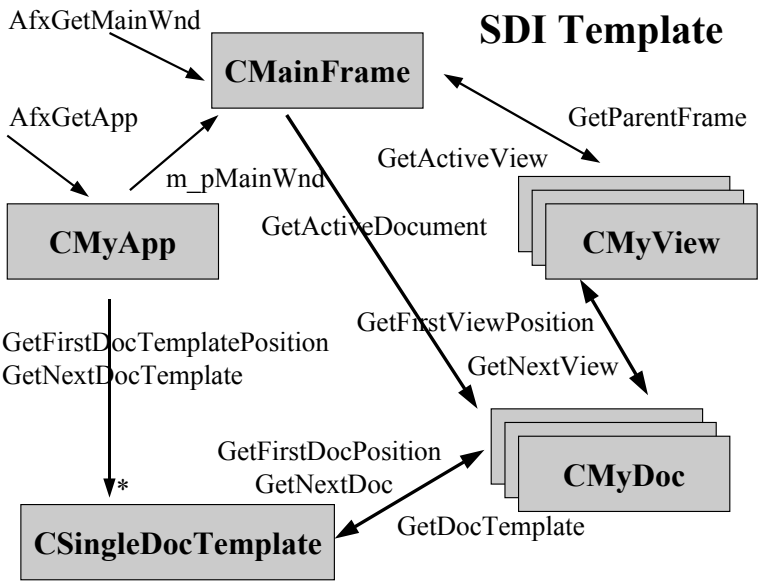
Az alkalmazásosztály tartalmazza a dokumentumsablonok listáját. Bár egyszerre csak egy dokumentum nyitható meg, az egymás után megnyitható dokumentumok lehetnek különböző osztályokhoz tartozók. Minden új dokumentumosztályhoz új dokumentumsablont kell rendelni (de ezek az objektumok mind a **CSingleDocTemplate** példányai). Ezért mutat az 6.8. ábra több dokumentumsablont.

A dokumentumsablonok mindegyike tartalmazza a saját dokumentumát. Dokumentumsablononként egy dokumentumosztály van, de minden egyes dokumentumsablonhoz saját dokumentumosztály tartozik. Tehát több dokumentumosztályunk lehet az alkalmazásban (6.8.ábra).

A dokumentumosztályok tartalmazzák a hozzájuk tartozó nézetablakok listáit. Egy dokumentumhoz több nézetablak tartozhat. A nézetablakok a keretablak osztály gyermekablakai.

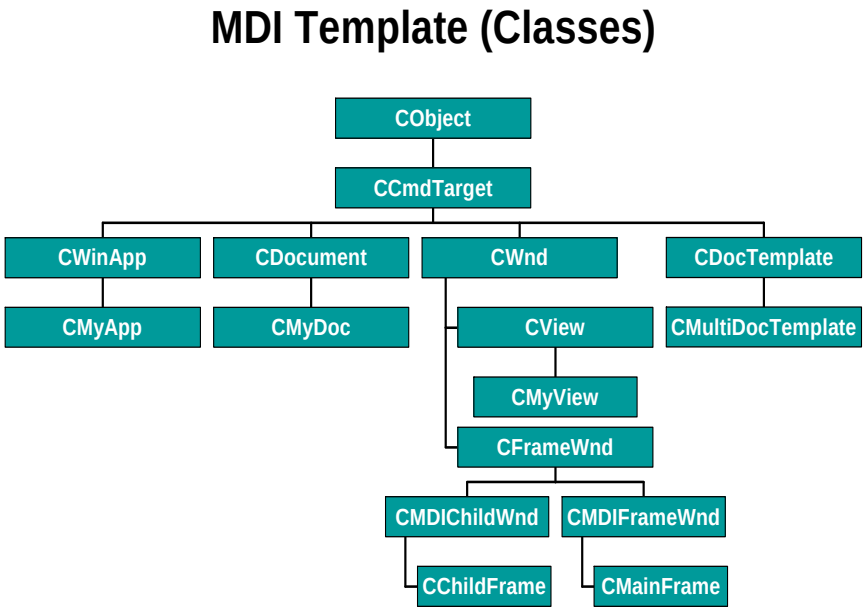
Az alkalmazásosztályból elérhetjük a főablakot. **A főablak a keretablak**.

Az alkalmazásvarázsló csak egy dokumentumsablon-osztályt készít el, a hozzá tartozó dokumentum- és nézetosztályokkal.

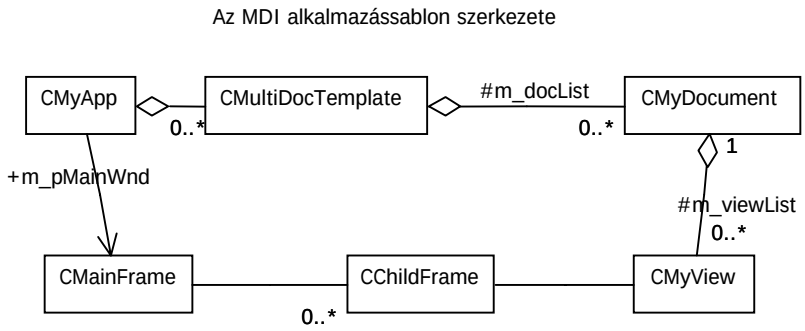


6.8. ábra Az SDI alkalmazássablon osztályai közti kapcsolatok függvényei

6.1.2.7. Az MDI alkalmazássablon osztályai



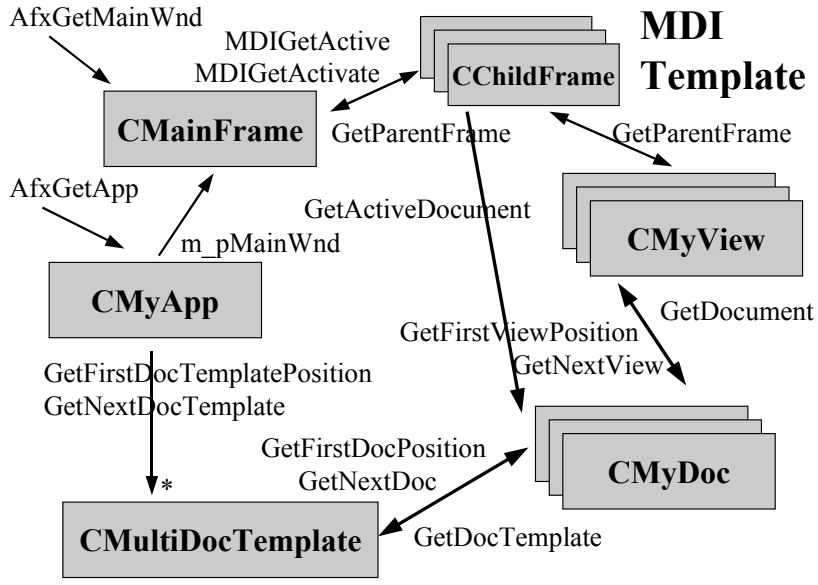
6.9. ábra Az MDI alkalmazássablon osztályhierarchia diagramja



6.10. ábra Az MDI alkalmazássablon osztályai közti kapcsolatok

Az alkalmazásosztály tartalmazza a dokumentumsablonok listáját. A dokumentumsablon objektumok mindegyike a CMultiDocTemplate példánya. A dokumentumsablonok tartalmazzák a már megnyitott dokumentumok listáját. (Lehet több azonos típusú dokumentum is, de lehetnek különbözőek is a különböző sablonokban.) Minden dokumentum tartalmazza a hozzá tartozó megnyitott nézetek listáját.

Az alkalmazás főablaka a CMainFrame objektuma, mely dokumentumonként tartalmaz egy gyermek keretablakot (CChildFrame), s ez a dokumentum nézeteit mint gyermekablakokat.



6.11. ábra Az MDI alkalmazássablon osztályai közti kapcsolatok függvényei

6.2. Szerializáció

Szükségünk van arra, hogy **az alkalmazás futtatása közti időszakokban is információkat őrizzünk meg az alkalmazás számára**. Erre két formát támogat az MFC, a **szerializáció**t (**serialization**) és az **alkalmazás állapotának megőrzését** (**application-state persistence**). A felhasználó által beállított állapotának megőrzésére szolgál a rendszerleíró-adatbázisban (registry) történő adattárolás. Erről a 7. fejezetben találunk információkat.

A szerializációval az alkalmazás adatainak megőrzését tehetjük kényelmessé. A szerializáció során adatfolyamokkal dolgozunk. Az adatokat a tartós tárolás érdekében fájlba írjuk.

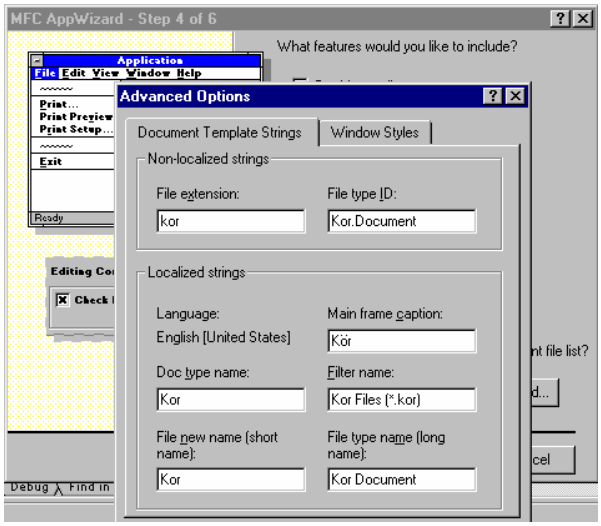
6.2.1. Az alkalmazáshoz rendelt fájlkiterjesztés

A fájlok kiterjesztéséből megtudhatjuk, hogy pl. futtatható-e az adott fájl (.exe, .com, .bat). Az adatokat tároló fájl nevéből (kiterjesztéséből) következtethetünk, milyen adatokat tartalmaz a fájl és arra is, hogy milyen alkalmazás segítségével nézhetjük meg ezeket az adatokat. A .txt kiterjesztésűek szöveget tárolnak, s szövegszerkesztővel olvashatók el. A .doc kiterjesztésűek a Word lehetőségeit felhasználó formázott szöveget, képet, egyéb adatokat tartalmaznak, a Word segítségével olvashatjuk el a tartalmat. A .bmp bitmapeket tárol és a Paint jeleníti meg őket. Az alkalmazások forrásfájljai között a .res kiterjesztésűek erőforrás-leírások, erőforrás-szerkesztővel, de szöveges fájlként is megnézhetjük tartalmukat. (Megnyitáskor ne az Auto módot válasszuk, hanem a szövegest!) Az általunk készített alkalmazások adatait tároló fájlnak mi határozhatjuk meg a kiterjesztését.

Az alkalmazásvarázslóval SDI / MDI alkalmazást készítve a 4. lépésben a párbeszédfelületen található az Advanced nyomógomb. Választására kinyílik egy párbeszédablak, melynek Document Template Strings fülét választva a File extension szövegmezőbe begépelhetjük az alapértelmezett kiterjesztést (6.12. ábra).

Ezt a kiterjesztést fogja az alkalmazás mentéskor automatikusa a fájl neve mögé írni, ha nem adunk meg más kiterjesztést.

Ebben az ablakban írhatjuk be a főablak címsorát is (Main frame caption).



6.12. ábra Az alkalmazás fájlkiterjesztése az alkalmazásvarázslóban

A fájlkiterjesztés hatására az alkalmazásvarázsló generál egy registry entry fájlt (.reg kiterjesztésű), melynek feladata a telepítő program hívása során a regisztrációs adatbázisba az alkalmazáshoz tartozó fájlkiterjesztés bejegyzése. Az alkalmazásosztály InitInstance tagfüggvényébe pedig bejegyzi a duplakattintásra történő megnyitáshoz (***EnableShellOpen()***; ***RegisterShellFileTypes(TRUE)***;) és a **drag & drop** technikával történő megnyitáshoz szükséges tagfüggvényeket (m_pMainWnd->***DragAcceptFiles()***;).

A registry kezeléséről további információkat találunk a 7. fejezetben.

Amennyiben az alkalmazásvarázslóban nem állítottuk be az alapértelmezett fájl kiterjesztést, vagy módosítani kívánjuk azt, megtehetjük azt a **Resource View String Table** erőforrást választva, az **IDR_MAINFRAME** azonosító Caption oszlopában (6.13. ábra). Az egyes szakaszokat \n választja el.

1. szakasz: SDI alkalmazásnál a főablak címsora, MDI alkalmazásnál nem használt, a főablak címsorát az IDR_MAINFRAME azonosítónál adjuk meg.

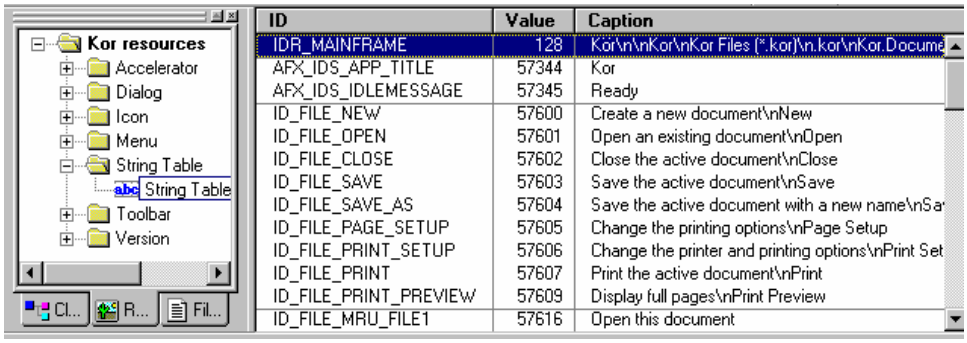
2. szakasz: MDI alkalmazásnál az adott dokumentumtípus keretablakának címsora, minden új dokumentum nyitása esetén egy növekvő sorszámmal megtoldva.

3. szakasz: MDI alkalmazásnál új dokumentum nyitásakor megkérdezi egy párbeszédablakban, hogy melyik dokumentumtípust szeretnénk megnyitni. E típusnak az elnevezése.

4. szakasz: Azt a szöveget tartalmazza, amit a fájlnyitáskor és mentéskor a dokumentum típusánál a legördülő menüben látunk.

5. szakasz: A fájlkiterjesztést adja meg.

6. szakasz: és 7. szakasz: A regisztrációs adatbázisban tárolt azonosítók (HKEY_CLASSES_ROOT), melyek meghatározzák, hogy a fájlkezelő vagy a nyomtató melyik alkalmazást nyissa meg az adott kiterjesztéshez.



6.13. ábra Az alkalmazás fájlkiterjesztése az erőforrás-szerkesztőben

A megnyitott dokumentum címe lekérdezhető a dokumentumosztály ***GetTitle()*** metódusával, míg útvonalát megkaphatjuk a ***GetPathName()*** metódussal. A cím a ***SetTitle()*** metódussal módosítható. (Pl. nem akarjuk angolul kiírni az Untitled szöveget.)

Ha még nem nyitottunk meg egyetlen dokumentumot sem, a GetTitle() az "Untitled" sztringet, míg a GetPathName() üres sztringet ad vissza.

6.2.2. Szerializáció

A szerializáció az alkalmazás adatainak mentése egy adatfájlba, vagy az adatok beolvasása egy adatfájlból.

Mivel az alkalmazás adatainak tárolásáért és kezeléséért (dokument / view architektúra esetén) a dokumentumosztály felelős, így **a szerializáció elindítása** is **a dokumentumosztály feladata**.

Az alkalmazásvarázsló létrehozza a CDocumentből származó osztályban a ***Serialize()*** függvényt, melynek paramétere a **CArchive& ar** hivatkozás.

A CAchive osztály egy kényelmes felületet biztosít a programozó számára ahhoz, hogy adatait fájlba mentse. Az objektumorientált programozás encapsulation elvét felhasználva lehetővé teszi, hogy a fejlesztő, kicserélje a CArchive által eltakart fizikai fájlt. Így tetszőleges szerkezetű fizikai fájlba egységes módon írhatjuk be adatainkat. Az alaptípusokra rendelkezésünkre állnak a << és >> operátorok (6.2.2.5. szakasz), mi is felülírhatjuk az operátorokat saját adattípusainkhoz. Az 'ar' objektum egy közbülső objektum a dokumentum és az adatfájl között. Konstrukciója

során egy CFile objektumot csatol hozzá a környezet, melynek a közvetlen fizikai fájllal történő kapcsolattartás a feladata. **Az 'ar' objektumot adatok beolvasására vagy mentésére használhatjuk, de mindkettőre nem**.

A dokumentumosztály vagy közvetlenül valósítja meg a szerializációt, vagy az 'ar' objektum átadásával az adattagokra bízza azt.

A szerializáció lényege, hogy az objektumok maguk felelősek adataik írásáért, olvasásáért. Minden objektumot teljes egészében szerializálunk, részleges szerializáció nem megengedett.

6.2.2.1. Szerializáció a dokumentumosztályban

Ha document / view archtekturájú, nem adatbázis alkalmazást készítünk, az alkalmazásvarázsló elkészíti a ***CDocument::Serialize()*** tagfüggvény felüldefiniálását a saját dokumentumosztályunkban. Nekünk csak a saját dokumentumosztályunk adattagjainak mentésével / beolvasásával kell feltöltenünk.

```
void CKorDoc::Serialize(CArchive& ar)
{
 if (ar.IsStoring())
 {
       // TODO: add storing code here
       ar << m_x << m_y;
 }
 else
 {
       // TODO: add loading code here
       ar >> m_x >> m_y;
 }
}
```

A ***CArchive::IsStoring()*** metódusa TRUE értékkel tér vissza, ha save vagy save as módon, tehát írásra nyitottuk meg a dokumentumot, FALSE, ha open, azaz olvasásra. A párja a ***CArchive::IsLoading()***, mely olvasásra nyitás esetén igaz.

! Ügyeljünk rá, hogy az adattagokat azonos sorrendben írjuk ki és olvassuk be a fájlból, mert egyébként hibás adatokat kapunk!

A szerializációt általában három paranccsal hívjuk meg, az új dokumentum, a megnyitás és a mentés paranccsal.

6.2.2.2. Új dokumentum parancs (New document)

SDI alkalmazás esetén az **új dokumentum** hívása a következő lépéseket jelenti:

- Az aktuális dokumentum ellenőrzése, történt-e módosítás (***IsModified()***). Ha igen, akkor a felhasználó dönti el, mentjük-e a módosításokat (szerializáció).
- Az adatok törlése a dokumentumobjektumból (***DeleteContents()***).
- Az új, üres dokumentum létrehozása.

Nincs konstruktor vagy destruktor hívás a dokumentumobjektumra.

MDI alkalmazás esetén új dokumentumobjektum jön létre, a saját nézeteivel együtt.

Mindkét esetben rendelkezésünkre áll az ***OnNewDocument()*** virtuális tagfüggvény az új dokumentum létrehozásához tartozó egyedi kódrészletek írásához. A fentiek alapján különösen SDI alkalmazás esetén fontos, hogy e kódot ne a dokumentumosztály konstruktorába írjuk. (Dinamikus adattag esetén az OnNewDocument a helye a new hívásnak és a DeleteContents() szabadítja föl a lefoglalt memóriát.) MDI alkalmazás az OnNewDocument metódust a konstruktor után hívja.

6.2.2.3. Létező dokumentum nyitása (Open)

SDI alkalmazás esetén

- Az aktuális dokumentum ellenőrzése, történt-e módosítás (***IsModified()***). Ha igen, akkor a felhasználó dönti el, mentjük-e a módosításokat.
- Megjelenik az Open párbeszédablak (**CFileDialog**), és bekéri a felhasználó által választott fájlt.
- Az adatok törlése a dokumentumobjektumból (***DeleteContents()***).
- Az új dokumentum adatainak beolvasása szerializációval (**deserializing**).

Nincs konstruktor- vagy destruktorhívás a dokumentumobjektumra.

MDI alkalmazás esetén új dokumentumobjektum jön létre, a saját nézeteivel együtt.

Mindkét alkalmazástípus esetén rendelkezésünkre áll az ***OnOpenDocument()*** virtuális tagfüggvény a már kiválasztott fájl esetén az egyedi kódrészletek megírásához. A fentiek alapján különösen SDI alkalmazás esetén fontos, hogy e kódot ne a dokumentumosztály konstruktorába írjuk.

Megjegyzés:

Amikor a program felhasználója megnyitja az első fájlt, a fájl neve felkerül a registry **MRU** listájára és a fájl menü Recent File felirata lecserélődik a fájl nevével. Az MRU listára annyi fájl kerül, amennyit az alkalmazásvarázslóban beállítottunk. Alapértelmezésben 4. Mivel a fájlok neve a registryben tárolódik, a következő indításkor a program kiolvassa őket a registryből. (Lásd 7.3. szakasz!)

6.2.2.4. A dokumentum mentése vagy bezárása

A fájl mentése azt jelenti, hogy fájlban tároljuk a dokumentum adatait. Ezt is a Serialize metódushívással tesszük.

Mielőtt lezárjuk a dokumentumot, a framework meghívja a ***CDocument::IsModified()*** metódusát, mely megvizsgálja, hogy a dokumentum adatai módosultak-e az utolsó mentés óta. Ha igen, a framework kitesz egy párbeszédablakot, mely megkérdezi a felhasználót, menteni akarja-e a módosított adatokat. Ez garantálja, hogy egyetlen módosított dokumentumot sem zárunk be mentés nélkül.

Láthatjuk, hogy a dokumentumosztály felelőssége az adatok tárolása és mentése. A fejlesztő feladata, hogy az általa létrehozott dokumentumosztály is meg tudjon felelni ennek az elvárásnak. Ezt úgy érheti el, ha minden módosítás után beállítja a módosítást jelző flaget a ***SetModifiedFlag()*** metódussal.

MDI alkalmazásnál a dokumentum bezárásakor a DeleteContents metódus végrehajtódik a konstruktor előtt.

Mivel az SDI alkalmazás egyetlen dokumentumot használ, gyakran nem találunk Close menüpontot az alkalmazás File menüjében.

Mindkét alkalmazástípus esetén rendelkezésünkre áll az ***OnSaveDocument()*** virtuális tagfüggvény a már kiválasztott fájl esetén az egyedi kódrészletek beírásához. Alapértelmezett megvalósítása megnyitja a fájlt, meghívja a Serialize metódust és a módosítást jelző flaget nem módosított állapotba helyezi.

6.2.2.5. Szerializáció különböző típusú adatok esetén

A standard Visual C++ iostream osztályához hasonlóan a **CArchive** osztály biztosítja számunkra az átdefiniált insertion << és extraction >> operátorokat a portábilis típusokra. Ilyenek a:

CTime, CTimeSpan	SIZE, CSize	float
WORD	CString	POINT, CPoint
DWORD	BYTE	RECT, CRect
double	LONG	COLORREF
	BOOL	

Nemportábilis típusok esetén konvertálni kell őket ekvivalens portábilis típusokra.

(Microsoft, 1997. 248. old.)

A szerializációhoz rendelkezésünkre áll a ***CArchive::Write()*** és a ***CArchive::Read()*** metódusa. Mindkettő egy típus nélküli mutatót (void*) és egy UINT számot vár paraméterül. A szám megadja a mutatóval jelölt struktúra méretét, bájtjainak számát. A függvények a második paraméterben megadott számú bájtot írnak ki / olvasnak be a CArchive objektumból. A Read visszatérési értéke egy UINT, mely megadja az aktuálisan beolvasott bájtok számát. Ha értéke kisebb, mint a paraméterben megadott érték, az azt jelenti, elértük a fájl végét. Ez az eset kivételkezelővel nem kezelhető le.

```
extern CArchive ar;
char pChar[100];
LOGFONT logfont;
//...
ar.Write(pChar, 100);
ar.Write(&logfont, sizeof(logfont))
```

6.2.2.6. Szerializálható osztályok készítése

A szerializáció lehetősége a **CObject** osztályban teremtődik meg, így minden utódosztálya élhet vele. Ilyen utód maga a dokumentumosztály is. A ***CObject::Serialize()*** metódus felülírásával valósítja meg a fejlesztő saját adatainak szerializációját.

Ahhoz, hogy létrehozott osztályunk szerializálható legyen, a következő lépéseket kell megvalósítanunk:

- Származzon **public** módon a **CObject** osztályból!
- Az osztály tartalmazzon **default konstruktor**t! (OOP 4.2. szakasz.)
- Az osztálydeklarációban szerepeljen a **DECLARE_SERIAL** makró, az osztály nevével mint argumentummal!

- Az implementációban szerepeljen az **IMPLEMENT_SERIAL** makró, melynek három paramétere: az osztály neve, az ősosztály neve és a sémaszám! (A sémaszámot lásd később!)
- Írjuk fölül a CObject-ből örökölt **Serialize()** virtuális metódust!

```
// My.h header file
//
/////////////////////////////////////////////////////////////

class CMy : public CObject
{
protected:
  int m_x;
  int m_y;
public:
  CMy(){ } //Default konstruktor!
  DECLARE_SERIAL(CMy);
  virtual void Serialize (CArchive& ar);
};

// My.cpp : implementation of the CMy class
//
/////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "My.h"

IMPLEMENT_SERIAL(CMy,CObject,1)

void CMy::Serialize (CArchive& ar)
{
  if (ar.IsStoring())
        ar << m_x << m_y;
  else
        ar >> m_x >> m_y;
}
```

A **sémaszám** (**schema number**) jelentősége akkor válik nyilvánvalóvá, **ha módosítjuk az adatszerkezetet**. Az egységbezárás (**encapsulation**) elve ezt határozottan támogatja (OOP 1.3.szakasz). Ilyenkor **kötelező a sémaszámot is növelni**! Például egy újabb adattag kerül be az osztályba. Az új verziójú mentés több adatot ment, s a beolvasás több adatot olvas be. Ha a régebbi verzióban mentett fájl adatait akarjuk beolvasni az új verzióval készült alkalmazásba, hiányozni fog egy érték. (Ha több adatot mentettünk, akkor pedig egy másik adat értékét olvasná be, mint az új objektum következő tagváltozóját.) Az ilyen hibák elkerülése érdekében a szerializáció ellenőrzi az aktuális sémaszámot és a mentett fájl sémaszámát, s ha azok nem egyeznek, hibaüzenetet küld.

6.14. ábra Hibaüzenet a sémaszám eltérése miatt

Ha osztályainkat szerializáció alkalmazására képessé tesszük, annak még egy előnye is lesz. Ha osztályunk elemeit az MFC által biztosított template collection (OOP 8. fejezet) osztályokban tároljuk, akkor elegendő a tároló osztályra meghívni a szerializációt, az automatikusan végrehajtódik az elemekre is.

```
CTypedPtrArray<CObArray,CCircle*> m_CircleArray;

void CKorDoc::Serialize(CArchive& ar)
{
 m_CircleArray.Serialize(ar);
}
```

A collection osztályok ***SerializeElements()*** metódusai minden egyes elemre meghívják a szerializációt.

A Serialize metódus mentéskor:

- Információkat ment a fájlba az adott objektum osztályáról.
- Meghívja az objetumra a Serialize() metódust, mely kiírja a fájlba az adatokat.

A Serialize metódus beolvasáskor:

- Betölti a fájlból az osztályról elmentett információkat, és készít egy az osztálynak megfelelő objektumot. A fenti példában CCircle*. Ehhez szükséges az osztály default konstruktora. Az objektum a collection egyik eleme lesz.
- Meghívja az objektumra az osztály Serialize() metódusát, mely betölti a fájlból az adatokat.

6.3. A keretablak interfészei

A parancsüzenetek interfészét és a tájékoztatás feladatát a keretablakok látják el. SDI alkalmazás esetén egy keretablakunk van, osztálya a CMainFrame, míg MDI alkalmazás esetén a fő keretablak (CMainFrame) és a gyermek-keretablakok (CChildFrame) megosztják a feladatokat. **A keretablak feladata a címsor, a gördítősávok, a státuszsor, a menüsor és az eszköztár kezelése. Ezeken a felületeken parancsüzeneteket küldhet a felhasználó az alkalmazásnak.** A parancsüzenetek bármely CCmdTarget utódosztályban kezelhetőek, így a kezelőfüggvények gyakran nem a keretablakhoz tartozó osztályban találhatóak meg. Parancsüzenetek esetén mindig gondosan mérlegelnünk kell a kezelő hozzárendelésekor, melyik osztály tudja legkényelmesebben kezelni az adott üzenetet.

6.3.1. Menükezelés

A menük eredeti feladata a beavatkozás lehetőségének biztosítása volt a program futásába. Ma a menük több feladatot is ellátnak:

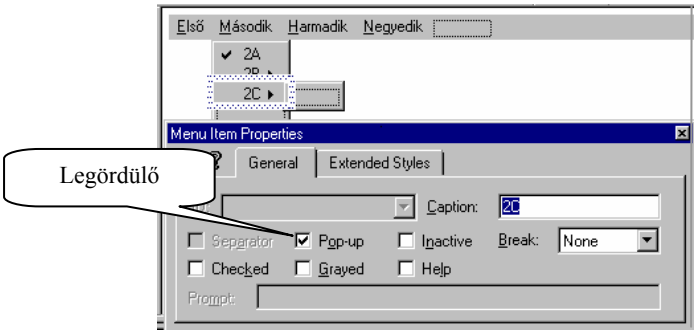
- Beavatkozás az alkalmazás futásába.
- Tájékoztatás az alkalmazás állapotáról.
- Tájékoztatás, mely opciók választhatók.

A menüpontokat a CMenu osztály interfészén keresztül dinamikusan kezelhetjük futásidőben. Ez azt jelenti, hogy a menüpont az alkalmazás aktuális állapotának megfelelően jelenik meg. Windows alkalmazásoknál természetes felhasználói elvárás, hogy az Edit menü Copy és Cut menüpontja szürke, amíg nincs kijelölt adat a másolásra / kivágásra. Hasonlóan a Paste is csak akkor választható, ha a vágólap nem üres. (Tájékoztatás az alkalmazás állapotáról.) A Fájl menüben pedig akkor jelennek meg az utoljára meghívott dokumentumok nevei (tájékoztatva bennünket a választás lehetőségeiről), amikor megnyitjuk az első dokumentumokat. Ha a Visual C++ fejlesztői környezet menüjére gondolunk, a Layout menüpont meg sem jelenik, míg egy párbeszéd-erőforrást meg nem nyitunk. S a Window menü Docking View menüpontja egy pipával jelzi az ablak dokkolt állapotát.

6.3.1.1. Menüpontok szerkesztése

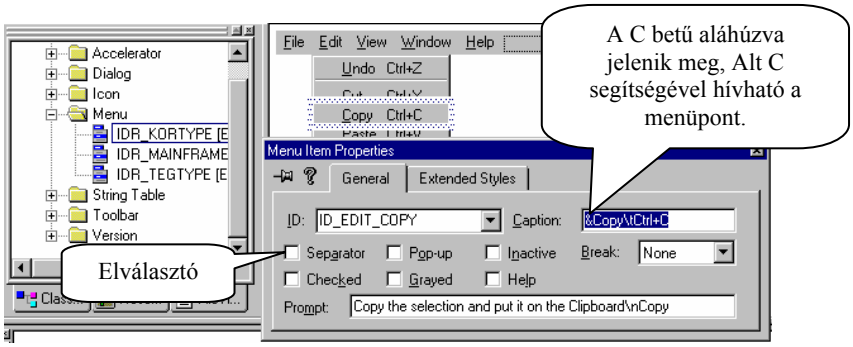
Az alkalmazásvarázsló Document / View architektúrájú alkalmazás esetén már kész menüvel (menükkel) vár bennünket. A Resource View / Menu-ben SDI alkalmazás esetén az IDR_MAINFRAME azonosító alatt találjuk a keretablak menüjét, MDI alkalmazások tartalmaznak egy menüt az üres fő keretablakhoz IDR_MAINFRAME (ha minden dokumentumot bezárunk) és egyet a generált dokumentumosztályhoz.

A menüpontok drag & drop technikával mozgathatók. Törölni egyszerűen a delete gombbal lehet. Minden menüpont lehet legördülő (drop-down) menü (pop-up jelölőnégyzet) vagy egy parancsot kezelő menüpont. A pop-up menüpontnak nincs saját azonosítója, nem rendelhető hozzá kezelő. Az oldalán egy nyíl jelzi, rákattintva újabb almenü nyílik ki. A pop-up tulajdonságú menüpont kiválasztására bekövetkező esemény az almenü (submenu) "legördülése".



6.15. ábra Menüpontok szerkesztése

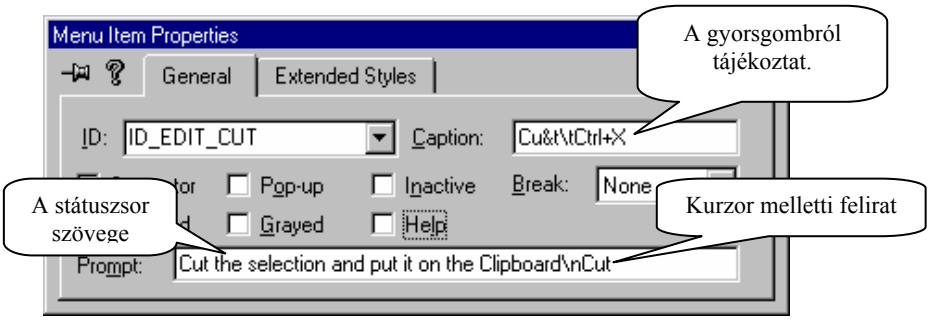
A menüpontok tulajdonságlapján a Caption mezőbe írhatjuk a menü szövegét. Amelyik betű elé & jelet teszünk, az aláhúzva jelenik meg, s az Alt gomb és az adott karakter leütésével hívhatjuk meg a billentyűzetről a menü kezelőjét.



6.16. ábra Menüpontok tulajdonságai

A 'Separator' jelölőnégyzet választással tehetünk a menüpontok közé elválasztó vonalat. A menüpont azonosítója az ID_FŐMENÜ_ALMENÜ elven készül. Ha a menüponthoz gyorsgombot csatoltunk, annak billentyűkombinációját a Caption mezőben a tabulátor jel (\t) mögé írhatjuk. Ez csak tájékoztató szöveg, a gyorsgombot még el kell készíteni.

A Prompt mezőbe írjuk a tájékoztató szöveget a menüpontról, melyet a státuszsorban olvashatunk, ha az egér épp a menüpontra mutat. A Prompt \n utáni szövege lesz az egérkurzor mellett a kis téglalapban megjelenő szöveg, ha a menüponthoz eszközgombot rendelünk, s az egeret az eszközgomb fölé mozgatjuk (ToolTip). Ide tehát csak rövid szöveget írjunk!



6.17. ábra Tájékoztató szövegek a menüpontról

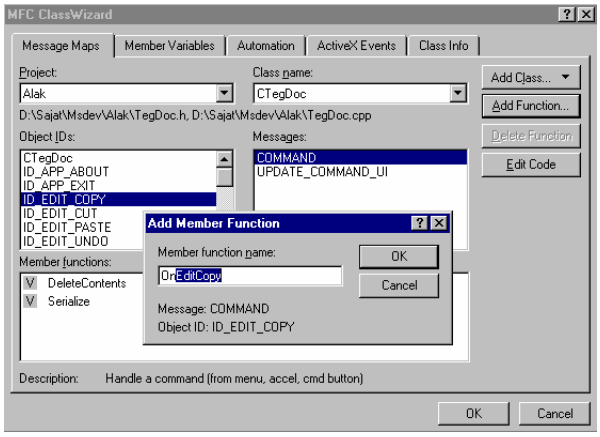
A prompt szövege nem más mint az adott paranccsal megegyező azonosítójú sztring erőforrás. A 6.17. ábra esetén az String Table ID_EDIT_CUT sorának Caption mezője.

6.3.1.2. Menüparancsok kezelése

A menükből parancsüzenetek érkeznek. A parancsüzeneteket több osztály is kezelheti. (Lásd 2.6.3. szakasz!) Mindig meg kell fontolnunk, melyik osztályra bízzuk a kezelést, hol a legkényelmesebb a kód megírása! Ha egy osztály lekezelte az eseményt, további kezelőmetódus nem hajtódik végre. Tehát a kezelést nem tudjuk az üzenetkezelő mechanizmussal több osztályra bízni.

 A 6.6.6. feladat teszteli a parancsüzenetek kezelését.

A kezelőt az osztályvarázslóval készíthetjük el. Válasszuk ki az adott menüpont azonosítóját az Object IDs ablakban, majd a Class name ablakban azt az osztályt, amelyikben a parancsüzenetet kezelni kívánjuk! A Messages ablakban a **COMMAND** legyen aktuális, és a kezelőt az Add Function gombbal rendelhetjük a menüponthoz! A kezelő neve a On-nal kezdődik, majd a főmenü és az almenü nevével folytatódik (6.18. ábra).



6.18. ábra Kezelőmetódus hozzárendelése a menüponthoz

Megjegyzés:

Ha egy menüponthoz nincs még kezelő rendelve, az a menüpont futás alatt szürke, tehát nem választható. Ha a dokumentumhoz több nézetablak tartozik, és azok nem mindegyike kezeli az adott menüpontot, a menüpont felirata csak akkor lesz választható, ha az őt kezelni tudó nézetablak az aktív.

A szokásos főmenü menüpontokhoz alapértelmezett kezelőt biztosít az MFC. Ezek a következők:

- File menü: New, Open, Close, Save, Save As, Page Setup, Print Setup, Print, Print Preview, Exit és az utoljára használt (MRU) fájlok listája.
- Edit menü: Clear, Clear All, Copy, Cut, Find, Paste, Repeat, Replace, Select All, Undo, Redo.
- View menü: Toolbar és Status Bar.
- Window menü: New, Arrange, Cascade, Tile Horizontal, Tile Vertical, Split
- Help menü: Index, Using Help, About. (Bennett, 1997, 67. old.)

A menüpontok feliratát természetesen módosíthatjuk, csak a megfelelő azonosítót kell hozzájuk rendelni. További részletes leírást találunk az MSDN Technical Notes 22. pontjában. (TN022)

 6.5. gyakorlatban több alapértelmezett menüpontot láthatunk működés közben.

6.3.1.3. A menüpont megjelenése

Az osztályvarázsló 6.18. ábrán látható ablakában a menüpont azonosítójához az **UPDATE_COMMAND_UI** üzenethez kezelőt rendelve a menüpont megjelenését kezelhetjük. A kezelő felkínált neve az OnUpdate + főmenü név + almenü. Paramétere a **CCmdUI* pCmdUI** arra az UI (user interface, felhasználói interfész) elemre mutat, amelyik meghívta a kezelőt. Az üzenethez rendelt kezelőmetódus, mint az a nevéből is kiderül, az előtt hajtódik végre, mielőtt a menüt kirajzolja az alkalmazás. A menüpont választása is frissíti a megjelenítést.

```
CMyClass::OnUpdateMenuItem(CCmdUI* pCmdUI)
{
pCmdUI->Enable(variable);
//A változó értékétől függően engedélyez.
}
```

A CCmdUI osztály tagfüggvényei:

- ***Enable(BOOL bOn = TRUE)***: Engedélyezi (TRUE) vagy tiltja a menüpont választását.
- ***SetCheck(int nCheck = 1)***: Kipipálja (1) nem (0) a menüpontot. A (2) meghatározatlan állapot csak a menüponthoz tartozó eszközgombon látható.
- ***SetRadio(BOOL bOn = TRUE)***: Kiteszi elé a pontot (TRUE) vagy nem.
- ***SetText.(LPCSTR lpszText)***: A paraméterben megadott szövegűre állítja.
- ***ContinueRouting()***: Az üzenetkezelés folytatását kényszeríti ki.

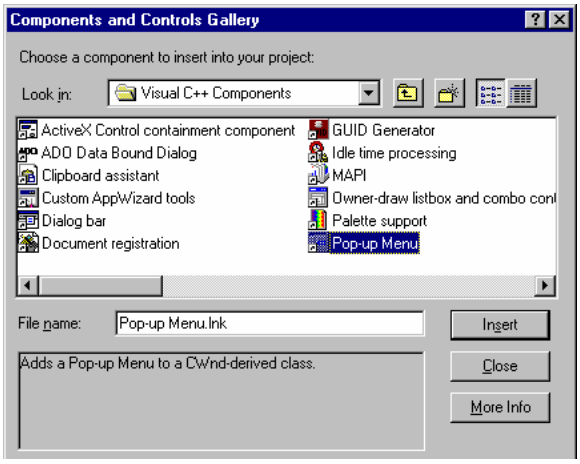
Megjegyzés:

A parancsüzenet kezelőit nem csak a felhasználó hívhatja meg, mi is elérhetjük őket bármelyik függvényünkből a SendMessage metódus segítségével. Pl. SendMessage(WM_COMMAND, ID_...).

6.3.1.4. Gyorsmenü

Az alkalmazásokban a jobb oldali egérgomb lenyomásával gyakran lehet egy felbukkanó menüt létrehozni. Ezt a menüt szokás gyorsmenünek (shortcut menu), felbukkanó, előbukó, lebegő vagy helyi menünek is nevezni.

Gyorsmenüt a Komponensgaléria (Component Gallery) (Project menü / Add to Project / Components and Controls) Pop-up Menu komponense segítségével hozhatunk létre.



6.19. ábra A komponensgaléria

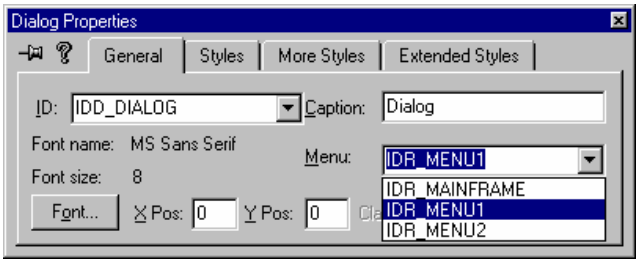
Az Insert hatására az erőforrás elkészül és a **WM_CONTEXTMENU** üzenet kezelőmetódusa a beszúrás során választott osztályba kerül. Ezután a főmenühöz hasonlóan csak a menüpontokhoz rendelt kezelőmetódusok megírására lesz szükség.

Gyorsmenüt a 6.6.5. feladatban készítünk.

6.3.1.5. Menü a párbeszédablakban

A párbeszédfelületre nem jellemző, hogy menüt kezelne, ez általában a keretablak feladata, de lehetőségünk van menüt létrehozni a párbeszédfelületen is. Ez a menü lehet gyorsmenü – az előző szakaszban leírt módszerrel elkészítve – vagy hagyományos menü a következő módon:

Az új menüerőforrás létrehozása után az erőforrás-szerkesztőben a **Dialog** erőforrások tulajdonságablakában megjelenő menü kombipanel listájából kiválasztható a párbeszédablakhoz tartozó menüerőforrás (6.20. ábra). Futáskor a menü a szokott módon a címsor alatt jelenik meg.



6.20. ábra Menü a párbeszédablakban

A menü kezeléséhez válasszuk a Dialog erőforráshoz tartozó párbeszédosztályt, és a menüpontokat is ebben kezeljük!

6.3.1.6. Dummy menü

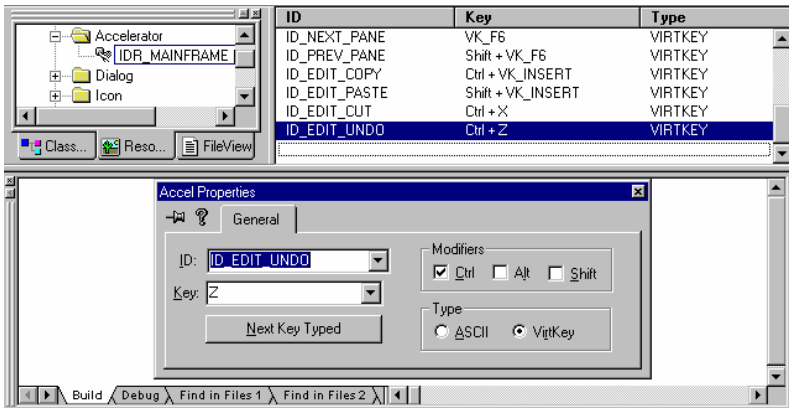
A dummy menu (statiszta menü) **olyan menüt jelent, amit menüként nem használunk**, tehát a felhasználói interfészen soha nem jelenik meg. **Célja, hogy azonosítókat hozzunk létre és azok kezelését kényelmesen, az osztályvarázslón keresztül biztosítsuk**. Ilyen eset pl. a státuszsor ablakainak kezelése, melyek azonosítóit nem lehet elérni az osztályvarázslóból (6.3.4.3. szakasz). Dummy menühöz nem rendelünk osztályt! Amikor az osztályvarázsló az új erőforráshoz osztályt rendelne, válasszuk a Cancel-t!


A dummy menü használatára a 6. gyakorlat 6.3.4. szakaszában látunk példát.

6.3.2. Gyorsgombok

A parancsüzeneteket, különösen a gyakran használtakat, el szokás látni **egy billentyűkombináció**val, **melynek hatására ugyanaz történik, mintha a menüpontot választottuk volna.** A menü mindig látható menüpontjainál (top-level menu) erre a célra megfelel az & jellel kijelölt betű által kapott Alt és a betű kiválasztásával meghatározott billentyűkombináció. Ezek a menüpontok viszont általában pop-up menük, így nem tartozik hozzájuk konkrét parancsüzenet. Az almenük esetében az &-el létrehozott a billentyűkombinációk csak az almenü legördülése után élnek, ami körülményessé teszi használatukat. Tehát szükségünk van a gyorsgombokra (accelerator key, hot key)!

A gyorsgombok beállításait a Resource View, Accelerator címszó alatt találjuk. Ügyeljünk rá, hogy több azonosat ne definiáljunk!



6.21. ábra A gyorsgombok beállítása

6.3.3. Az eszköztár

Az alkalmazásvarázsló által létrehozott eszközgombok a hozzájuk csatolt menüpontok szintjén működnek. Az eszköztár dokkolható.

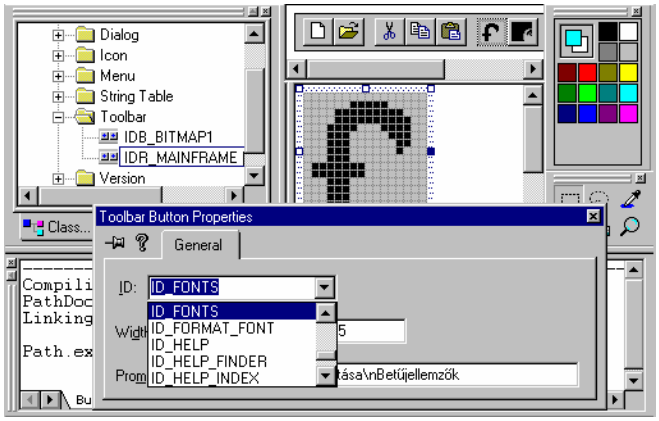
Figyeljük meg, hogy az eszközgombok ikonjai a Windows alkalmazásokban egységesek, pl. a kinyitott mappa a Fájl menü Open menüpontját hívja! Ez garantálja a felhasználó megszokott környezetét. Alkalmazásaink készítése során mi is törekedjünk e szabály betartására! A Visual C++ fejlesztői környezet biztosítja számunkra a szokásos ikonokat. A telepítés során a grafikus elemek között telepíthetőek számunkra például az Office vagy a Windows ikonjai. Alapértelmezésben a Common \ Graphics \ Icons alkönyvtárban találhatóak meg.

6.22. ábra A rendelkezésünkre álló ikonok

6.3.3.1. Az eszköztár szerkesztése

A Resource View Toolbar folder alatt található az **IDR_MAINFRAME** eszköztár. Ez a főablakhoz tartozó eszköztár. Az üres helyre új eszközgombot vehetünk fel. Az eszközikonra duplán kattintva az ID-t a legördülő listából választhatjuk ki. A prompt sor automatikusan kitöltődik az azonosítóhoz tartozó menüpont prompt sorának tartalmával. Futáskor az eszközgombot választva az ID által meghatározott menüponthoz rendelt kezelőmetódus hívódik meg.

Ha olyan eszközgombot szeretnénk, ami nem menüponthoz tartozik, egyedi azonosítót (ID) adjunk a gombnak! Az azonosító bekerül az osztályvarázsló Object IDs listájára, s ugyanúgy rendelhetünk kezelőt hozzá, ahogy a menüpontokhoz.



6.23. ábra Az eszköztár szerkesztése

Az eszköztár elemeinek sorrendjét drag & drop technikával módosíthatjuk. Az eszközgombot félig a mellette levő ikonra vagy közre mozgatva az eszközgomb csoportokat elválasztó hely létrejön vagy megszűnik. Törléshez egyszerűen le kell húzni az ikont az eszköztárról. Ha van két eszköztár: az ikonok egyikről a másikra másolhatók. (Bitmapből és ikonból is másolhatunk képet.)

Megjegyzés:

Készíthetünk olyan eszköztárat is, ahol az eszközgombok egy bitmap-ből jönnek létre. Vegyük föl a bitmap erőforrást! Legyen az az aktuális! Ekkor a főmenüben megjelenik az Image menüpont. Válasszuk ki a Toolbar Editor almenüpontot! Egy párbeszédablakban beállíthatjuk az eszközgombok kívánt méretét. A szokásos méret 16x15. Ha magasabb a bitmap, küld egy figyelmeztető üzenetet, hogy le kell vágnia a kép alját. A bitmapünk ezzel átkerül a Toolbar folder alá és elemeiből eszközgombok lesznek.

6.3.3.2. Több eszköztár az alkalmazásban

Alkalmazásunk több eszköztárat is használhat. Ehhez a következő lépéseket kell megtennünk:

- Új eszköztár (toolbar) erőforrás létrehozása és szerkesztése.

- Az osztályban vagy metódusban, amely az eszköztárat használja, vegyünk fel egy CToolBar osztályhoz tartozó objektumot! Amint ezt a CMainFrame-ben láthatjuk a m_wndToolBar adattag esetén. (Konstruktorhívás az objektum létrehozásakor.)

- Mivel a CToolBar osztály az ablakosztály utóda, a létrehozás két lépésből áll. A ***Create()*** metódussal készítsük el kezelőt! A Create első paramétere az eszközsor szülőablakára mutat, többnyire ebben az osztályban hozzuk létre, tehát this. További paraméterei alapértelmezettek. Ha Create(this) hívással hozzuk létre, az eszközsor a szülőablak tetején jelenik meg. A második paraméter a stílus, melyet az ablakosztály stílusai és az eszközsor stílusai segítségével állíthatunk be. Alapértelmezésben: *dwStyle* = **WS_CHILD** | **WS_VISIBLE** | **CBRS_TOP**. Ha az ablak aljára szeretnénk tenni az eszközsort, a CBRS_TOP helyett a **CBRS_BOTTOM** stílust kell megadnunk. További stílusok a súgóban érhetők el.

- Az eszköztár-erőforrás betöltéséhez az objektumra hívjuk meg a ***LoadToolBar()*** metódust, paraméterként átadva neki az eszköztár azonosítóját!

- A ***Create()*** által beállított stílust később is módosíthatjuk, ahogy ezt a CMainFrame osztály ***OnCreate()*** metódusában a fő keretablak eszközsoránál láthatjuk:

```
// TODO: Remove this if you don't want tool tips or a
resizeable toolbar
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
            CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
```

- A **CToolBar** osztály a **CControlBar** osztály közvetlen utóda, így dokkolhatóvá tehető az ősosztálytól örökölt ***EnableDocking()*** metódussal. A függvény paramétereként meghatározhatjuk, hogy a szülőablak melyik széléhez tapadhat hozzá az eszközsor. A **CBRS_ALIGN_ANY** paraméter a dokkolást bármely oldalhoz megengedi. Ha 0 a paraméter, az tiltja a dokkolást. Ahhoz, hogy a keretablak fogadni tudja a dokkolást, a keretablakra is meg kell hívni az EnableDocking metódust célszerűen ugyanazzal a paraméterrel. (Lásd 4.8.4. szakasz!) Majd a keretablak ***DockControlBar()*** metódusával beállíthatjuk az eszköztárra a dokkolást.

Ha az alkalmazásunkban több dokumentumsablont használunk, akkor azok mindegyikéhez más és más erőforrás-azonosítót rendelhetünk. A különböző erőforrás-azonosítók mögött pedig különböző menük és eszköztárak lehetnek. Ezek cseréjét az MFC biztosítja számunkra.

6.3.4. Az állapotsor

A státuszsor az alkalmazás állapotáról tájékoztat bennünket. Az osztályvarázsló által létrehozott állapotsor a Caps Lock, a Num Lock és a Scroll Lock gomb állapotát mutatja. Ha a felhasználó egy menü elemet vagy egy eszközgombot választ, a státuszsor megjeleníti a hozzá tartozó prompt tartalmát, ami a választott elem feladatát írja le a menüpont szövegénél bővebben.

6.3.4.1. Állapotsor a kódban

A státuszsort a **CStatusBar** objektumaként vesszük fel. Hozzá tartozik egy tömb, ami az ablakok (panes) azonosítóit tartalmazza (**indicators[ ]**).

```
static UINT indicators[] =
{
 ID_SEPARATOR,           // status line indicator
 ID_INDICATOR_CAPS,
 ID_INDICATOR_NUM,
 ID_INDICATOR_SCRL,
};
```

Az állapotsor ablakai a keretablak alján vízszintesen helyezkednek el, a tömbelemek sorrendjében balról jobbra. A tömb ablakai a státuszsorban jobbra rendezettek. A 0. helyen található a felhasználói interfész prompt sorait megjelenítő, az összes fennmaradó helyet elfoglaló ablak. Ezt a tömbben az **ID_SEPARATOR** azonosító képviseli.

6.24. ábra Az állapotsor

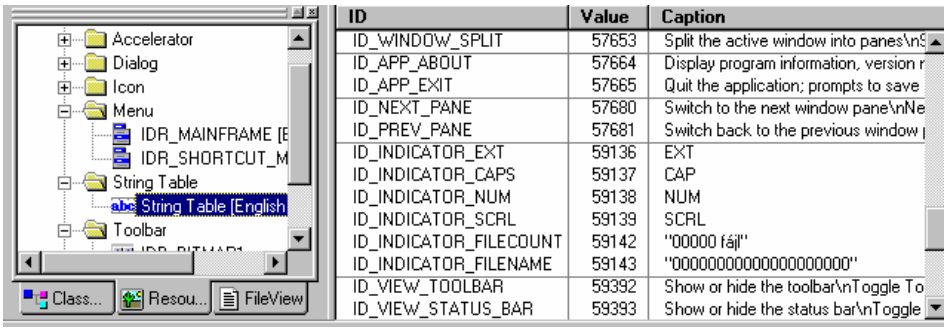
Az állapotsor objektumot a keretablak tartalmazza (m_wndStatusBar). Létrehozása két lépésben történik (konstruktor, Create()). Megjelenítése az eszközsorhoz hasonlóan a keretablak ***OnCreate()*** metódusában történik. A ***CStatusbar::SetIndicators()*** metódusa kapcsolja össze az indicators tömb azonosítóit az objektummal.

```
if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
        sizeof(indicators)/sizeof(UINT)))
{
      TRACE0("Failed to create status bar\n");
      return -1;      // fail to create
}
```

Az alkalmazásban használt szövegeket a Resource View, String Table tárolja. Mi is tehetünk ide szövegeket, melyekre aztán azonosítójukkal hivatkozhatunk. Már

használtuk az IDR_MAINFRAME azonosítót az alkalmazás által támogatott fájlkiterjesztés beállításakor.

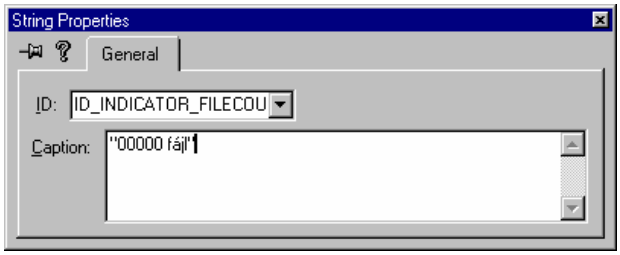
A sztringeket csoportokban tárolja, egy-egy csoport maximum 16 elemű. A státuszsor ablakainak azonosítóit az INDICATOR szakaszban találjuk. Az osztályvarázsló által generált elemek már a tömb elemei. Pl. az **ID_INDICATOR_CAPS** azonosító azt az ablakot azonosítja, amelyikben megjelenik a CAP szöveg, ha a Caps Lock gombot "bekapcsoljuk".



6.25. ábra A String Table az INDICATOR tömb elemeivel

6.3.4.2. Új ablak az állapotsorban

Új ablakot az állapotsorba új elem felvételével hozhatunk létre. (Jobb oldali egérgomb, New String.) Adjuk meg az azonosítót (ID_INDICATOR_...)! A Caption-be a nullák az ablak méretének meghatározása miatt kerülnek, az ablakban nem jelennek meg. Konstans és változó tartalmú szöveget egyaránt írhatunk bele.



6.26. ábra Új ablak a státuszsorban

A MainFrame.cpp-ben található indicators tömb elemeinek száma dönti el a státuszsorban látható ablakok számát és sorrendjét. Tehát föl kell vennünk az indicators tömbbe az új azonosítót!

6.3.4.3. Kiírás az állapotsorba

- Ha csak az állapotsor üzenetablakában akarunk szöveget megjeleníteni, felhasználhatjuk, hogy a CWnd osztály ősosztálya a CStatusBar osztálynak. A ***CWnd::SetWindowText()*** segítségével írhatunk az állapotsorba, de csak az első ablakba tudunk így írni (a 0. pozícióba).
  - Bármely ablak szövegét a ***CStatusBar::SetPaneText()*** metódusával módosíthatjuk a legegyszerűbben. Sikeres kiírás esetén nem nulla a visszatérési érték.

    BOOL SetPaneText(int *nIndex*, LPCTSTR *lpszNewText*, BOOL *bUpdate* = TRUE);

    Első paramétere az ablak indexe az indicators tömbben, második az új szöveg, harmadik megadja, hogy az invalidate flaget frissítendőre állítsa-e. Alapértelmezésben igen.
  - Az ID_INDICATOR_... azonosítók nem jelennek meg az osztályvarázslóban, tehát nem tudunk **az állapot**uk **módosításához kezelőmetódus**t (command UI handler) a szokott módon hozzárendelni. Egyik lehetőségünk, hogy a vezérlők csoportos kezelésénél látott módon (3. Gyakorlat 3.1.2. szakasz) kézzel írjuk be a szükséges függvényeket.

    ↳ Az osztályban, ahol kezelni akarjuk az ablakot, deklaráljunk egy tagfüggvényt:

    ```
    //}}AFX_MSG
            afx_msg void OnUpdateMyPane(CCmdUI* pCmdUI)
    DECLARE_MESSAGE_MAP().
    ```

    ↳ A .cpp fájlban:

    ```
    //}}AFX_MSG_MAP
       ON_UPDATE_COMMAND_UI(ID_INDICATOR_MYPANE, OnUpdateMyPane)
    END_MESSAGE_MAP()
    ```

    ↳ Írjuk meg az `afx_msg void OnUpdateMyPane(CCmdUI* pCmdUI)` függvény implementációját!
  - Másik lehetőség, hogy az azonosítót nem a String Table-ben vesszük fel, hanem **létrehozunk egy dummy menü**t, **és a menüpontok azonosítóit az indicators tömbbe írjuk**. (Természetesen a menü azonosítói is beíródnak a String Table-be.) Ekkor a menüpont tulajdonságlap prompt mezőjébe kerül a méretet és az állandó szöveget meghatározó sztring. A menühöz nem rendelünk osztályt. A menüponthoz ezután kényelmesen rendelhetünk **UPDATE_COMMAND_UI** kezelőt az osztályvarázslóból.

! A statiszta menü menüpontjaiban a prompt kitöltése kötelező! Ez határozza meg az ablak méretét és a kiírás kezdetéig a szövegét a 6.3.4.2. szakaszban leírtaknak megfelelően. Ha üres a prompt, futáskor a "**Failed to create empty document**" hibaüzenetet kapjuk!

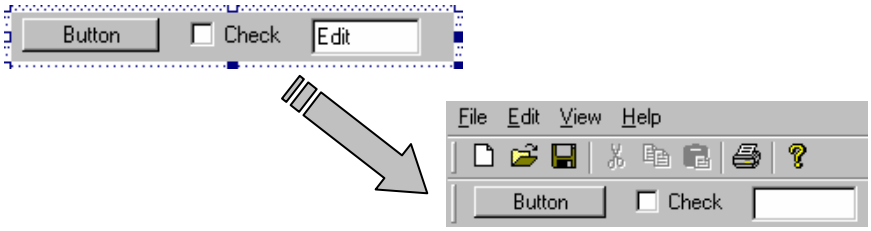
A ***CStatusBar::SetPaneInfo()*** metódusával lehetőségünk van az ablakok azonosítójának, stílusának, méretének futásidejű módosítására.

A framework a státuszsort **pihenőidő**s feldolgozással kezeli. (Idle time processing.) Ha pihenőidőben is változó értéket akarunk megjeleníteni, gondoskodnunk kell a pihenőidő megszakításáról. (Lásd 2.1. szakasz!)

 A 6.6.4. feladatban látunk példát az idő kiírására a státuszsorban.

6.3.5. További elemek

- A Resource View eleme a Version kategória is, mely az alkalmazásra vonatkozó információkat tárolja. Ezek is szöveges információk. Megadhatjuk benne a verzión kívül, pl. a copyright információkat vagy a CompanyName-ben a gyártó cég nevét.
- Az alkalmazásvarázsló Document / View architektúrájú alkalmazásaiban a 4. lépésben a "How do you want your toolbars to look?" kérdésre az Internet Explorer ReBars választógombot kijelölve alkalmazásunkba egy **CDialogBar m_wndReBar** vezérlősor kerül, melyet az erőforrás-szerkesztőben kényelmesen feltölthetünk a kívánt vezérlőkkel. Kezelésük azonosítóik segítségével bármely osztályból megtörténhet. Ha változót kívánunk hozzájuk rendelni, ahhoz egy olyan osztályt kell kiválasztanunk, amely párbeszédfelülettel rendelkezik. Tehát pl. a szövegmező EN_CHANGED üzenetét bármelyik osztály kezelni tudja, de ahhoz, hogy egy int típusú tagváltozót hozzárendeljünk a vezérlőhöz, ahhoz egy CDialog vagy CFormView... utódosztályt kell a projektünkbe betenni. (Az osztályvarázsló rákérdez.) Ebben az osztályban hozzá lehet rendelni a vezérlőhöz a változót az osztályvarázslón keresztül.



6.27. ábra DialogBar szerkesztése és a futás képe

A 6.3. gyakorlat kezeli a keretablak felhasználói interfészeit.

6.4. Több ablak az alkalmazásban

A párbeszédfelületen több vezérlő (ablak) lehet egy dialog ablakban. Document / View architektúra esetén is létrehozhatunk több ablakot egy nézetben.

Az SDI és MDI alkalmazásoknál egyaránt tartozhat egy dokumentumhoz több nézet is. A dokumentumosztály tárolja a nézetek listáját.

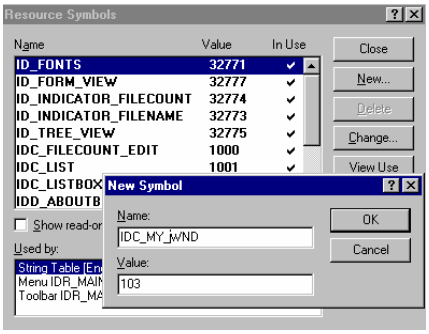
6.4.1. Egy nézet, több ablak

A nézetosztályban felvehetünk olyan tagokat, melyek a CWnd utódosztályainak példányai. Az ablakok létrehozása itt is két lépésben történik. A View konstruktora hívja a tagváltozók konstruktorát. A rendszerablak létrehozása a View osztály **WM_CREATE** üzenetkezelőjében, az ***OnCreate()*** metódusában történik. m_objektum.Create(). A CWnd::Create() metódusa minden ablakra hívható.

virtual BOOL Create(LPCTSTR *lpszClassName*, **LPCTSTR** *lpszWindowName*, **DWORD** *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID*, **CCreateContext*** *pContext* = **NULL);**

- **lpszClassName**: Az ablakosztály neve. NULL paraméter esetén default attribútumokkal dolgozik.
- **lpszWindowName**: Az ablak neve. (Lásd 6.6.4. feladat nézetablaka!)
- **dwStyle**: Az ablak stílusa, ablakstílusokból (WS_) és az utódosztályok stílusaiból (BS_, LBS_, ES_...) bitenkénti vagy művelettel jön létre.
- **rect**: Az ablak mérete. Később a szülőablak (View) WM_SIZE üzenetkezelőjében kényelmesebben beállíthatjuk a méretet. Emiatt gyakran lényegtelen a rect paraméter értéke.

- **pParentWnd**: A szülőablak mutatója. Többnyire this.
- **nID**: Az új ablak azonosítója. Ha tartozik az ablakhoz erőforrás (pl. FormView), akkor annak azonosítóját írjuk be, egyébként a View menü Resource Symbols ablakában kell új azonosítót felvenni. (6.28. ábra)
- **pContext**: A ***CMainFrame::OnCreateClient()*** metódusában paraméterként adott, tehát felhasználható. Mivel alapértelmezett NULL értéke van, megadása elhagyható. (Részletesebb leírása a fejezet végén.)



6.28. ábra Új azonosító felvétele

A szülőablak méretének módosítása magával vonja a gyermekablakok méretváltozását is. A gyermekablakot rögzíthetjük például adott távolságra a szülőablak széleitől. A szülőablak (most a View) **WM_SIZE** üzenetkezelőjében az ***OnSize()*** metódusban kényelmesen beállíthatjuk a méretet a ***MoveWindow()*** metódussal. Az OnSize két paramétere cx és cy a szülőablak (most a View) kliensterületének szélességét és magasságát adja. A teljes kliensterület a (0,0,cx,cy) hívás. Az OnSize és vele a gyermekablak méretének beállítása minden méretváltozásnál meghívódik.

A **CMainFrame** osztályban az állapotsor a **CStatusBar** m_wndStatusBar és az eszközsor **CToolBar** m_wndToolBar is gyermekablakok, és a fent leírt módon jönnek létre.

Ha a nézetablakba további gyermekablakokat hozunk létre, a view és a gyermekablakok egymás fölé kerülnek. Az írás az átlapolt (egymás fölé lógó) gyermekablakokban felveti a következő problémát.

> Ha először a szülőablakot rajzoljuk újra, majd a gyermekablakokat, akkor a szülőablakban fölösleges a gyermekablak alatti területet megrajzolni, nemcsak a hatékonyság miatt, hanem mert ez fölösleges villogást okoz. A szülőablakban a **WS_CLIPCHILDREN** stílus beállítása esetén a szülőablak nem rajzolja ki a gyermekablakai alatt fekvő területet.

```
BOOL CMyView::PreCreateWindow(CREATESTRUCT& cs)
{
 // TODO: Modify the Window class or styles here by modifying
 //  the CREATESTRUCT cs
 cs.style |= WS_CLIPCHILDREN;
 return CView::PreCreateWindow(cs);
}
```

6.4.2. Egy dokumentum, több nézet

A dokumentum / nézet architektúrában a dokumentum osztály feladata az adatok tárolása, a nézetosztályé a dokumentum adatainak megjelenítése. Egy dokumentum adatait többféleképpen is képernyőre vihetjük. Ez a különböző nézetek feladata.

Több nézet lehet azonos típusú azért, hogy a dokumentum más-más részeit mutassa, vagy különböző típusú, hogy a dokumentum adatait más-más módon mutassa.

A dokumentumosztály egy **CPtrList** m_viewList listában (OOP 8. fejezet) tárolja a hozzá tartozó nézetablakokat. A lista kezeléséhez kényelmes függvények állnak rendelkezésünkre.

- ***GetFirstViewPosition()***: Visszaadja a nézetlista kezdőpozícióját.

  ```
  POSITION pos = GetFirstViewPosition();
  ```

- ***GetNextView()***: Paramétere egy **POSITION** típusú változó. Visszaadja a pos által mutatott nézetablak mutatóját és növeli a pos paramétert, hogy a következő elemre mutasson.

  ```
  CView* pView = GetNextPosition(pos);
  ```

  Az utolsó elem után a pos értéke NULL lesz.

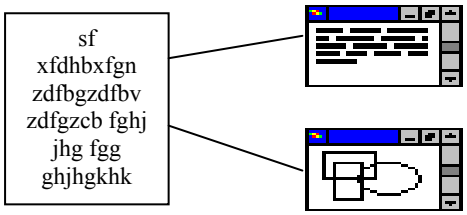
- ***AddView()***: A paraméterben megadott új nézetet veszi fel a nézetlistára.

- ***RemoveView()***: A paraméterben megadott nézetet veszi le a nézetlistáról.

Ha módosul a nézethez tartozó dokumentum tartalma, a módosítást a nézetablakon az ***OnUpdate()*** metódus végzi el. Az OnUpdate metódust meghívja a ***CView::OnInitialUpdate()*** és a ***CDocument::UpdateAllViews()*** metódus egyaránt. Alapértelmezésben a teljes kliensterületre beállítja az újrafestés szükségességét.

A nézeteket keretablak veszi körül, mely biztosítja a felhasználói interfészeket (keret, címsor, menü, eszközgombok...) a nézetekhez.

- **SDI** alkalmazásoknál a nézeteknek egy közös keretük van.
- **MDI** alkalmazásoknál a közös fő keretablakon kívül minden önálló nézet saját kerettel rendelkezik.



6.29. ábra Egy dokumentum több nézet

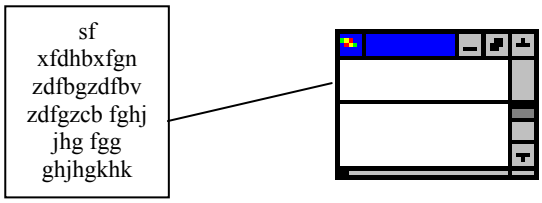
A nézetek lehetnek önálló nézetek (6.29.ábra) vagy **osztott nézetek** (**splitview**) (6.30, 6.31. ábra).

6.4.2.1. Osztott nézetek

Az osztott nézetek közös keretablakkal rendelkeznek.

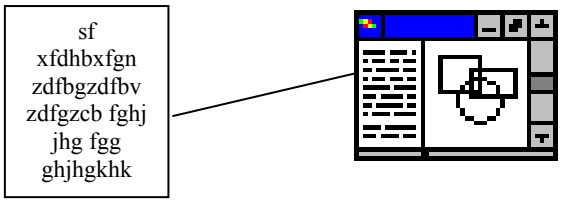
Az osztott nézeteknek két csoportját különböztetjük meg.

- A **Dinamikus osztott nézetek csak azonos CView utódosztályokhoz tartozhatnak! Kezdetben csak egy ablak jön létre, a továbbiak csak a splitter gomb elhúzása hatására dinamikusan keletkeznek**. Nem kell megadnunk az osztályukat, mert az az első nézetablakéval azonos. A dinamikus osztott nézetek biztosítják, hogy egy dokumentum több részét nézzük, módosítsuk egyszerre. (6.30. ábra.)



6.30. ábra Dinamikus, osztott nézet

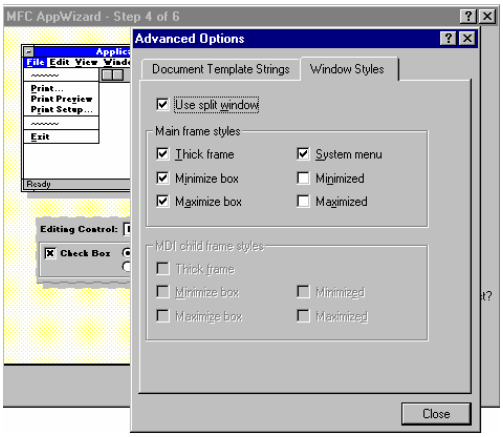
- **Statikus osztott nézetek lehetnek azonos vagy különböző osztályokhoz tartozók. Ami közös bennük, hogy az összes nézet létrejön a keretablakkal együtt**, legfeljebb nem látható. Tehát az ablakok nem dinamikusan jönnek létre. A statikus osztott nézetek mérete is változtatható. Ha a létrehozáskor megadott minimális méretnél kisebbre húzzuk össze, nem látható az ablak. (6.31. ábra.)



6.31. ábra Statikus, osztott nézet

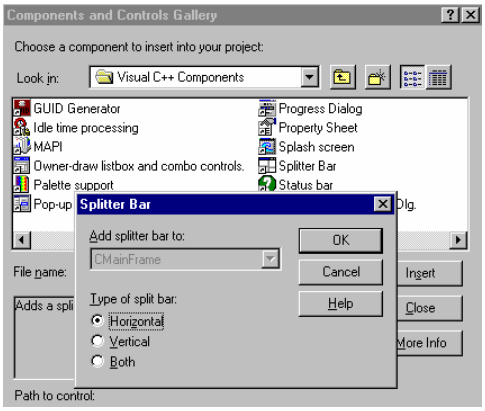
Osztott nézetet Dialog based alkalmazás esetén nem használhatunk!

Azt, hogy az alkalmazásunk használjon dinamikus splitviewt, már az **osztályvarázsló**ban eldönthetjük. A negyedik lépésben az Advanced Options gomb Window Styles fülében választható a Use split window jelölőnégyzet.



6.32. ábra Osztott nézet beállítása az alkalmazásvarázslóban

Ha később, már a fejlesztés során szeretnénk az alkalmazásunkban mégis lehetővé tenni használatát, rendelkezésünkre áll a **komponensgaléria** (Project menü / Add To Project / Components and Controls /Visual C++ components / Splitter Bar). A választás visszaigazolása után egy párbeszédablakban beállíthatjuk, hogy vízszintes, függőleges vagy mindkét irányban osztani akarjuk-e az ablakunkat (6.33. ábra).



6.33. ábra Dinamikus osztott nézet beállítása komponensgaléria segítségével

A beállítások után keletkező kód a következő:

A **CMainFrame** osztályban új protected adattag jön létre: **CSplitterWnd** m_wndSplitter néven. Az ***OnCreateClient()*** metódus felüldefiniálva:

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
 // CG: The following block was added by the Splitter Bar
component.
 {
        if (!m_wndSplitter.Create(this,
                                  2, 1,
              // TODO: adjust the number of rows, columns
                                  CSize(10, 10),
              // TODO: adjust the minimum pane size
                                  pContext))
        {
              TRACE0("Failed to create splitter bar ");
              return FALSE;    // failed to create
        }

        return TRUE;
 }
}
```

Nem hívja az ősosztály OnCreateClient metódusát, mert az csak egy nézetet generál.

A ***CFrameWnd::OnCreateClient()*** metódusát a framework az OnCreate végrehajtása során hívja meg. Ne hívjuk közvetlenül! Alapértelmezett megvalósítása a pContext paraméterében megadott adatok alapján létrehoz egy CView objektumot. A CREATESTRUCT paraméter módosítására ne ezt a függvényt, hanem a CWnd::PreCreateWindow metódust használjuk!

A ***CSplitterWnd::Create()*** metódusa dinamikus osztott nézetet hoz létre. Első paramétere a szülőablak, jelenleg a fő keretablak. A második paraméter a sorok, míg a harmadik az oszlopok száma. Egyik sem lehet kettőnél több (és egynél kevesebb)! A negyedik paraméter adja meg, hogy 10 egységnél kisebb ablak már nem látható. Ezt az értéket érdemes a megjelenített adatoktól függővé tenni. Szöveges adatnál fél sor már nem olvasható, ikonok esetén beállíthatjuk, hogy legalább egy sor ikon látsszon. Az utolsó paraméter az OnCreateClient **CCreateContext** paramétere, továbbadjuk az ablak létrehozásához.

Különböző nézeteket tartalmazó statikus osztott nézet kódját kézzel kell megírnunk. A statikus splittert a ***CSplitterWnd::CreateStatic()*** metódusával hozhatjuk létre.

BOOL CreateStatic(CWnd* *pParentWnd*, **int** *nRows*, **int** *nCols*, **DWORD** *dwStyle* **= WS_CHILD |WS_VISIBLE, UINT** *nID* **= AFX_IDW_PANE_FIRST);**

Mivel a függvény első három paraméterén kívül a többi alapértelmezett, három értékkel is hívható. Ez a három: a szülőablak, a sorok és oszlopok száma. (16-nál nem lehet több!)

```
m_wndSplitter.CreateStatic(this,2,1);
```

Most kötelezően létre kell hoznunk a nézetablakokat is a ***CSplitterWnd::CreateView()*** metódusa segítségével!

virtual BOOL CreateView(int *row*, **int** *col*, **CRuntimeClass*** *pViewClass*, **SIZE** *sizeInit*, **CCreateContext*** *pContext* **);**

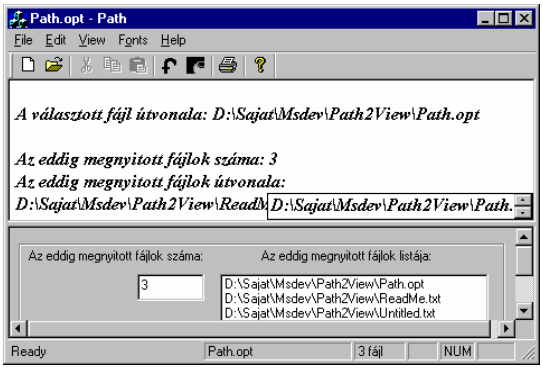
Paraméterei:

- **row**: Az itt létrehozott ablak a splitwindownak hányadik sorában található.
- **col**: Az itt létrehozott ablak a splitwindownak hányadik oszlopában található.
- **pViewClass**: A **RUNTIME_CLASS**(class_name) makróhívással állítható elő. (Ahhoz, hogy az osztályokra hivatkozni tudjunk, be kell szerkeszteni az osztály fejlécfájlját a keretablakosztályba. Egy további hibaüzenet miatt pedig a nézetosztályba kell beszerkeszteni a doc.h fejlécfájlt.)

A további paraméterek megegyeznek az előző függvényével.

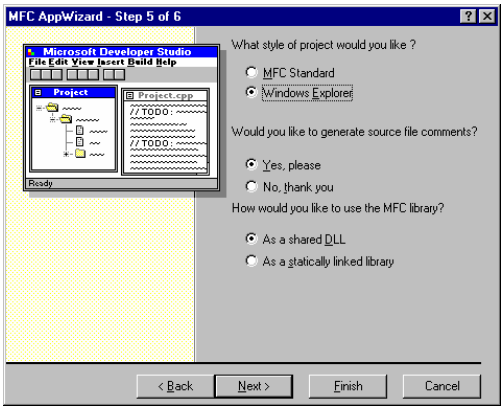
Természetesen a nézetosztályainkat létre kell hoznunk, méghozzá oly módon, hogy a RUNTIME_CLASS makró hívható legyen rájuk (4.3.szakasz). FormView esetén először az erőforrást készítjük el, majd az osztályvarázslóval hozzárendeljük a **CFormView** utódosztályt, kiválasztva a párbeszédablakban az erőforrás azonosítóját is az osztályhoz.

Ne feledjük el, hogy az alapértelmezett nézetosztályt az alkalmazásvarázsló utolsó lépésében, amikor összefoglalja az alkalmazás osztályait, kijelölhetjük!



6.34. ábra A 6.4. gyakorlat osztott ablakai

Az osztályvarázslóval készíthetünk böngésző stílusú alkalmazást is. Ehhez az SDI / MDI alkalmazás 5. lépésében (Step 5 of 6)-ban az MFC Standard jelölőnégyzet helyett a Windows Explorert válasszuk (6.35. ábra)! Ekkor az alkalmazás szerkezete kiegészül egy **CLeftView** nevű osztállyal, mely a függőleges Split bar-ral elválasztott bal oldali nézet osztálya. Ősosztálya a **CTreeView**. A jobb oldali nézetablak (**CMyView**) sem közvetlenül a CView utódosztálya, őse a **CListView** osztály.



6.35. ábra Explorer típusú alkalmazás

Az osztott ablakok létrehozását a 6.4. gyakorlat segítségével próbálhatja ki.

6.4.2.2. Önálló nézetek

Az önálló nézetablakokat is a keretablak OnCreateClient metódusában célszerű létrehozni, mert itt rendelkezésünkre áll a CCreateContext struktúra.

A **CCreateContext** (nincs ősosztálya) mutatókat tartalmaz a dokumentum (**m_pCurrentDoc**), a keretablak (**m_pCurrentFrame**), a nézet (**m_pLastView**, **m_pNewViewClass**) és a dokumentumsablon (**m_pNewDocTemplate**) objektumra. A CreateContext bármelyik tagja lehet NULL. Gyakran a teljes struktúra opcionális, ezért pl. a CWnd::Create utolsó paramétere alapértelmezésben NULL. Ha valamelyikre mégis szükség van, Assertion hibaüzenetet kapunk.

A CreateContextnek köszönhetően a létrehozott nézetablakot nem kell sajátkezűleg felfűzni a dokumentumosztály nézetablak listájára. Mivel a nézetek mutatója a dokumentum nézetlistán szerepel, az ablak területének felszabadítása a memóriából a dokumentumobjektum felszámolásakor megtörténik. Ha önálló változóval is elérhetjük a nézetablakot, ez pointer típusú legyen, hogy ne szabadítsuk fel kétszer a lefoglalt memóriaterületet (6.6.7. gyakorlat)!

A CView absztrakt osztály az MFC-ben is több utódosztállyal rendelkezik (4.8.5. szakasz). Saját nézetosztályunkat származtathatjuk közvetlenül a CView-ból, de bármelyik utódosztályából is.

A nézetablakok eléréséhez rendelkezésünkre állnak a dokumentumosztály ***GetFirstViewPosition()*** és ***GetNextView()*** tagfüggvényei (6.4.2. szakasz eleje). Általában az ***UpdateAllViews()*** segítségével módosítjuk az összes ablakban megjelenített adatokat.

Ha az aktuális nézetet cserélni akarjuk, nem elég aktívvá (SetActiveView) tenni a kívánt nézetet. A mi feladatunk, hogy az új nézet látható is legyen, míg a régi nem (ShowWindow(SW_SHOW / SW_HIDE)). Ezen kívül be kell állítanunk az első lap azonosítóját is (AFX_IDW_PANE_FIRST). Lásd 6.6.7. OnViewTreeView() függvénye.

6.5. Kérdések

1. Mi a különbség az SDI és az MDI alkalmazás között?
2. Sorolja fel a Dokument / View architektúra osztályait, ismertesse felelősségüket!
3. Milyen függvények segítségével juthatunk a különböző osztályok objektumaihoz SDI és MDI alkalmazássablon esetén?
4. Hogyan adhatjuk meg az alkalmazáshoz rendelt fájlkiterjesztést?
5. Hogyan működik a szerializáció? Melyik osztály támogatja a fájlba írást? Hogyan érhetjük el, hogy a dokumentum bezárásakor figyelmeztetést kapjunk a változtatások elmentésére?
6. Új osztály létrehozásakor mivel biztosíthatjuk, hogy szerializálható legyen? Mi a sémaszám?

7. Ismertesse a keretablak interfészeit!
8. Milyen menüket ismer? Milyen tulajdonságokat kell beállítanunk egy menüpont esetén? Mely osztályokból kezelhetjük a menüpontokat? Hogyan módosíthatjuk a menüpont megjelenését?
9. Hogyan kezeljük az eszköztár gombjait? Hogyan hozhatunk létre több eszköztárat az alkalmazásban?
10. Hogyan hozhatunk létre gyorsgombokat? Hol tárolja az alkalmazás a verziószámot? Mi a vezérlősor?
11. Mi az állapotsor feladata? Hogyan írhatunk az állapotsorba? Hogyan hozhatunk létre új ablakot az állapotsorba?
12. Hogyan hozhatunk létre a nézetablakunkba új ablakot?
13. Mi hozza létre a dokumentum első nézetét? Hogyan hozhatunk létre egy dokumentumhoz több nézetet? Hogyan érhetjük el a különböző nézeteket?
14. Mit jelent az osztott nézet? Milyen fajtái vannak? Ismertesse jellemzőiket! Hogyan hozhatjuk létre a különböző osztott nézeteket? Melyik módszerrel lehet a dokumentum adatait azonos módon egymás alatt három ablakban megnézni úgy, hogy mindhárom ablak a dokumentum adatainak más-más részét mutassa?

6.6. Összefoglalás

- Az **MFC** biztosítja számunkra a **Document /View architektúrá**t, melynek két alaposztálya a dokumentum- és a nézetosztály. A **dokumentumosztály** felel az adatok tárolásáért, beolvasásukért és mentésükért. A **nézetosztály** az adatok megjelenítését végzi. A **keretablakosztály** biztosítja a felhasználó számára a kommunikációs lehetőséget a programmal. Felelőssége a felhasználói interfészek kezelése. Az **alkalmazásosztály** indítja az alkalmazást, létrehozza az alkalmazás fő szálát, a **dokumentumsablon-osztály** segítségével pedig az alkalmazás szerkezetének alaposztályait. Az alkalmazásosztály tárolja a dokumentumsablonokat, a dokumentumsablonok MDI alkalmazás esetén a megnyitott dokumentumok listáját, SDI alkalmazásban a megnyitott dokumentumot, a dokumentumosztályok pedig a nézetablakok listáját.
- **Az SDI (Single Document Interface) alkalmazás egyszerre csak egy dokumentumot tud kinyitni.** Új dokumentum nyitásakor az előzőt bezárja. Az adatok tárolására (azonos dokumentumosztály esetén) ugyanazt a dokumentumobjektumot használja, mint az előző esetben. Egy dokumentumhoz is tartozhat több nézet, tehát egyszerre több ablakunk lehet nyitva.
- **Az MDI alkalmazás egyszerre több dokumentumot tud nyitva tartani**. Itt is lehet egy dokumentumról több nézetünk. Az alkalmazásnak van egy fő keretablaka, és minden önálló nézethez tarozik egy gyermek-keretablak.

- **A keretablakosztály feladata a felhasználói interfészek kezelése**. A menüpontok, a gyorsgombok, az eszköztár gombjai parancsüzeneteket küldenek, melyeket minden – a CCmdTarget osztályból származó osztály kezelni tud. A fejlesztő felelőssége eldönteni, melyik üzenetet melyik osztály kezelje le. Egy azonosítóhoz két üzenetet kezelhetünk. A **COMMAND** üzenethez tartozik a választás parancsának kezelése. Az **UPDATE_COMMAND_UI** üzenet kezelője az interfész megjelenítését szabályozza.
- **A menü feladata**, hogy lehetővé tegye a **beavatkozást az alkalmazás futásába**, hogy tájékoztasson bennünket az **alkalmazás állapotáról** és hogy tájékoztasson arról, **mely opciók választhatóak**. A menüpontoknál a tulajdonságlapon beállíthatjuk, hogy **separator, pop-up** vagy szokásos menüpontról van-e szó. Az előző kettőhöz nem tartozik azonosító, a legördülő menü további menüpontokat nyit. A menü Caption mezőjében az & jel jelöli azt a karaktert, mellyel a billentyűzetről is elérhetjük a menühöz tartozó parancsot. (Az Alt és a karakter segítségével.) **Gyorsmenüt a komponensgaléria Pop-up menü komponense segítségével készíthetünk**.
- **A menüpontokhoz vagy eszközgombok azonosítóihoz gyorsgombot rendelhetünk.** A gyorsgombbal közvetlenül érhetjük el a hozzá rendelt parancsot. A menüpont szövegében (Caption) a tabulátor jel után tájékoztatni szokás a felhasználót a gyorsgombról.
- **A gyakran használt menüpontokhoz eszközgombot rendelhetünk.** Az eszköztár gombjait az erőforrás-szerkesztőben készíthetjük el. A Visual Studio a szokásos Microsoft ikonokat rendelkezésünkre bocsátja. Az alkalmazásvarázsló egy eszköztárat hoz létre, de mi **több eszköztárat is használhatunk alkalmazásunkban.** A menüpont tulajdonságlapján a prompt sorba a \n mögé írt szöveg – az úgynevezett ToolTip – az egeret az eszközgomb fölé mozgatva jelenik meg.
- **Az állapotsor az alkalmazás állapotáról tájékoztat bennünket.** Lehetőségünk van az állapotsorba szöveget kiírni. Az indicators tömbbe újabb azonosítót felvéve tudunk a státuszsorba új ablakot felvenni. Az állapotsor szerkesztéséhez nincs erőforrás-szerkesztő, legegyszerűbben statiszta menüvel (dummy menu) kezelhetjük ezeket az ablakokat. Az menüpont tulajdonságlapján a prompt sorba írt szöveg az állapotsorban jelenik meg.
- **Egy nézetablak több gyermekablakot is tartalmazhat.** Ezeket, mint minden ablakot, két lépésben hozhatjuk létre, a konstruktor és a Create metódus segítségével.

- **Egy dokumentumosztályhoz több nézetablak is tartozhat.** Ezek lehetnek osztott nézetek vagy önálló nézetek. A **dinamikus osztott nézetek** a split bar elhúzásakor jönnek léte, **csak azonos osztályhoz tartozhatnak** és számuk legfeljebb 2x2 lehet. A **statikus osztott nézetek** a splitwindow-val együtt jönnek létre, **különböző nézetosztályhoz tartozhatnak**, számuk legfeljebb 16x16. A nézeteket a dokumentumosztályban tárolt listán keresztül érhetjük el a ***GetFirstViewPosition()***, ***GetNextView()*** metódusok segítségével. Frissíteni leggyakrabban a dokumentumosztály ***UpdateAllViews()*** metódusával szokás őket.

7. A registry használata

A registry egy egységes konfigurációs adatbázis. A registryben tárolhatjuk egyaránt a rendszerszintű és a felhasználó által létrehozott beállításokat. A registryben találhatók mindazok az információk, melyek az eszközmeghajtók, a szolgáltatások, az egyes programok és a biztonsági rendszer működéséhez szükségesek.

A registry a Microsoft korábbi operációs rendszereinek alábbi konfigurációs fájljait helyettesíti:

MS-DOS: **config.sys, autoexec.bat**

Windows 3.1: **system.ini, win.ini, reg.dat** és az egyes programok **.ini** kiterjesztésű konfigurációs állományait.

Gyakorlatilag minden, a rendszerbeállításokat megmutató és megváltoztató program pl. vezérlőpult, felhasználókezelő a registryből olvas és a registrybe ír. (Kis, 1999, 112. old.)

Amíg az alkalmazások kezdeti beállításait az alkalmazások .ini fájljai tárolták, több problémával kellett szembesülnünk.

- Mivel minden alkalmazás saját inicializáló fájllal rendelkezett, ezt a fájlt könnyen letörölhették, áthelyezhették vagy módosíthatták a felhasználók. **A registry az összes információt a regisztrációs adatbázisban tárolja.**
- Az .ini fájlok text fájlok, míg **a registry binárisan tárolja az adatokat**. A szöveges adatok szövegszerkesztővel módosíthatóak, míg a registry adatait a

RegEdit eszköz segítségével vagy a CWinApp osztály metódusai segítségével módosíthatjuk.

Megjegyzés:

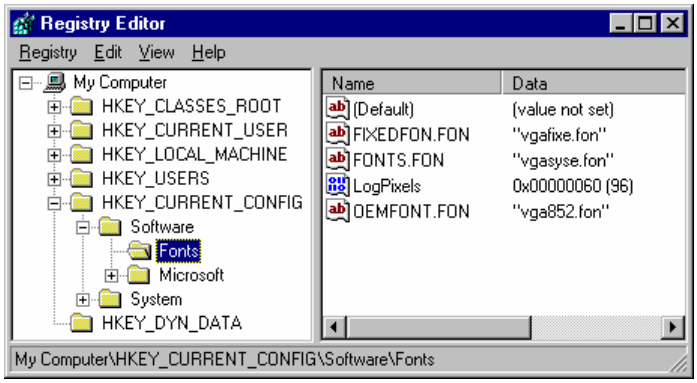
A registry nem egy adatbázis, tehát nem szabad konkurensen több alkalmazásból írni és olvasni! Kiolvasásra optimalizált.

7.1. A regisztrációs adatbázis szerkezete

A registry hierarchikus felépítésű. Szerkezete a könyvtárstruktúrához hasonló. Az adatokhoz kulcsok (key, subkey) megadásával juthatunk el. Hat elsődleges kulcsot használ:

- **HKEY_LOCAL_MACHINE**: A számítógép rendszerszintű beállításait tartalmazza. A telepített hardverek beállításait, az eszközmeghajtók, az operációsrendszer-szolgáltatások paramétereit, valamint a biztonsági adatbázist tárolja. A rendszer minden felhasználója számára hozzáférhető. A kulcs tartalmát többnyire a rendszerkönyvtár WINDOWS \ SYSTEM32 \ Config alkönyvtár állományai alkotják.
- **HKEY_CURRENT_USER**: Az aktuális felhasználó rendszerre és alkalmazásokra vonatkozó beállításai. A felhasználó bejelentkezésekor jönnek létre. A felhasználói profil user.dat állományában találhatók.
- **HKEY_USERS**: Minden a gépre bejelentkező felhasználóról tartalmaz információkat. Ezek az adatok lehetnek felhasználóspecifikusak vagy az összes felhasználóra vonatkozóak (.DEFAULT). Az éppen dolgozó felhasználó beállításai az S-...-...-.. kulcs alatt találhatóak. A kulcs neve a felhasználó biztonsági azonosítószáma.
- **HKEY_CLASSES_ROOT**: Az állománytípusok (kiterjesztések) mellett megadja azt a programot, amit alapértelmezés szerint a megadott kiterjesztésű állomány megtekintéséhez, szerkesztéséhez használunk (duplakattintás az adott kiterjesztésű fájlon). Tartalmaz drag & drop protokolokat, nyomtatási konfigurációkat is. Főként COM objektumok használják.
- **HKEY_CURRENT_CONFIG**: Aktuális hardverkonfiguráció.
- **HKEY_DYN_DATA**: A különböző meghajtók dinamikus állapotinformációi. A plug & play technológia használja.

A registry a **RegEdit** (**Registry Editor, Rendszerleíró-adatbázis-szerkesztő**) eszköz segítségével tekinthető meg, illetve módosítható (Start menü / Run / regedit) (7.1. ábra). Hasonló célú a **RegEdt32** (Windows NT) eszköz (7.2. ábra).

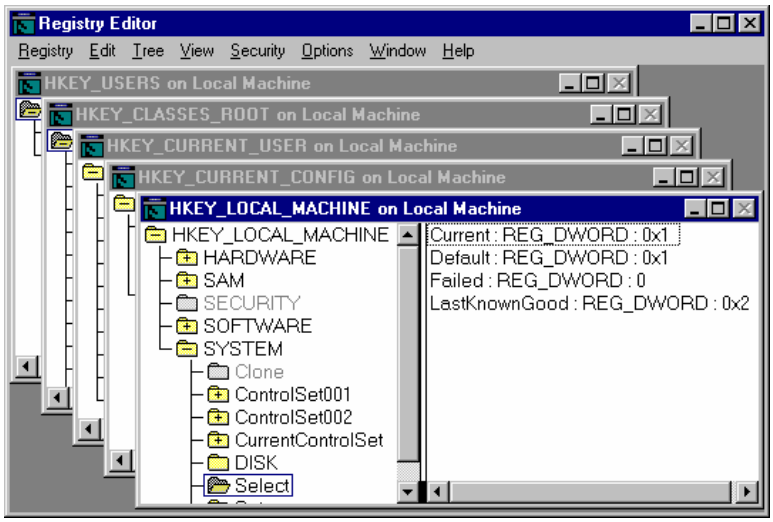


7.1. ábra A RegEdit

A regisztryben a kulcsokat és változókat elérési útjukkal adhatjuk meg, így pl. a számítógépünk nevét tartalmazó változót a következő úton érhetjük el:

HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ ComputerName \ ComputerName.

Az előredefiniált kulcsokat és az alkulcsokat a 7.1. ábra bal oldali ablaka, a változók neveit és értékeit a jobb oldali ablak mutatja. A RegEdt32-ben a változók típusát is láthatjuk az értékük előtt (7.2. ábra).



7.2. ábra A RegEdt32

A **HKEY_LOCAL_MACHINE legmagasabb szintű kulcsai** speciális kulcsok:

- **HARDWARE**: A számítógép hardverének aktuális konfigurációs beállításai, a gép indításakor dinamikusan jönnek létre, a lemezen levő adatbázisban nem szerepelnek.
- **SAM** és **SECURITY**: Biztonsági adatbázis és a biztonsági rendszer beállításai. Ne módosítsuk a registry editorral!
- **SOFTWARE**: A számítógépen telepített programok (beleértve az operációs rendszert is) környezeti beállításai. A HKEY_CURRENT_USER állítja a felhasználó egyedi beállításait.
- **SYSTEM**: Az operációs rendszer szolgáltatásainak beállításai.

7.2. A regisztrációs adatbázis módosítása

A beállítások módosítását több alkalmazásból végezhetjük. Pl. vezérlőpult, felhasználó-kezelő, alkalmazások beállításai, regisztrációs adatbázis-szerkesztő.

7.2.1. Módosítás szerkesztővel

Ha a módosítást közvetlenül a Registry Editorból végezzük, figyeljünk rá, hogy beállításaink azonnal tárolódnak a regisztrációs adatbázisban. Nem kell őket menteni, nincs is Save parancs erre. A közvetlen módosítások könnyen oda vezethetnek, hogy használhatatlanná válik a számítógép. A súgóban találunk információkat, hogyan lehet a registry előző állapotát visszaállítani. A RegEdt32 Registry menüjében találunk **save key funkciót**, melyet célszerű használni a kulcs értékének módosítása előtt. Segítségével visszaállítható a kulcs akkori állapota amikor elmentettük. A regisztrációs adatbázis-szerkesztőt csak végszükség esetén használjuk módosításra!

A kulcs név szerinti kereséséhez használjuk a View menü Find Key parancsát! Az Edit menü parancsaival tudunk új kulcsot vagy új változót felvenni, illetve törölni. Módosítani az Edit menü Modify parancsával lehet (RegEdit), illetve a változót kiválasztva, duplán kattintunk rajta (RegEdt32)

A változó típusok közül néhány itt következik:

- **REG_BINARY**: számérték kettes számrendszerben megadva (tetszőleges bitsorozat).
- **REG_DWORD**: 32-bites számok sorozata hexadecimálisan megadva.
- **REG_SZ**: tetszőleges karaktersorozat.
- **REG_MULTI_SZ**: karaktersorozat-tömb.

A RegEdt32 Options menüjében a szerkesztő Read Only Mode beállítással a véletlen módosítások elkerülése érdekében csak olvashatóra állítható.

7.2.2. Módosítás telepítéskor

A setup programok registry entry fájlokat használnak, hogy regisztrálják adatfájljaikat. **A registry entry fájl egy szöveges fájl .reg kiterjesztéssel.** Amikor a regiszrációs adatbázis-szerkesztő feldolgoz egy registry entry fájlt, a benne található információt hozzáadja a regisztrációs adatbázishoz, vagy módosítja az ott található értéket. Nem használható adatok törlésére. A telepítő program meghívja a registry editort és átadja neki a registry entry fájl nevét mint parancssor argumentumot. A szerkesztő a fájlt előbb ellenőrzi, hogy az **első sora REGEDIT**-e. Ezután olvassa csak be az adatokat.

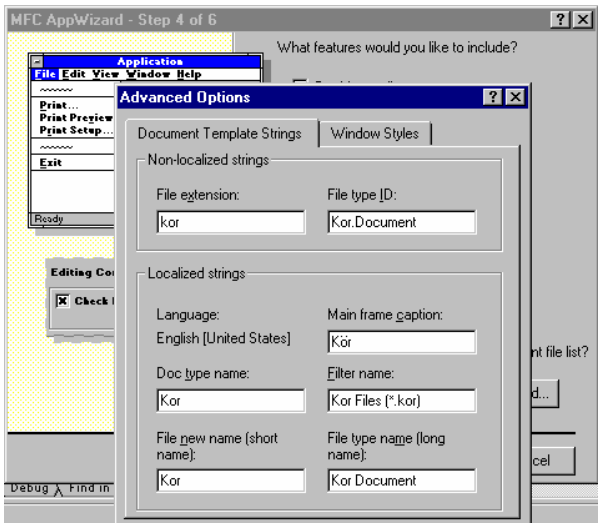
A registry entry fájl szerkezete:

Első sora kötelezően REGEDIT. A megjegyzéseket ';' után írhatjuk, a kulcs és változó bejegyzések mindegyikét külön sorba írjuk.

```
REGEDIT
;Ezt a .reg fájlt használja a My alkalmazás setup programja.
HKEY_CLASSES_ROOT\.my = My.Document
...
```

Registry Entry fájl létrehozása az alkalmazásvarázslóval.

Amikor az alkalmazásvarázslóban a 4. lépésnél az Advanced Options párbeszédablakban beállítjuk a fájl kiterjesztést..., egy registry entry fájlt hozunk létre (7.3. ábra).



7.3. ábra A Registry Entry fájl létrehozása az alkalmazásvarázslóval

A beállítás után létrejövő Kor.reg fájl tartalma:

```
REGEDIT
; This .REG file may be used by your SETUP program.
; If a SETUP program is not available, the entries below will be
;   registered in your InitInstance automatically with a call to
;   CWinApp::RegisterShellFileTypes and
COleObjectFactory::UpdateRegistryAll.

HKEY_CLASSES_ROOT\.kor = Kor.Document
HKEY_CLASSES_ROOT\Kor.Document\shell\open\command = KOR.EXE %1
HKEY_CLASSES_ROOT\Kor.Document\shell\open\ddeexec = [open("%1")]
HKEY_CLASSES_ROOT\Kor.Document\shell\open\ddeexec\application =
KOR
    ; note: the application is optional
    ;  (it defaults to the app name in "command")

HKEY_CLASSES_ROOT\Kor.Document = Kor Document
```

7.3. A registry módosítása az alkalmazásból

Ez a leginkább használható módszer, hogy megőrizzük alkalmazásunk beállításait az egymás utáni futtatások között. Például a főablak elhelyezkedését, méretét... elraktározhatjuk.

Az MFC a CWinApp osztály függvényeivel biztosítja a registry elérésének a támogatását. Természetesen API hívások is rendelkezésünkre állnak.

Az első függvény, amit meg kell hívnunk a registry használatához a ***CWinApp::SetRegistryKey()***. A hívást az alkalmazásosztály InitInstance metódusába célszerű beírni. A függvény paramétere a szoftver gyártójának neve. **A függvény hatására a registryben, ha még nincs ilyen, akkor létrejön egy kulcs a gyártó nevével és egy alkulcs (subkey) az alkalmazás nevével. Amikor írunk a registrybe, ez alá a kulcs alá kerülnek a kulcsok, amikbe a változókat elhelyezzük.** A bejegyzések a következő formájúak lesznek.

HKEY_CURRENT_USER \ Software \ <Gyártó név> \ <Alkalmazás név> \ <Kulcs név> \ <Változó név>

A függvény hatására beállítódik az alkalmazásosztály m_pszRegistryKey adattagja, melyet a registry írásakor, olvasásakor használnak a metódusok az írás helyének meghatározására. **Ha ezt a függvényt meghívtuk, akkor az utoljára megnyitott fájlok listája (MRU most recently-used) is tárolásra kerül a registryben** (7.4. ábra).

Az újabb verziókban az alkalmazásvarázsló SDI / MDI alkalmazásoknál beírja az alkalmazásosztály InitInstance metódusába a SetRegistryKey hívást.

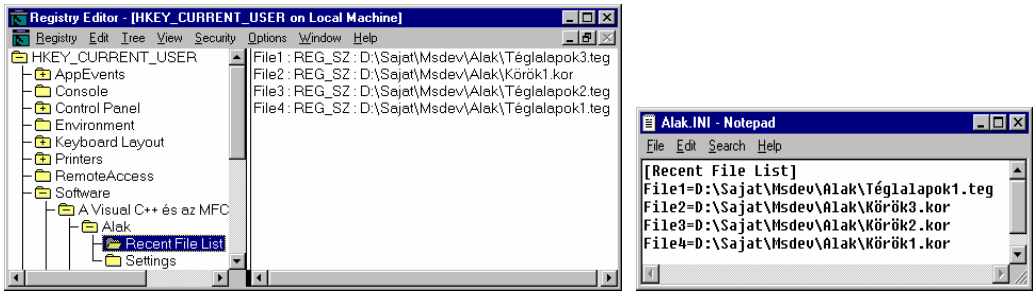
```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));

LoadStdProfileSettings();
```

```
// Load standard INI file options (including MRU)
```

Csak át kell írnunk saját adatainkra a SetRegistryKey függvény paraméterét. A ***LoadStdProfileSettings()*** ahogy a mögötte álló megjegyzésből is kiderül, az MRU fájlok betöltését végzi. Paramétere az MRU fájlok maximális száma, alapértelmezett **UINT** *nMaxMRU* = **_AFX_MRU_COUNT** az alkalmazásvarázsló 4. lépésében az Advanced nyomógomb melletti mezőben meghatározott érték. Ha 0 paraméterrel hívjuk, nem hoz létre MRU listát.

Megjegyzés:

Ha egy .ini fájlt akarunk létrehozni, ne hívjuk meg a SetRegistryKey függvényt! Az .ini fájl neve az alkalmazás neve lesz, és a Windows alkönyvtárba kerül (7.4. ábra).



7.4. ábra Az MRU lista a registryben és az ini fájlban

7.3.1. Adatok írása a regisztrybe

A CWinApp osztály négy függvényt biztosít a registry írásához:

- ***WriteProfileString()***: Egy sztringet ír a registrybe. Három paramétere van: a kulcs neve, a változó neve és a változó értéke. Ha nincs még ilyen nevű kulcs, akkor létrehozza. Ha a változónak NULL értéket adunk, a változó neve törlődik a bejegyzések közül.
- ***GetProfileString()***: Egy sztringet kiolvas a registryből. Három paramétere: a kulcs neve, a változó neve, a visszatérési érték, ha nincs ilyen változó. A harmadik paraméter alapértelmezett, default értéke NULL. A függvény visszatérési értéke CString típusú, és ha megtalálta a változót, annak értékével tér vissza, egyébként a harmadik paraméterrel.
- ***WriteProfileInt()***: Egy integert kiír a registrybe.
- ***GetProfileInt()***: Egy egészet beolvas a registryből. Mivel visszatérési értéke UINT, ha előjeles egészet tároltunk, konvertálni kell int-re! Visszatérési értéke

itt is a harmadik paraméter, ha nem találja a bejegyzést. Ha az adott bejegyzés létezik, de nem integer, akkor 0-val tér vissza.

A GetProfile függvények harmadik paraméterébe adjuk meg azt az értéket, ami eddig a konstruktorban adott kezdőérték volt! Ha a függvény nem találja a bejegyzést, akkor ugyanis ezzel tér vissza.

A függvények az alkalmazás bármely metódusából hívhatók, mivel az ***AfxGetApp()*** metódussal megkaphatjuk az alkalmazásobjektum mutatóját. Ha nem egész számot akarunk tárolni, használjuk a WriteProfileString metódust! Előtte alakítsuk át a változót a CString osztály Format tagfüggvényével sztringgé!

7.3.2. Dokumentumok megnyitása duplakattintásra

Ha meg akarjuk engedni alkalmazásunknak, hogy az általa kezelt dokumentumokon a felhasználó duplakattintására megnyíljon, akkor ezt a ***CWinApp::RegisterShellFileTypes()*** metódussal tehetjük meg. A tagfüggvény a Windows File Manager számára regisztrálja a dokumentumsablonban található összes fájltípust. Tehát az alkalmazásosztályunk ***InitInstance()*** metódusában az ***AddDocTemplate()*** hívások után írjuk be. Hívása előtt kötelező az ***EnableShellOpen()*** meghívása a főablakra!

7.3.3. Az utolsó használt dokumentum nyitása indításkor

Az itt következő függvényhívásokat is az alkalmazásosztály Initinstance metódusában találjuk. A **CCommandLineInfo** cmdInfo objektumba a ***ParseCommandLine(cmdInfo)*** függvény beolvassa a parancssorból egyenként a paramétereket a ParseParam függvény segítségével. A ***ProcessShellCommand(cmdInfo)*** kezeli az argumentumokat. (Információk az argumentumokról a súgóban.)

Az utolsó használt dokumentumot úgy indíthatjuk el, hogy nevét kiolvassuk a registry **MRU** listájából, és átadjuk a cmdInfo objektumnak, mintha a parancssor argumentuma lett volna. A ProcessShellCommand ezután megnyitja az MRU első fájlját.

Tehát a következő kódrészlet kerüljön a ParseCommandLine és a ProcessShellCommand hívások közé:

```
BOOL CMyApp::InitInstance()
{
//...
// Parse command line for standard shell commands, DDE, file
//open
  CCommandLineInfo cmdInfo;
  ParseCommandLine(cmdInfo);
```

```
    if (cmdInfo. m_strFileName== "")
    // Ellenőrizzük, hogy a felhasználó megadott-e dokumentumot a
    //parancssorban!
    {
          CString lastFile=GetProfileString("Recent File List",
                                                "File1",NULL);
          if (lastFile !="")
     {
          cmdInfo.m_strFileName = lastFile;
          cmdInfo.m_nShellCommand = CCommandLineInfo::FileOpen;
     }
    }
    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
          return FALSE;
}
```

A kód már megnyitja az MRU-ban szereplő fájlokat, de ha közben letöröltük őket, akkor nem indul el paraméter nélküli hívásnál. Tehát gondoskodnunk kell róla, hogy ha nem találja az MRU első fájlját, akkor induljon el úgy, mintha üres lenne az MRU.

```
    if (!ProcessShellCommand(cmdInfo))
    {
          cmdInfo.m_strFileName= "";
          cmdInfo.m_nShellCommand = CCommandLineInfo::FileNew;
          if (!ProcessShellCommand(cmdInfo))
                return FALSE;
    }
```

7.4. Kérdések

1. Milyen szerkezetben tárolja a regisry az adatokat?
2. Milyen eszközökkel módosíthatjuk a regisztrációs adatbázist?
3. Milyen beállításokat végezhetünk el az alkalmazásból?

7.5. Összefoglalás

- A regisstryben tárolhatunk rendszerszintű és a felhasználó által létrehozott beállításokat. **A registry binárisan tárolja az adatokat**.
- **A regisztrációs adatbázis hierarchikus felépítésű**. A fő kulcsai HKEY_ kezdetűek. A vezérlőpultból és az alkalmazásokból állíthatjuk értékeit. Közvetlen szerkesztés lehetőségét kínálja a **RegEdit** és a **RegEdt32** szerkesztő. Telepítéskor a .reg kiterjesztésű **registry entry fájl**ok írják be a szükséges adatokat a registrybe.

- Az alkalmazásból is módosíthatjuk a registry bejegyzéseit. A ***SetRegistryKey()*** felveszi a szoftver gyártó nevét és az alkalmazás nevét mint kulcsot a registrybe. A függvény hatására tárolódik az **MRU** a registryben. A ***LoadStdProfileSettings()*** tölti be az MRU-t az alkalmazásba. A registrybe a ***WriteProfileString()*** és a ***WriteProfileInt()*** CWinApp metódusokkal írhatunk, és a ***GetProfileString()***, ***GetProfileInt()*** metódusokkal olvashatjuk ki az adatokat.

- Beállíthatjuk, hogy a dokumentumon duplán kattintva megnyíljon a hozzá rendelt alkalmazás. A dokumentumot drag & drop technikával az alkalmazás fölé vonszolva nyíljon meg a dokumentum! Beállíthatjuk azt is, hogy az utoljára használt dokumentummal nyíljon meg legközelebb az alkalmazás.

"INFORMATICS IS LIKE LOVE,

YOU CANNOT LEARN IT FROM BOOKS."

(Graffiti on a glare screen)

(Az informatika olyan, mint a szerelem, könyvekből nem tanulható!)

GYAKORLATOK

1. Gyakorlat Bevezető feladat

A könyv gyakorlatok része többnyire egymásra nem épülő feladatokat tartalmaz. Az első kettő közülük tekinthető bevezetőnek. Az első fejezethez tartozó Hello alkalmazás célja inkább ismerkedés a fejlesztői környezettel, mint egy megvalósításra váró probléma megoldása. A következő feladatban pedig épp a fordítottja történik. Alig használjuk a fejlesztői környezet lehetőségeit, inkább a kézzel megírt kódsorokon keresztül ismerkedünk egy alkalmazás szerkezetével. Fejlesztéseink során használjuk a fejlesztői környezet kínálta lehetőségeket, de lesz rá példa bőven, amikor kézzel kell megírnunk az egyes kódrészleteket.

A Hello alkalmazás készítése során egy nagyon egyszerű alkalmazás látványos részeit módosítjuk. Kóstolgatjuk a Visual C++-t.

1.1. Hello alkalmazás

Ezen a gyakorlaton elkészítjük a Visual C++ felületet használó első alkalmazásunkat. Használjuk az **alkalmazásvarázsló**t (AppWizard), megvizsgáljuk a rendszer által generált fájlokat, szerkesztjük a párbeszédablak felületét. Elkészítjük a nyomógombokhoz a kezelő függvényeket. Módosítjuk a program ikonját, lehetővé tesszük az ablak minimalizálását, maximalizálását. Megtanuljuk, hogyan nyithatunk meg egy régebben írt projectet.

➢ Az első feladat előkészítése

Hozzon létre egy könyvtárat, melyben a tanulás során a Visual C++-ban írt programjait tárolni fogja! Ha nem teszi, az alkalmazásvarázsló felkínálja a MyProjects alkönyvtárat.

➢ Indítsuk el a Visual C++ alkalmazást!

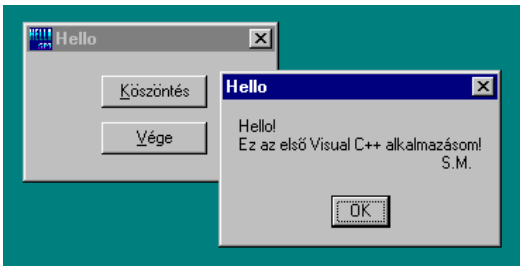
Start

Programs

Microsoft Visual Studio

Microsoft Visual C++

Az 1.Gyakorlat 1. ábra a kész futó alkalmazás képét mutatja. Ha a köszöntés gombot leütjük, megjelenik az alsó ablak.



1. Gyakorlat 1. ábra A Hello alkalmazás

➢ Készítsünk egy új projektet!

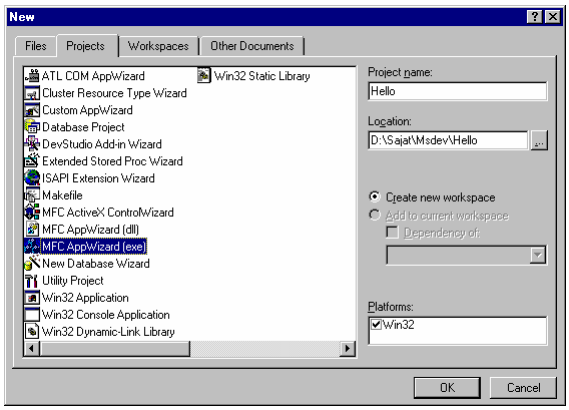
File menü

New menüpont

Projects fül

MFC AppWizard (exe) **Projekt name: Hello** **OK**

Itt állíthatjuk be szükség esetén a project helyét a könyvtárstruktúrában, a Location mezőben.



1. Gyakorlat 2. ábra Új alkalmazás létrehozása

A 'Create new workspace' választógomb alapértelmezett, s nekünk jó is ez a beállítás. Lehetőségünk lenne több projektben is dolgozni egyetlen workspace-t használva, nem élünk vele. (1.Gyakorlat 2. ábra)

Az alkalmazásvarázsló – és más varázslók is – jelentős mennyiségű kérdésre várnak választ, de többnyire rendelkeznek alapbeállításokkal. Jelen esetben a kérdések nagy részére még nem tanultuk a választ, így sokáig tartana (és esetleg értelmetlen lenne, a tananyag felépítését felborítaná) ha a válaszok magyarázatába kezdenénk. Meg kell szoknunk a lényegre koncentrálást, mert különben elveszünk a részletekben. A tanulás előrehaladtával egyre több kérdést fogunk megérteni, s tudatosan kiválasztani a helyes megoldást.

A könyvben a gyakorlatok során nem említjük azokat a kérdéseket, melyekre az alapbeállítás a válasz. Ha megismerjük az alkalmazásvarázsló működését, többnyire fejből tudjuk, hogy csak az utolsó lépésben kell valamit módosítanunk, s addig olvasatlanul lépünk át ablakokat, vagy már az első lépés után a Finish gombot választjuk.

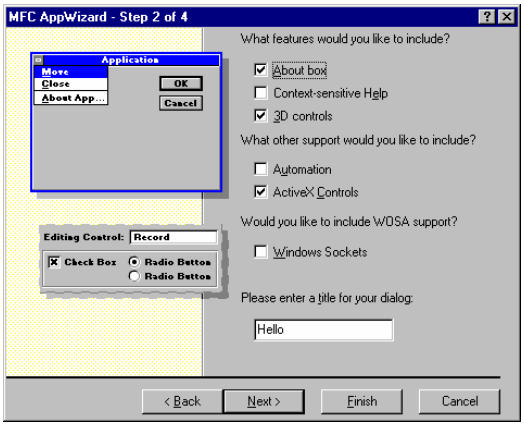
Feladatmegoldásaink során hatékonysági szempontok miatt jellemzően nem módosítjuk az alapbeállításokat. A feladatok egyszerűek, nem fejleszthetők jelentősen a hatékony lehetőségek kihasználásával. Megoldásuk során a bemutatni kívánt problémára koncentrál a leírás.

Elindul az alkalmazásvarázsló MFC AppWizard Step1 ablak. (A választások után Next segítségével léphetünk tovább.)

Az alkalmazás típusa: Dialog based **nyelve: angol** (magyar nincs)

Speciális sajátosságok: Minden alapbeállítás

Az ablak alján megadhatjuk az alkalmazás címsorában megjelenő címét. Ez jelen esetben megegyezik a projekt nevével, Hello. (1.Gyakorlat 3. ábra)



1. Gyakorlat 3. ábra Az alkalmazásvarázsló

A tárgykódban megjegyzések: Yes, please.

Dinamikusan szerkesztett (As a shared DLL)

Az MFC-t csatolhatjuk statikusan és dinamikusan alkalmazásunkhoz. **A dinamikus csatolás előnye a kisebb .exe fájl, de futtatásához minden gépen mellékelni kell a csatolt DLL-t**. Az 'As a statically linked library' választás esetén a kész alkalmazás futtatásához nincs szükség az MFC-t tartalmazó DLL-re. (Olyan gépen is fut, ahol nincs telepítve a Visual C++, pontosabban nem érhető el a használt .dll.) Ekkor azonban a statikusan beszerkesztett könyvtárak miatt a futtatható állomány mérete jelentősen nő. Ha több különböző, de azonos DLL-t használó alkalmazásunk fut ugyanazon a gépen, akkor előnyösebb az alapbeállítást, a dinamikus szerkesztést választani.

Milyen osztályok és fájlok kellenek: Alapbeállítás.

Elfogadjuk a felkínált osztályokat és a tároló fájlok neveit, de nézzük át őket. A fehér hátterű mezők módosíthatóak.

Finish - A beállítások újra átnézhetők, majd OK.

A Hello feladat kerete elkészült.

A Workspace párbeszédablak alján három fül található. (1.Gyakorlat 4. ábra)

- Első a ClassView mellyel a feladat osztályait nézhetjük meg és szerkeszthetjük.
- ResourceView (Erőforrások, dialógus dobozok, menük, ikonok, BMP képek) nézete és szerkesztése, tehát a vizuális környezet beállításai.

- FileView mutatja a feladat összes fájlját.

Láthatóan alig tettünk valamit, máris rengeteg fájl, osztály és erőforrás készen áll a projektben. Nézzük át őket!

Build menü

Set Active Configuration

Release (Rövidebb, gyorsabb.) **OK**

Build menü

Build Hello.exe

Lent az Output ablakban követhetjük a fordítás szerkesztés lépéseit.

Build menü

Execute Hello.exe 

Az üres alkalmazás már futtatható.

A Hello.exe fájl a megadott könyvtár Hello alkönyvtárában található, mivel alapbeállításban **release** volt beállítva a fordítás és szerkesztés előtt, tehát az exe fájl a Hello\Release könyvtárban található.

Ha a feladat fordítása és szerkesztése előtt a debug opciót állítottuk volna be, akkor a rendszer a Hello könyvtárban a **Debug** alkönyvtárat hozta volna létre, benne a nyomkövetéshez szükséges fájlokkal, s itt lenne a hello.exe is, persze nagyobb méretben.

Ha munkánk során kiderül, hogy szükségünk van nyomkövetésre, a Build menü / Set Active Configuration menüpontjában felkínálja a Debug választását is. Természetesen az alkalmazást újra fel kell építeni, a Build menü / Build Hello.exe parancsával.

A fejlesztés során a Debug beállítást szokás használni, mely a program tesztelését támogatja, a kész programot aztán release-ben fordítva szokás használni.

➢ A látható felület átalakítása

Workspace ablak

ResourceView fül

Hello resources kibontva

- Dialog
- Icon

1. Gyakorlat 4. ábra ResourceView

- String Table
- Version Lehetőségeket találjuk.

Nyissuk ki a párbeszédablakot (Dialog)

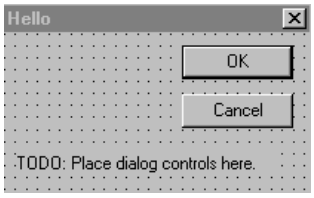
IDD_ABOUTBOX Ez az About párbeszédablak azonosítója.

IDD_HELLO_DIALOG A Hello alkalmazás főpárbeszédablakának azonosítója. (IDD = IDentifier Dialog)

Kattintsunk kétszer rá, hogy szerkeszthető legyen a Hello program főablaka!

Láthatjuk, három vezérlő már található a párbeszédablakon:

- OK gomb
- Cancel gomb
- Egy állandó szöveg kiíró.



Kattintsunk az OK gombra, majd üssük le a Delete billentyűt!

1. Gyakorlat 5. ábra Hello Dialog

Ismételjük meg mindezt a Cancel gombbal és a szöveggel!

Csökkentsük az ablak méretét szélének behúzásával akkorára, hogy a tervezett két nyomógomb kényelmesen elférjen rajta!

A két új nyomógomb elhelyezéséhez használjuk a vezérlő eszköztárat (Controls)! (1.Gyakorlat 6. ábra)

Húzzuk az eszköztárból az egérrel a nyomógombot a megfelelő helyre!

Kattintsunk a jobb egérrel a Button1 gombra, s a legördülő menüből válasszuk ki a tulajdonságokat (Properties)!

A gomb szövegét (Caption) módosítsuk Köszöntés -re!

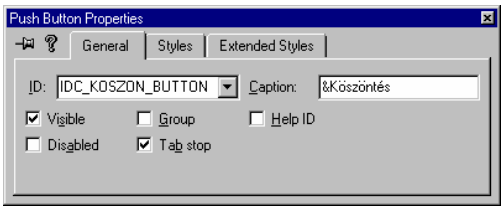
A gomb azonosítóját is célszerű átnevezni. (A későbbiekben megtérül, ha beszédes azonosító neveket használunk.) Pl. IDC_KOSZON_BUTTON.

Megállapodás:

1. Gyakorlat 6. ábra

Az azonosítókat csupa nagybetűvel adjuk meg. Hagyományosan a #define kulcsszó után szerepelő azonosítók csupa nagybetűs nevek.

Ha egy név valamelyik betűje elé az & jelet tesszük a Caption beviteli mezőben, akkor a gomb az ALT és az adott betű leütésével is választható. (1.Gyakorlat 7. ábra)



1. Gyakorlat 7. ábra A tulajdonságablak

Zárjuk be a tulajdonságok párbeszédablakot! A gomb méretét és helyét az alkalmazásokban megszokott módon, drag&drop segítségével állíthatjuk be.

Készítsük el az előbbi módon a Vége gombot is!

Munkánk során használhatjuk a teljes képernyőt is. Válasszuk a View menü Full Screen parancsát! Kilépni az ESC gomb segítségével lehet.

Mentsük el az elkészített főablakot a File menü Save parancsával!

Bár nem írtunk egyetlen sor kódot sem, a látható felülettel elkészültünk. Nézzük meg a kész ablakot milyen lesz futás közben! Ehhez nem szükséges újrafordítani a programot, a Dialog eszköztár Test gombja futtatás nélkül mutatja meg a futáskor várható képet. 

➢ Kód csatolása a gombokhoz

View menü

Osztályvarázsló (Class Wizard) 

Az osztályvarázsló közvetlenül is elérhető a Ctrl W gyorsgombbal vagy ikonjával.

Message Map fül (Üzenettérkép)

Az Osztály neve kombipanelben a legördülő listadobozból a CHelloDlg osztály legyen kiválasztva, melyet a rendszer a Hello program főablakához rendelt. (A kezdő C betű a Class szó rövidítése.) Ezt az osztályt az MFC AppWizard hozta létre.

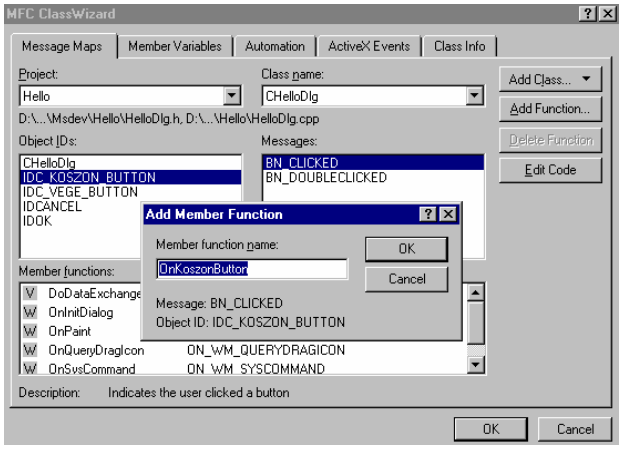
Válasszuk az IDC_KOSZON_BUTTON-t az objektumazonosítók listadobozából! Két lehetséges üzenetet küldhetünk neki: (1.Gyakorlat 8. ábra)

BN_CLICKED Ezt válasszuk ki!

BN_DOUBLECLICKED

Válasszuk az Add Function gombot!

A rendszer felkínálja az OnKoszonButton függvényt, melyet ha akarunk átnevezhetünk, de fölösleges. Tehát OK. Ez a függvény fog végrehajtódni a Köszöntés gomb lenyomása után.



1. Gyakorlat 8. ábra Az osztályvarázsló

Az osztályvarázsló csak a függvény vázát készíti el, a valóságos kód megírása a mi feladatunk. Ehhez az Edit Code gombot kell választanunk. Az osztályvarázsló válasza, hogy megnyitja a HelloDlg.cpp fájlt az OnKoszonButton() függvénnyel, s a kurzort a megírandó kód helyére állítja.

A kódban a következő megjegyzéssorral találkozunk:

```
// TODO: Add your control notification handler code here
```

Ez a sor mutatja, hol szokás az adott függvény kódját kiegészíteni, összetettebb kódrészletnél komoly segítséget jelent. Ki van jelölve, így ha azonnal elkezdjük a kód írását, eltűnik a forráskódból. Ha meghagyjuk, akkor a későbbiekben tájékoztat bennünket arról, hogy hol kezdtük írni a saját kódunkat, s amikor hibás a kód, mely részeit kell átnéznünk. E könyvben is több helyen olvasható ez a megjegyzéssor. Az .exe fájl mérete ettől nem növekszik, hisz a fordító a megjegyzéseket figyelmen kívül hagyja.

```
void CHelloDlg::OnKoszonButton()
{
 MessageBox("Hello!\nEz az első Visual C++
alkalmazásom!\n\t\t\tS.M.");
}
```

File menü, Save All.  (Most több fájlban dolgoztunk egyszerre.)

Build menü, Execute Hello.exe. (Ctrl F5) (A rendszer érzékeli a változásokat, és rákérdez, akarom-e újra fordítani, szerkeszteni az alkalmazást?) Igen.

Most a Vége gomb kódját is készítsük el! Az OnVegeButton() függvénybe csak az OnOK() függvényhívást írjuk be, mely a CHelloDlg osztály tagfüggvénye, és becsukja a Hello program főablakát. A CHelloDlg osztály a CDialog osztály utódosztálya, s az OnOK függvényt a CDialogból örökölte. A függvény írásakor figyeljünk a kis- és nagybetűkre! (A gombot az OK gomb szövegének megváltoztatásával – Vége feliratúra – is elkészíthettük volna.)

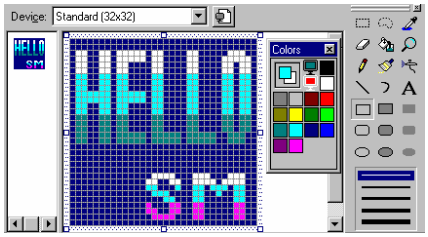
➢ A Hello program ikonja

Az alkalmazás ikonja futáskor a bal felső sarokban található. Még alapértelmezett.

Workspace / ResourceView

Icon

IDR_MAINFRAME



1. Gyakorlat 9. ábra Az ikonszerkesztő

A grafikus és a színek eszköztár segítségével az ikon képe a szokott módon szerkeszthető.

File menü / Save All

Build menü / Execute Hello.exe.

Futás közben az alkalmazás bal felső sarkában az általunk szerkesztett ikont látjuk. (1.Gyakorlat 1. ábra) Alul a Taskbar-on megjelenített ablakban is. De ha megnézzük – a fájlkezelőben a Hello könyvtár Release és Debug alkönyvtárában a Hello.exe előtt nem a saját ikonját jelzi (feltéve, hogy a small icons beállítás érvényes). Ha kitesszük az asztalra, mint futtatható programot, ott már az általunk tervezett ikonnal jelenik meg. Mi a magyarázat a különbségre? Az, hogy **minden ikonnak két megjelenítési módja van**. Az egyik a standard, 32x32 képpontos, a másik a small, 16x16 képpontos. Ha a fájlkezelőben a View / Large Icons beállítás érvényes, az általunk rajzolt Hello ikont kapja az alakalmazás. Amint azt az 1.Gyakorlat 9. ábra beállítása mutatja, mi csak a 32x32 képpontos ikont módosítottuk. Tegyük meg ezt a 16x16 pixelessel is!

Megjegyzés:

Az ikon rajzolásakor nem feltétlen előnyös a sok színárnyalat használata, mert a Visual C++ beállításától függően csak az aktuális palettának megfelelő színeket menti el az erőforrás-szerkesztő. (A többi szín helyett a hozzá legközelebb állót helyettesíti.) Tehát a színek keverése fölösleges munkának bizonyulhat.

Megjegyzés:

Ha az asztalra szánjuk ikonunkat, tervezésekor gondoljunk a kép bal alsó sarkában megjelenő shortcut jelzésre! A kis nyíl miatt a bal oldali 10 X 10 -es négyzet nem fog látszani az ikonból.

➢ Egy régi projekt megnyitása

Bezárjuk a projektünket: File menü Close Workspace paranccsal.

Megnyitunk egy újat: File menü Open Workspace paranccsal.

Pl. a Hello könyvtár Hello.**dsw**-t választva.

➢ Kisebb módosítások

1. Adatok beírása az About párbeszédablakba.

A Workspace / ResourceView / Dialog / IDD_ABOUTBOX választása után húzzunk statikus szöveget az ablakba (Aα ikon)! Majd írjuk bele a kívánt szöveget! A többi szöveget is módosíthatjuk, csak rá kell kattintani a megfelelő beviteli mezőre.

2. A teljes képernyő és a minimalizáló ikon elhelyezése a bal fölső sarokba.

Válasszuk a Workspace / ResourceView / Dialog / IDD_HELLO_DIALOG párbeszédablak szerkesztését! Az ablak olyan részén, ahol nincs vezérlő a jobb egérgombbal kattintva, a legördülő menüből a Properties-t választva, a Styles fülben a Minimize box, Maximize box jelölőnégyzeteket kiválasztva láthatjuk az ablak címsorának jobb oldalán a megjelenő ikonokat.

Az alkalmazás fordítása után futáskor is ellenőrizhetjük beállításaink működését.

Feladat:

A program a Vége gomb leütése után köszönje meg, hogy együtt dolgozhatott velünk, s csak azután legyen vége az alkalmazásnak!

2. Gyakorlat Az alkalmazás szerkezete

A minimális alkalmazást kézzel írjuk, a fejlesztői környezet lehetőségeit alig használva. Célja az, hogy megértsük, hogyan működik egy alkalmazás. Úgy gondolom, ez nagyon fontos eleme a továbblépésnek, s az alkalmazásvarázsló által generált kódrészletek értelmezésének.

2.1. Kézzel írt minimális alkalmazás

Ezen a gyakorlaton egy kézzel írt alkalmazást készítünk el, melynek célja, hogy segítségével könnyebben megértsük a Visual C++ alkalmazások szerkezetét.

2. Gyakorlat 1. ábra A Min alkalmazás ablakai

- **A project létrehozása**

  File menü

  New

Projects fül

Win32 Application **Project name: Min** **OK.**

An empty project **Finish**

- **Használjuk az MFC-t!**

Project menü

Settings

Settings For:

All Configurations

General fül

Microsoft Foundation Classes:

Use MFC in a Static Library / OK.

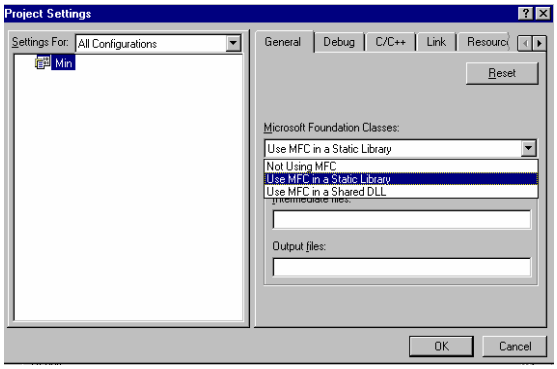
Itt határozhatjuk meg az alapbeállításokat. Pl. használjuk vagy nem az MFC-t mint statikus könyvtárat, vagy mint dinamikusan beszerkesztett könyvtárat. (2. Gyakorlat 2. ábra)

Build menü

Set Active Configuration

Debug // alapértelmezés

A Release és Debug közti különbséget az előző feladat elején tárgyaltuk.



2. Gyakorlat 2. ábra Az MFC használatának beállítása

➢ Fájl beszúrása a projectbe

Workspace

FileView fül

Min files jobb egérgomb

Add Files to Project

File name: Min.cpp OK

Ez a fájl nem létezik, akarja a hivatkozását a projecthez csatolni? OK.

Nyissuk meg a Min files-t, ott lesz benne a Min. Duplán rákattintva: A fájl nem létezik, akarja, hogy létrehozzuk? OK. Írhatjuk a kódot.

Ily módon lehet már meglevő fájlokat csatolni a projektünkhöz. Nem létező fájl esetén a rendszer figyelmeztet bennünket, ahogy tapasztaltuk. A csatolás hatására a Workspace fájljai közt megjelenik a csatolt fájl.

Ha nem létező fájlt csatoltunk, és azt meg akarjuk nyitni, ha arra a kérdésre, hogy létrehozza-e – igent válaszolunk, akkor jön létre a könyvtárban fizikailag is a fájl (*.cpp, *.h).

A következő sorokat írjuk be a min.cpp-be:

```
//
// Minimal hand-coded MFC application
//

#include <afxwin.h>

// Class declarations ////

class CMinApp : public CWinApp
{
};

// Class implementations ////

// Application //

CMinApp myApp;
```

➢ A WinMain tagfüggvény

Az osztálykönyvtár keretszolgáltatásának része.

WinMain → **InitApplication**
→ **InitInstance**
→ **Run**
→ **ExitInstance**

Mint minden Windows alkalmazás, a mi alkalmazásunk is tartalmaz egy WinMain függvényt (afxwin.h). Nekünk nem kell megírnunk, biztosítja az osztálykönyvtár, és meghívódik, ha az alkalmazás elkezd futni. A WinMain hívja az alkalmazás tagfüggvényeit, hogy inicializálja és futtassa az alkalmazást.

Az alkalmazás inicializálásához a **WinMain** hívja az alkalmazás **InitApplication** és **InitInstance** tagfüggvényeit. Ahhoz, hogy futtassuk az alkalmazás üzenet-feldolgozását, **WinMain** hívja a **Run** tagfüggvényt, ami a befejezéshez hívja az alkalmazáspéldány **ExitInstance** tagfüggvényét.

➢ A Min alkalmazás futtatása

Build menü / Build Min.exe

Build menü / Execute Min.exe

A végrehajtás csak a rövid időre megjelenő homokórából látszik.

Ha Debug üzemmódban lefuttatjuk az alkalmazást, először Step Into (F11) segítségével lépjünk be az Appmodul.cpp-ből az **AfxWinMain** kódjába (2*F11), és nézzük a winmain.cpp végrehajtását Step Over (F10) üzemmódban!

A fontosabb sorok, melyre felhívnám a figyelmet:

```
CWinThread* pThread = AfxGetThread();
CWinApp* pApp = AfxGetApp();
AfxWinInit(...);
pApp->InitApplication();
pThread->InitInstance();
pThread->Run();
AfxWinTerm();
```

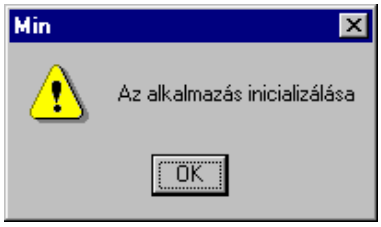
Figyeljük meg, ha hosszabb ideig egy-egy változón... tartjuk a kurzort, annak értéke látható lesz!

Az InitInstance() tagfüggvénye az alkalmazásnak üres. Az `if (!pThread->InitInstance())` sornál F11-gyel beléphetünk a kódba. Az appcore.cpp-ben a következőt találjuk:

```
BOOL CWinApp::InitInstance()
{
 return TRUE;
}
```

Írjuk felül!

```
//
// Minimal hand-coded MFC application
//


#include <afxwin.h>

// Class declarations ////

class CMinApp : public CWinApp
{
 public:
 BOOL InitInstance();
};
```



2. Gyakorlat 3. ábra Az első üzenetablak

```
// Class implementations ////
BOOL CMinApp::InitInstance()
{
 AfxMessageBox("Az alkalmazás inicializálása");
 return CWinApp::InitInstance();
}

// Application //

CMinApp myApp;
```

Futtatáskor az üzenetablak megjelenik, de ne feledjük: az alkalmazásnak még nincs főablaka!

➢ A Min főablaka

Hozzunk létre egy CMinFrameWnd osztályt, melyet az alkalmazás használ!

Mivel az inicializáláskor az ablak rejtett, az aktivizálás során láthatóvá is kell tennünk (ActivateFrame())!

Az InitInstance-ban *dinamikusan* hozzuk létre a főablakpéldányt!

```
//
// Minimal hand-coded MFC application
//
 #include <afxwin.h>
// Class declarations ////

class CMinApp : public CWinApp
{
public:
 BOOL InitInstance();
};
```

```
class CMinFrameWnd:public CFrameWnd
{
public:
 void ActivateFrame(int nCmdShow = -1);
};

// Class implementations ////

BOOL CMinApp::InitInstance()
{
 AfxMessageBox("Az alkalmazás inicializálása");

 CMinFrameWnd* pMFW = new CMinFrameWnd;
 pMFW->Create(NULL,"Min alkalmazás");
 pMFW->ActivateFrame();

 m_pMainWnd = pMFW;

 return CWinApp::InitInstance();
}

void CMinFrameWnd::ActivateFrame(int nCmdShow)
{
 AfxMessageBox("Az alkalmazás keretablaka");
 CFrameWnd::ActivateFrame(nCmdShow);
}

// Application //
CMinApp myApp;
```

Az ActivateFrame függvény deklarációja:

```
CFrameWnd:: ActivateFrame ( int nCmdSow = -1 ) :
```

Az nCmdShow paraméter meghatározza a CWnd::ShowWindow függvénynek átadott paramétert. Alapértelmezésben a keret látható, és teljesen helyreállítódik.

Akkor hívjuk e tagfüggvényt, ha a keretet aktivizálni akarjuk vagy helyre akarjuk állítani oly módon, hogy a felhasználó számára látható és elérhető legyen.

Az m_pMainWnd a CWinApp osztály adatmezője, inicializálása az InitInstance feladata.

Látszólag fölösleges a pMFW segédváltozó bevezetése, hisz kapásból az m_pMainWnd-ben is létrehozhattuk volna a keretablakot. Ne feledjük azonban, az m_pMainWnd mutató CWnd* típusú, s így többször kellett volna a típuskényszerítést alkalmaznunk! A későbbiekben, ha az m_pMainWnd adatmezővel dolgozunk, ne feledkezzünk meg a típuskényszerítésről a tagfüggvényhívások és az adattag-hivatkozások során!

! Ha elfelejti a pMFW->ActivateFrame() hívást (a zárójel is fontos!), a főablak létrejön, de nem látjuk a képernyőn, így nem lehet bezárni! A javítás után nem lehet az exe fájlt szerkeszteni, hisz futó programot nem lehet fölülírni! (Természetesen a Ctrl+Alt+Del, majd Task Manager segítségével le lehet állítani a futást.)

Dinamikusan létrehozott keretablakot soha explicit módon ne töröljön! Hol (melyik tagfüggvényben) kellene ezt megtenni? A CWinApp::ExitInstance feladata ez a művelet.

A Create függvény nevéből tudhatjuk, hogy feladata az objektum létrehozása. Ha a súgóban rákeresünk a CFrameWnd::Create-re, az MFC-re jellemző függvénnyel találkozunk. A függvénynek rengeteg, egyelőre számunkra érthetetlen argumentuma van, de a harmadik argumentumtól alapértelmezett paraméterekkel rendelkezik, így nem kell az összes paramétert megadnunk hívásakor. A következő szakasz is csak az első három argumentum részleges magyarázatát tartalmazza, mégis bepillantást jelent egy gyakran hívott és meglehetősen összetett függvény működésébe.

BOOL Create(LPCTSTR *lpszClassName*, LPCTSTR *lpszWindowName*, DWORD *dwStyle* = WS_OVERLAPPEDWINDOW, const RECT& *rect* = rectDefault, CWnd* *pParentWnd* = NULL, LPCTSTR *lpszMenuName* = NULL, DWORD *dwExStyle* = 0, CCreateContext* *pContext* = NULL);

Visszatérési értéke nem 0, ha sikeres; 0 egyébként.

lpszClassName Ha NULL, az előredefiniált alapértelmezett CFrameWnd jellemzőket használja.

lpszWindowName Az ablak neve a címsorban.

A stílusokat a súgó tartalmazza angolul. Az alapstílusok rövid ismertetése itt következik, de további stílusokat is használhatunk, részben az extended styles kategóriából, részben pedig az adott ablakra jellemző stílusok közül, például gombok esetén a button styles.

Styles:

WS_BORDER Keretes ablak.

WS_CAPTION Van címsora (beleértve a WS_BORDER-t). Nem használható a WS_DLGFRAME-mel.

WS_CHILD Gyermekablak. Nem használható a WS_POPUP-pal.

WS_CLIPSIBLINGS Egymáshoz kapcsolja a gyermekablakokat. Ha egyik kap egy WM_PAINT üzenetet, a hozzá csatolt gyermekablakokat kicsatolja a felülírt területről. Csak a WS_CHILD-dal együtt használható.

WS_DISABLED Alapértelmezésben nem engedélyezett.

WS_DLGFRAME Dupla keret, de cím nincs.

WS_GROUP Meghatároz egy első vezérlőt, melyen keresztül a vezérlés mozgatható az ablakok közt. Minden ezután definiált ablak ugyanabba a csoportba tartozik, amíg egy újabb WS_GROUP stílusú vezérlő nem következik. Ez lezárja az előző csoportot, s kezdi az újat.

WS_HSCROLL Függőleges gördítősáv.

WS_MAXIMIZE Maximális méretű ablak.

WS_MAXIMIZEBOX Van maximalizáló ikonja.

WS_MINIMIZE Alapértelmezésben minimalizált ablak. Csak a WS_OVERLAPPED-dal együtt használható.

WS_MINIMIZEBOX Van minimalizáló ikonja.

WS_OVERLAPPED Overlapped ablak. Mindig van címe és kerete.

WS_OVERLAPPEDWINDOW Overlapped ablak WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX és WS_MAXIMIZEBOX stílussal.

WS_POPUP Pop-up ablak. Nem használható a WS_CHILD-dal.

Ne használjuk Create-ben, helyette a CreateEx metódust hívjuk!

WS_POPUPWINDOW Pop-up ablak WS_BORDER, WS_POPUP és WS_SYSMENU stílussal. Kötelező a WS_CAPTION stílus használata, hogy a vezérlő menüt láthatóvá tehessük.

WS_SYSMENU Vezérlő menüdoboza van a címsorban. Csak címsoros ablakhoz használható.

WS_TABSTOP Vezérlők, melyek közt a TAB billentyűvel mozoghatunk.

WS_THICKFRAME Vastag keretes ablak, amivel méretezhetjük az ablakot.

WS_VISIBLE Alapértelmezésben látható ablak.

WS_VSCROLL Vízszintes gördítősáv.

Az Extended styles ld. a helpben.

Néhány példa a **Create** hívására:

```
RECT rect = {50,50,500,500};
pMFW->Create(NULL, "Első alkalmazás",WS_OVERLAPPEDWINDOW, rect);

pMFW->Create(NULL, "Első alkalmazás", WS_SYSMENU | WS_CAPTION);
```

Nem méretezhető (ikonokkal sem)!!!

```
pMFW->Create(NULL, "Első alkalmazás", WS_SYSMENU | WS_CAPTION |
WS_VISIBLE, rect);
```

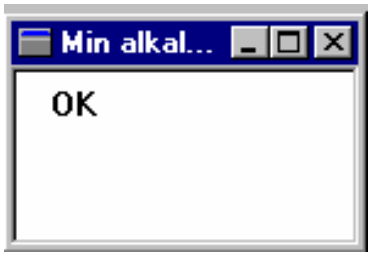
Nem kell hozzá az ActivateFrame() hívás.

A feladat lényegi megoldásával végeztünk, a feladat következő részében a későbbi fejezetek néhány elemét vesszük elő, hogy ötleteinket megvalósíthassuk.

➢ Kiírás az ablakba

```
CDC* pDC = m_pMainWnd->GetWindowDC();
```

A pDC a teljes ablakra mutat, amibe beletartozik a címsor is. A beállítások adatai a GetSystemMetrics API függvény (lásd Help) segítségével lekérdezhetők.



2. Gyakorlat 4. ábra A Min alkalmazás

```
pDC->TextOut(20,30,"OK");
m_pMainWnd->ReleaseDC(pDC);
```

Az InitInstance teljes kódja:

```
BOOL CMinApp::InitInstance()
{
 AfxMessageBox("Az alkalmazás
inicializálása");

 CMinFrameWnd* pMFW = new CMinFrameWnd;
 pMFW->Create(NULL,"Min alkalmazás");
 pMFW->ActivateFrame();

 m_pMainWnd = pMFW;

//Ez a kiírás még nem frissül (pl. ha ráhúzunk egy másik
//ablakot, és nem ismerjük még a DC használatát sem, csak
//azt mutatja, hogy az ablak kész fogadni az üzeneteket.
 CDC* pDC= m_pMainWnd->GetWindowDC();
 pDC->TextOut(20,30,"OK");
 m_pMainWnd->ReleaseDC(pDC);

return CWinApp::InitInstance();
};
```

2. Gyakorlat 5. ábra Színes kiírás

A színek az úgynevezett **RGB** (Red Green Blue) értékükkel adottak. (0,0,0)=fekete, (255,255,255)=fehér. A **COLORREF** típust az RGB(red,green,blue) számhármassal állíthatjuk elő.

```
pDC->SetBkColor(RGB(0,255,0));
pDC->SetTextColor(RGB(255,0,0));
```

A teljes ablak kiszínezését a 5. fejezetben tanuljuk.

Megjegyzés:

Ha az ablak méretét módosítjuk, a kiírás eltűnik. Ha fölé más ablakot nyitunk–zárunk az újra megjelenő ablakunkban nem lesz ott a szöveg. Tehát nem frissül a kirajzolás. Ezt az ablak újrafestésével kapcsolatos problémát a 5. fejezetben tárgyalja a könyv. Az ablak frissítése üzenetkezelést igényel. Jelenlegi tudásunkkal legföljebb annyit tehetünk, hogy nem engedjük méretezni az ablakunkat, és nem engedünk fölé nyitni másikat. (Mi magunk pedig nem mozgatjuk a képernyőn kívülre a feliratot.)

```
RECT rect = {50,50,200,150};
pMFW->Create(NULL,"Min alkalmazás",WS_SYSMENU|WS_CAPTION, rect);
```

- **Mindig legfelül (Top)!**

Ha azt szeretnénk, hogy az ablakunk mindig a többi ablak fölött helyezkedjen el, a stílusát kell megváltoztatnunk. Ezt a legegyszerűbben a PreCreateWindow tagfüggvényben tehetjük. Írjuk felül a CWnd::PreCreateWindow tagfüggvényét a CMinFrameWnd osztályban! Amint a neve is mutatja, ez a függvény meghívódik, mielőtt a Create() az ablakot létrehozná. Soha ne hívjuk direkt módon!

```
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

Visszatérési érték: Nem 0, ha az ablak létrehozása folytatható 0 hiba esetén.

A felülíráshoz keressük meg a helpben a **CREATESTRUCT** struktúra adattagjait s azok leírását!

```
typedef struct tagCREATESTRUCT {
   LPVOID    lpCreateParams;
   HANDLE    hInstance;
   HMENU     hMenu;
   HWND      hwndParent; // szülőablak, ha NULL, főablak
   int       cy;         // magasság
   int       cx;         // szélesség
   int       y;          // bal felső sarok y
   int       x;          // bal felső sarok x
   LONG      style;      // ld. előzőekben
   LPCSTR    lpszName;   // az ablak neve
   LPCSTR    lpszClass;
   DWORD     dwExStyle;  // további stílusok
} CREATESTRUCT;
```

A kód a következőképp módosul:

```
class CMinFrameWnd:public CFrameWnd
{
public:
 void ActivateFrame(int nCmdShow = -1);
 BOOL PreCreateWindow(CREATESTRUCT& cs);
};

BOOL CMinFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{
// cs.x=50; cs.y=50; cs.cx=150; cs.cy=100;
//cs.style= WS_SYSMENU | WS_CAPTION;     // Itt is beállíthatók
 cs.dwExStyle=WS_EX_TOPMOST;
 return CFrameWnd::PreCreateWindow(cs);
};
```

A **WS_EX_TOPMOST** stílus hatására az ablak mindig a többi alkalmazásablak fölött van, ha nem aktív, akkor is!

Az ablak mérete és stílusa itt is beállítható. (A kódban megjegyzés mögött.) A különbség a rect adatai és a cx, cy között abból adódik, hogy a rect a jobb alsó sarok koordinátáit tartalmazza, míg cx és cy a szélesség és magasság méretei.

Próbáljuk ki a `cs.dwExStyle=`**WS_EX_TRANSPARENT** stílus hatását!

Az ablak a Visual C++ fölé nyílik, majd "eltűnik". Próbáljuk, meg megjegyezni, hol van a fejléce, és mozgassuk arrébb! Ha minimalizáljuk a Developer Studio-t, az összes egyéb futó alkalmazáson is eltűnik, az asztalon megtartja a fejlécet, de az ablak belseje ilyenkor is elég furcsán, átlátszón viselkedik! A képek szerkesztése során alkalmazható a módszer.

Állítsuk vissza a TOPMOST esetet!

➢ Feladatok:

Hozzunk létre minimalizáló / maximalizáló ikont az alkalmazás jobb fölső sarkában!

Lehessen a keret segítségével az ablak méretét változtatni!

➢ A project fájljainak feladata:

*.obj A tárgykódot tartalmazó, más projectekbe beszerkeszthető fájl.

*.exe A futtatható állomány.

*.pch (precompiled header) Ha a fordításhoz be van állítva a /Yu (Use precompiled header opció) – alapértelmezés – a compiler az újrafordításkor az adott .cpp fájlban nem hajtja végre az #include "stdafx.h" fölötti sorokat, hanem a My.pch előfordított

kódját használja. Ezáltal lerövidül a fordítási idő. Ha valamit elő szeretnénk fordítani, *mert tudjuk, hogy nem változtatjuk meg*, akkor azt érdemes az stdafx.h-ba tenni.

*.res , *.rc Az .rc fájlt a resource editor hozza létre a szerkesztés során. Az .rc kiterjesztésű fájl az erőforrásleíró. ASCII formátumú erőforrás definíciókat valamint más bináris vagy ASCII formátumú fájlokban elhelyezett erőforrásokra vonatkozó hivatkozásokat tartalmaz. Az rc.exe fordító segítségével fordítható .res bináris állománnyá. Nyissuk meg a Hello. rc fájlunkat auto, majd text módban (Open ablak Open as:)! Próbáljuk ugyanezt a Hello.res fájllal!

*.plg Projekt log fájl az utolsó fordítás és szerkesztés információit tartalmazza html formátumban.

Debug alkönyvtárban ezeken kívül keletkező fájlok:

*.ilk (incremental link) Információk az inkrementális linkeléshez.

*.pdb (program database) Nyomkövetés alatti adatok tárolása.

project.idb (Incremental Compilation) Csak azokat a függvényeket fordítja újra, melyek az utolsó fordítás óta változtak. (Az .obj fájl Enable Incremental Compilation mellett nagyobb méretű.)

VC40.idb azoknál fájloknál, melyeket a projekt nélkül fordítottunk.

További előforduló fájlok:

*.def Szekesztés (Linker) definíciós fájl. Olyan DLL esetén, melynek exportáljuk a függvényeit.

*.aps Bináris resource fájl. (Recource editáláskor használjuk.)

*.clw Az osztályvarázsló állapotát határozza meg.

*.bsc Browser információk forrásfájlja. *.sbr Browser modul inform.

*.vcp Projekt státusz fájl. *.lib Library fájl.

További információk fájl kiterjesztésekről az 1.2.2. elméleti részben, valamint a súgóban érhetők el. A projektünk readme.txt fájlja is tartalmaz a kiterjesztésekre vonatkozó információkat.

3. Gyakorlat Párbeszédalapú alkalmazások

A gyakorlat négy részből áll.

A 3. Gyakorlat 1. rész lépésenként leírja egy párbeszédalapú alkalmazás fejlesztését, végigkövetve a felület szerkesztését, a vezérlőkhöz változók rendelését, majd újabb párbeszédablakok csatolását az alkalmazáshoz. A cél a tökéletes feladatmegoldás helyett az volt, hogy minél több lehetőséget mutasson meg a feladat. Ennek köszönhetően másként kezeli a banán árának változását, mint a körte árváltozását. A megoldás során néhány trükkel is találkozunk, pl. hogyan lehet nyomógombokat választógombként üzemeltetni, vagy hogyan rendelhetünk tömböt több szerkesztődobozhoz.

A 3.Gyakorlat 2. részében a vezérlők kezelésére egy másik módszert látunk (Value típusú helyett Control típusú változókkal), így egy feladat kétféle megoldásával vizsgálhatjuk a különböző módszerek közti különbségeket.

A 3.Gyakorlat 3. rész további – az elmélet feldolgozását segítő feladatot tartalmaz, melyek az olvasó önálló megoldására várnak. A megszerzett ismeret akkor ér valamit, ha hasznosítani tudjuk, ki tudjuk egészíteni saját ötleteinkkel. Amennyiben mégis elakadna az olvasó, a 3. Gyakorlat 4. rész ötleteket tartalmaz a 3. Gyakorlat 3. rész *-al jelölt feladatainak megoldásához.

3.1. Gyümölcsárazó automata

Készítsük el a nagyobb áruházakban található mérleggel összekötött gyümölcsárazó automata szimulációját! Az automata felülete legyen a 3.Gyakorlat 1. ábrán látható! Kezdetben a súlymező legyen üres, és a Fizetendő-mező ne legyen látható! Egyik gyümölcs gombja se legyen lenyomva, és az árakat szürke mezők mutassák! Begépelve a gyümölcs súlyát kilogrammokban (a mérleg mérése helyett), majd kiválasztva a gyümölcs nyomógombját, jelenjen meg a fizetendő összeg (a vonalkód nyomtatása helyett)! Új súly bevitele esetén a fizetendő összeg tűnjön el addig, amíg a gyümölcsöt nem választottuk ki, nehogy megtévesszen minket az előző ár, és ne legyen engedélyezett az előzőleg választott gyümölcs sem!

Lehessen változtatni a gyümölcsök árát is, de ezt csak a jelszót ismerő egyének tehessék meg!

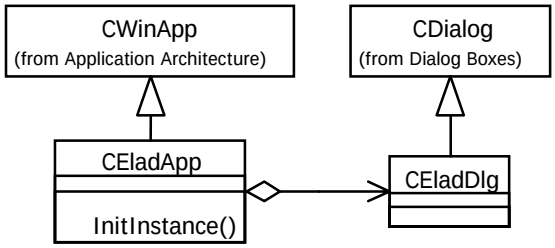
3. Gyakorlat 1. ábra A Gyümölcsárazó automata felülete működés közben

3.1.1. Terv

A feladatot **párbeszédalapú alkalmazás** segítségével oldjuk meg. Az alkalmazásvarázsló generálja a főbb szerkezeti osztályokat. (3.Gyakorlat 2. ábra.) Mivel az alkalmazásvarázslóval ki- vagy bekapcsolható az Aboutbox ablak, és az alkalmazás szempontjából ennek nincs jelentősége, így ezzel az osztállyal nem foglalkozunk.

Az eseménykezelés során az üzenetek továbbítását az operációs rendszer készen kínálja, a mi feladatunk a kezelő metódusok üzenetekhez csatolása és a kód elkészítése.

Az osztálydiagramot Visual Modeler (OOP2. fejezet) segítségével készíthetjük el.



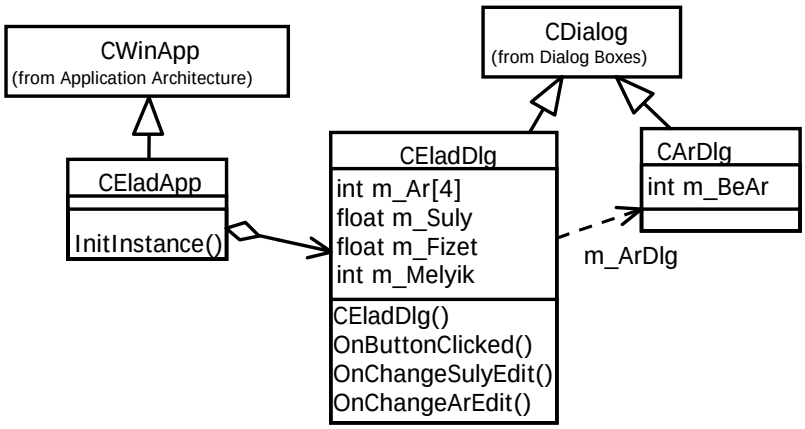
3. Gyakorlat 2. ábra Az alkalmazásvarázsló által generált osztályok

Először is nézzük meg milyen adatokra lesz szükségünk!

- Tárolnunk kell a gyümölcsök árait, a súly és a fizetendő összeg értékét!
- A gyümölcsválasztó-gombokhoz is rendelhetünk egy változót, hogy ne kelljen mindig lekérdeznünk az aktuálisat (int m_Melyik).

Milyen üzeneteket kell kezelnünk?

- A változók inicializálása konstans érték esetén a konstruktorban (CEladDlg()).
- Ha kiválasztjuk valamelyik gyümölcs gombját, jelenjen meg az ára! Legyen a metódus neve: OnButtonClicked()!
- Ha írunk a súly szerkesztőmezőbe, az új gyümölcsöt jelent, tehát ne legyen kiírva a régi ára! OnChangeSulyEdit() a metódus neve.
- Ha módosítjuk a gyümölcsök árát pl. OnChangeArEdit().



3. Gyakorlat 3. ábra Az Elad alkalmazás osztálydiagramja

A gyümölcs árának módosításához használhatunk újabb párbeszédablakot is (CArDlg), melyben a gyümölcs új árát biztosan tárolni akarjuk (m_BeAr). A tervet a 3.Gyakorlat 3. ábra mutatja.

3.1.2. A párbeszédfelület és kezelése

➢ A párbeszédalapú keretprogram elkészítése

File menü **- New** menüpont

Projects

Név beírása**: Elad**

MFC AppWizard(exe).

Location: A könyvtár meghatározása **OK**

Az alkalmazás típusa és nyelve: Dialog based Next

Nem kellenek az ActiveX vezérlők és az About ablak sem.

Írjuk be címnek: Gyümölcsárazó automata Next

MFC Standard,

A tárgykódban megjegyzések: Yes, please

Dinamikusan szerkesztett (As a shared DLL) Next

Milyen osztályok és file-ok kellenek: Alapbeállítás Finish

A beállítások újra átnézhetők. (Itt olvashatjuk el a beszerkesztett DLL nevét.) **OK.**

Build menü / Set Active Configuration / Debug (Ez az alapbeállítás. Ha kész a program, itt kell Release-re állítani, hogy kisebb legyen a kód.)

➢ Az Elad.exe párbeszédablak elkészítése

A képernyőn nyomógombokat kívánunk látni, amelyek úgy viselkednek, mint a választógombok. Vagy elkészítjük őket nyomógombbal, és le kell kezelni minden gomb kiválasztásakor a többi inaktivizálását, vagy választógombként kezelve őket lekérdezhetjük, melyik az aktív, s csak azon végezzük el a megfelelő műveletet. Ez utóbbit választjuk. (Tanulságos kipróbálni a másik utat is.)

Először töröljük a generált párbeszédablak vezérlőit!

Helyezzük el egymás alatt a gyümölcsök **választógomb**jait (Radio Button)! Nevezzük el őket, és látsszanak a képernyőn nyomógombnak! **Fontos, hogy egymás**

után válasszuk ki őket, mert csak így tudnak egy választógombcsoporthoz tartozni! (Az egymás után választott vezérlőknek az erőforrás-szerkesztő egymás után következő számmal ellátott azonosítót rendel.) Lehet vágólapra másolni és beszúrni őket.

Arról, hogy egymás utáni azonosítót kaptak-e, a View menü Resource Symbols menüpontjában győződhetünk meg.

Properties

IDC_BANAN / Caption: Banán

Styles fül Push-like // Ettől látszanak nyomógombnak.

…

A többi azonosító hasonló logikával készül (IDC_KORTE...), így három pont helyettesíti kiírásukat a későbbiekben is. Ha mind a négy gomb készen van, a vezérlők a Shift vagy a Ctrl gomb lenyomásával egyszerre kijelölhetők. Ilyenkor az utoljára kijelölt lesz az aktuális. Az egérrel téglalapot rajzolva köréjük szintén kijelölhetők, ebben az esetben a rajzolás kezdeténél elhelyezkedő vezérlő az aktuális. A rendezés az aktuális elem megfelelő tulajdonságát veszi alapul. **A Dialog eszközsor segítségével rendezzük őket** azonos méretűre, azonos távolság legyen közöttük, és a bal szélük legyen azonos távolságra a párbeszédablak szélétől!

Ha több vezérlőnek szeretnénk azonos tulajdonságot beállítani, kijelöljük őket, és a tulajdonságlapon egyszerre állíthatjuk be a kívánt elemet.

Ezután vegyük fel a **szerkesztőmező**ket (**Edit box**), a mellettük álló gombnak megfelelő azonosítóval ellátva őket! A szerkesztőmezőket is egymás után vegyük fel, a gombokkal azonos sorrendben, hogy azonosítójuk ugyanúgy következzen, mint a nyomógomboké!

Properties

IDC_BANAN_EDIT / Disabled

Styles: Multiline // Enter-re ne záródjon be az ablak!

Extended Styles: Right aligned text

(IDC_KORTE_EDIT, IDC_ALMA_EDIT, IDC_NARANCS_EDIT)

Rendezzük őket a Dialog eszközsor segítségével! Írjuk melléjük a Ft/kg statikus szöveget!

Megjegyzés:

Ha elhibáztuk, nem következnek egymás után az azonosítók, akkor módosítanunk kell a vezérlő(k) azonosítóját. Először nézzük meg (View menü / Resource Symbols), hogy a kívánt érték nem foglalt-e, mert ha igen, akkor nem adhatjuk értékül más vezérlőnek! Ha nem foglalt az érték, akkor az azonosítót egyenlővé tehetjük az általunk választott értékkel. Properties / ID / ICD_MY_CONTROL = 1006. Figyelem! Használjuk az erőforrás-szerkesztő által is használt számokhoz közeli értéket, nehogy pl. az MFC által már használt azonosító sorszámát adjuk! (További információk az azonosítók értékeiről az MSDN Technical Note 20. fejezetében ID Ranges cím alatt.)

Állítsuk be a Banán és a Banán szerkesztődoboz ablakokra a Properties / General fülnél, hogy **Group**! Ezzel megadtuk a választógombok első elemét (Banán) és a következő csoport első elemét, ezzel a csoport utolsó elemét is.

A kép alján készítsünk még két szerkesztőmezőt

IDC_SULY_EDIT

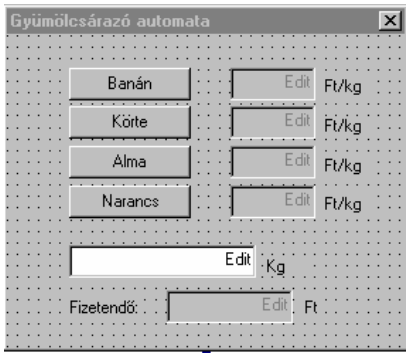
IDC_FIZET_EDIT / Visible kikapcsolva!

Styles: Multiline // Enter-re ne záródjon be az ablak!

Extended Styles: Right aligned text

a hozzájuk tartozó statikus szövegekkel (kg, Fizetendő, Ft). A Fizetendő és Ft feliratoknak adhatunk külön azonosítót: IDC_STATIC_FIZ, IDC_STATIC_FT, ha azt akarjuk, hogy ne lehessen őket látni, amikor a fizetendő ablakot eltüntetjük!

Állítsuk be a párbeszédablak méretét is! (3.Gyakorlat 4. ábra)



3. Gyakorlat 4. ábra A párbeszédablak a szerkesztőben

Nézzük meg a Dialog eszközsor Test gombjával, milyen lenne futás közben az ablak!

➢ Rendeljünk változókat a vezérlőkhöz!

Vagy osztályvarázsló / Member variables fül / ID kiválasztása / Add variable

Itt később is megnézhetjük, mihez mit rendeltünk.

Vagy CTRL + duplakattintás a szerkesztőmezőn.

IDC_BANAN_EDIT	int m_BananAr
...	
IDC_SULY_EDIT	float m_Suly
IDC_FIZET_EDIT	float m_Fizet
IDC_BANAN	int m_Melyik // Választógombok

Nézzük meg, mi változott a kódban! Keressük meg a változó hozzárendelés függvényeit!

Mivel a CEladDlg osztály tartozik a párbeszédfelülethez, ebbe kerültek a változók:

Az EladDlg.h-ban:

```
// Dialog Data
 //{{AFX_DATA(CEladDlg)
 enum { IDD = IDD_ELAD_DIALOG };
 int          m_BananAr;
 int          m_KorteAr;
 int          m_AlmaAr;
 int          m_NarancsAr;
 float        m_Suly;
 float        m_Fizet;
 int          m_Melyik;          // Választógombok (Group!)
 //}}AFX_DATA
```

Az EladDlg.cpp-ben a CEladDlg osztály DoDataExchange metódusában találhatók a vezérlőket a változókkal összekötő DDX metódusok:

```
void CEladDlg::DoDataExchange(CDataExchange* pDX)
{
 CDialog::DoDataExchange(pDX);
 //{{AFX_DATA_MAP(CEladDlg)
 DDX_Radio(pDX, IDC_BANAN, m_Melyik);   // Választógombok
 DDX_Text(pDX, IDC_BANAN_EDIT, m_BananAr);
 …
 DDX_Text(pDX, IDC_SULT_EDIT, m_Suly);
```

```
    DDX_Text(pDX, IDC_FIZET_EDIT, m_Fizet);
    //}}AFX_DATA_MAP
}
```

Az EladDlg.cpp-ben a konstruktorban a kezdő értékadások:

```
CEladDlg::CEladDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CEladDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CEladDlg)
    m_Melyik = -1;            // Választógombok
    m_BananAr = 0;
    m_KorteAr = 0;
    m_AlmaAr = 0;
    m_NarancsAr = 0;
    m_Suly = 0.0f;
    m_Fizet = 0.0f;
    //}}AFX_DATA_INIT
// Note that LoadIcon does not require a subsequent
// DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

Az osztályvarázsló csak nulla kezdőértéket tud adni, de mi azt átírhatjuk. Adjunk kezdőértéket az adatoknak! (Írjuk be az árakat!) Nézzük meg, hogy futás közben megjelennek az új értékek! (DoModal)

➢ Az események kódja

Ha a Banán gombot választjuk, ezt lehessen látni a gomb megjelenésén, és a többi gomb ne legyen vele együtt kiválasztva! Ezt a feltételt teljesíti az a tény, hogy gyümölcs-választógombjaink nem egyszerű nyomógombok, hanem választógombok, csak nyomógombnak látszanak. Ha nyomógombokat tettünk volna az ablakra, az egymást kizáró választást kóddal tudtuk volna megadni.

A Banán gomb választására **legyen fehér a banán ár háttere**, ezzel is jelezve a választást és tájékoztatva a felhasználót az aktuális árról. Jelenjen meg a fizetendő összeg, melyet a súlymezőből és az árból számítunk!

Rendeljünk üzenetkezelő metódust a Banán gomb lenyomása eseményhez!

Osztályvarázsló (Ctrl W)

Class Name: CEladDlg

ID: IDC_BANAN

Messages: BN_CLICKED

Add Function

Member function name: OnBanan OK

Edit Code

```
void CEladDlg::OnBanan()
{
 UpdateData();// Beolvassa m_Melyik értékét és a súlyt.
 GetDlgItem(IDC_BANAN_EDIT)->EnableWindow(TRUE);
 m_Fizet=m_Suly*m_BananAr;
 GetDlgItem(IDC_FIZET_EDIT)->ShowWindow(SW_SHOW);
 GetDlgItem(IDC_STATIC_FIZ)->ShowWindow(SW_SHOW);
 GetDlgItem(IDC_STATIC_FT)->ShowWindow(SW_SHOW);
 UpdateData(FALSE);  // Megjeleníti a kiszámított adatokat.
}
```

Ha a többi gyümölcs gombját választjuk, hasonló szöveget kellene elmondanunk, és értelemszerűen hasonló kódot kellene megírnunk. A hasonló kód írása nem probléma, hisz a kódot egyszerűen átmásolhatjuk a már kész metódusunkból, s csak a konkrét azonosítóknál kell kijavítanunk a szöveget. De ki az, aki nem tudja, hogy milyen kényelmetlen mindezt gombonként tesztelni, s milyen könnyen előfordul, hogy elfelejtjük valamelyik azonosítót átírni? Az ilyen típusú hibák észrevétele elég nehéz. Ezen kívül, ha később módosítunk a kódon (szinte biztos), akkor azt mindig négy helyen kell megtennünk, négy gombra kell tesztelnünk.

➢ Több vezérlő együttes kezelése

Tehát jó lenne együtt kezelni az eseményeket! **Erre lehetőségünk van, ha kézzel írjuk meg a kezelőfüggvényt**. Keressük meg az előbb elkészített OnBanan metódust! A gombokon való kattintást szeretnénk kezelni, így a metódus neve legyen: ***OnButtonClicked(UINT nID).*** Paramétere a gomb azonosítóját tartalmazza, melyen kattintottunk. Mi nem írhatunk az osztályvarázsló számára fenntartott megjegyzések közé, tehát **az //}}AFX_MSG sor mögé írjuk a kódot**:

Az EladDlg.h-ban:

```
// Generated message map functions
//{{AFX_MSG(CEladDlg)
virtual BOOL OnInitDialog();
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnBanan();   // Az osztályvarázsló készítette.
//}}AFX_MSG
afx_msg void OnButtonClicked(UINT nID);
DECLARE_MESSAGE_MAP()
```

Ezt a függvényt fel kell vennünk az üzenettérképbe, és a **BN_CLICKED** üzenethez csatoljuk az IDC_BANAN azonosítótól kezdődően az IDC_NARANCS azonosítóig

a gombokat! (Ezért volt fontos, hogy az azonosítók egymás után következzenek.) A kódot itt is az osztályvarázsló megjegyzései mögé írjuk!

Az EladDlg.cpp-ben:

```
BEGIN_MESSAGE_MAP(CEladDlg, CDialog)
 //{{AFX_MSG_MAP(CEladDlg)
 ON_WM_PAINT()
 ON_WM_QUERYDRAGICON()
 ON_BN_CLICKED(IDC_BANAN, OnBanan)      // Az osztályvarázsló!
 //}}AFX_MSG_MAP
 ON_CONTROL_RANGE(BN_CLICKED, IDC_BANAN, IDC_NARANCS,
OnButtonClicked)
END_MESSAGE_MAP()
```

Meg kell írni a kezelő függvény kódját is!

```
afx_msg void CEladDlg::OnButtonClicked(UINT nID)
{
    // A kódot később írjuk meg! (A következő oldalon.)
}
```

➢ Rendeljük egy tömb elemeit a szerkesztőmezőkhöz!

Amikor a Fizet mezőt számoljuk, az árat a különböző nevű változókból csak egy case elágazással tudnánk kiszedni. Ha készítünk egy tömböt, ami ezeket az adatokat tárolja, könnyebb a hivatkozás. A tömb legyen int m_Ar[4]!

Ha nem akarjuk az adatokat folyton másolgatni a szövegmezőkhöz rendelt változókból a tömbbe, vagy vissza, akkor célszerű lenne a tömb elemeit a vezérlőhöz rendelni. Azonban **az osztályvarázsló nem enged tömbelemet vezérlőhöz rendelni!** A megoldás az, hogy mi írjuk át a kódsorokat. Az osztályvarázsló által generált sorokat **átmásolva** (Nem kitörölve, mi ne nyúljunk az osztályvarázsló kódjához!) az osztályvarázsló megjegyzései közül, a megjegyzések mögé, a megfelelő helyeken átírhatjuk a változók nevét. Hasonló módon a konstruktor kezdőértékadását és a **DDX** függvényeket is módosítanunk kell! Csak mindezek után töröljük ki az osztályvarázsló által generált változókat az osztályvarázslóval! A művelet után a következő kódsorokat kapjuk:

Az EladDlg.h:

```
// Dialog Data
 //{{AFX_DATA(CEladDlg)
 enum { IDD = IDD_ELAD_DIALOG };
 int    m_Melyik;    // Választógombok
 float  m_Suly;
```

```
 float m_Fizet;
 //}}AFX_DATA
 int   m_Ar[4];
```

Az EladDlg.cpp-ben:

```
CEladDlg::CEladDlg(CWnd* pParent /*=NULL*/)
 : CDialog(CEladDlg::IDD, pParent)
{
 //{{AFX_DATA_INIT(CEladDlg)
 m_Melyik = -1;
 m_Suly = 0.0f;
 m_Fizet = 0.0f;
 //}}AFX_DATA_INIT
 // Note that LoadIcon does not require a subsequent
DestroyIcon in Win32
 m_Ar[0] = 150;
 m_Ar[1] = 300;
 m_Ar[2] = 200;
 m_Ar[3] = 250;
 m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}


void CEladDlg::DoDataExchange(CDataExchange* pDX)
{
 CDialog::DoDataExchange(pDX);
 //{{AFX_DATA_MAP(CEladDlg)
 DDX_Radio(pDX, IDC_BANAN, m_Melyik);
 DDX_Text(pDX, IDC_SULT_EDIT, m_Suly);
 DDX_Text(pDX, IDC_FIZET_EDIT, m_Fizet);
 //}}AFX_DATA_MAP
 DDX_Text(pDX, IDC_BANAN_EDIT, m_Ar[0]);
 DDX_Text(pDX, IDC_KORTE_EDIT, m_Ar[1]);
 DDX_Text(pDX, IDC_ALMA_EDIT, m_Ar[2]);
 DDX_Text(pDX, IDC_NARANCS_EDIT, m_Ar[3]);
}
```

Ha tesztelni akarjuk a kódot, a banán kezelő függvényét ki kell törülnünk, mert már nem létező változókra hivatkozik, és egyébként sincs rá szükség! Előbb írjuk meg a közös kezelő függvény kódját, s csak azután töröljük a banánét, és teszteljük a programot!

```
afx_msg void CEladDlg::OnButtonClicked(UINT nID)
{
 UpdateData();// Beállítja m_Melyik értékét és a súlyt
 GetDlgItem(IDC_BANAN_EDIT+m_Melyik)->EnableWindow(TRUE);
 m_Fizet=m_Suly*m_Ar[m_Melyik];
 GetDlgItem(IDC_FIZET_EDIT)->ShowWindow(SW_SHOW);
 GetDlgItem(IDC_STATIC_FIZ)->ShowWindow(SW_SHOW);
 GetDlgItem(IDC_STATIC_FT)->ShowWindow(SW_SHOW);
 UpdateData(FALSE);
}
```

Vegyük észre, hogy itt használjuk ki azt, hogy a szerkesztőmezők azonosítója egymás után következik a választógomboknak megfelelően, így hivatkozhatunk a gomb szerkesztőmező párjára egy egyszerű összeadás segítségével!

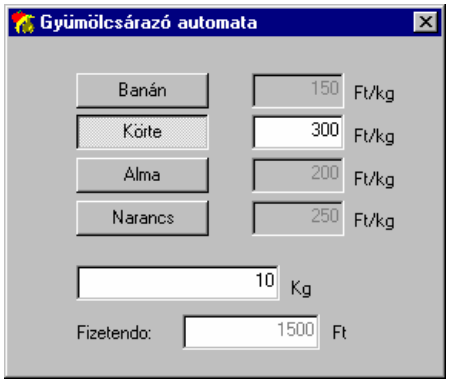
```
GetDlgItem(IDC_BANAN_EDIT+m_Melyik)->EnableWindow(TRUE);
```

Hasonló módon az árakat tartalmazó tömb indexe is lehet a választógomb sorszáma.

Az OnBanan metódus törlése két lépésben történik. Az osztályvarázsló törli a deklarációt és a message map függvényét, de nem törli a függvény implementációt. Ezt nekünk kell kézzel megtennünk. A magyarázat; az implementáció nincs az osztályvarázsló megjegyzései között, és egyébként is gyakran úgy törlünk metódust, hogy a kódját még jó lenne átmásolni egy másik függvénybe, mint ebben az esetben is, ha az OnBanan metódust már az OnButtonClicked létrehozásakor töröltük volna. Ezt teszi lehetővé ez a megoldás. Nem felejtődhetnek kódrészletek a forrásfájlokban, mert a fordító figyelmeztet, ha elfelejtjük törölni a kódot. Ha a kód törlésével kezdjük, a ClassView vagy a Wizard Bar még rá tud állni a függvény elejére a kód törlésének megkezdéséhez, ha viszont először az osztályvarázslóból töröljük ki, utána már nekünk kell a kódot megkeresnünk.

➢ Új gyümölcs választása

A teszt során a következő probléma adódik: Ha pl. 5kg körte után 10 kg almát szeretnénk, a 10 kg beütése után és az alma kiválasztása előtt a 3.Gyakorlat 5. ábra képét látjuk, ami téves



3. Gyakorlat 5. ábra Hibás adatmegjelenítés

Ha írunk a súly bevitel ablakába, tehát új gyümölcs került a mérlegre, az előző fizetendő tűnjön el, az előző gyümölcs árának kiemelése szűnjön meg!

Osztály varázsló (Class Vizard)

ID: IDC_SULY_EDIT

Messages EN_CHANGE

Add Function / OnChangeSulyEdit / OK / Edit Code

```
void CEladDlg::OnChangeSulyEdit()
{
   // Eltüntetik a fizetendő mezőt
   GetDlgItem(IDC_FIZET_EDIT)->ShowWindow(SW_HIDE);
   GetDlgItem(IDC_STATIC_FIZ)->ShowWindow(SW_HIDE);
   GetDlgItem(IDC_STATIC_FT)->ShowWindow(SW_HIDE);
   // Megszüntetik a gyümölcs kiválasztását
   if (m_Melyik!=-1)
   {
     GetDlgItem(IDC_BANAN_EDIT+m_Melyik)->EnableWindow(FALSE);
     ((CButton*)GetDlgItem(IDC_BANAN+m_Melyik))->SetCheck(0);
   }
}
```

Figyeljük meg, mi történik, ha nem tesszük ki az if (m_Melyik !=-1) feltételt! Kezdetben, amíg még nem választottunk egyetlen gombot sem, a konstruktor -1 értéket ad a választógombok változójának (m_Melyik).

Az IDC_BANAN_EDIT-1 épp a narancs nyomógombjának azonosítója, ami az első szám beírásakor tiltott választás lesz emiatt. Tehát programunkból nem választhatjuk többé, vagyis egyáltalán nem választhatjuk a narancsot.

Megjegyzés:

Ha szám típusú adattagot rendelünk a szerkesztődobozhoz, pl. IDC_SULY_EDIT, és amikor üres, meghívjuk rá az UdateData() metódust (OnButtonClicked), egy "Please enter a(n) number (integer)" feliratú figyelmeztető üzenetet kapunk. Az üzenet elkerüléséről az elméleti rész 4. fejezet CString osztállyal foglalkozó bekezdésében lesz szó. (4.9.1. szakasz vége.)

➢ Elhibáztuk a gyümölcsválasztást!

Ha változtatjuk a gyümölcsök nyomógombját anélkül, hogy új súlyadatot vinnénk be, akkor több ár jelenik meg fehér háttérrel.

Az előző gyümölcs aktivizált szerkesztőmezőjét le kell tiltanunk! A OnButtonClicked-ben ezt még az ***UpdateData()*** hívás előtt meg kell tennünk, mert a letiltáshoz tudnunk kell az előző választást, amit az m_Melyik őriz az UpdateData() hívásig!

```
if (m_Melyik!=-1)
  GetDlgItem(IDC_BANAN_EDIT+m_Melyik)->EnableWindow(FALSE);
UpdateData();
```

➢ A Fizetendő összeg ne legyen módosítható!

1. Megtehetjük azt, hogy csak olvashatóra állítjuk a szerkesztőmezőt.

   Properties / Styles / Read-only

   Ekkor a kurzort bevihetjük a mezőbe, kijelölhetjük a számot, csak átírni nem tudjuk.
2. Vagy az osztályvarázsló segítségével

   Az ID_FIZET_EDIT EN_CHANGE üzenetéhez csatolt függvényben:

   (Add Function / Edit Code)

   ```
   UpdateData(FALSE); //Visszaírja a változó értékét a vezérlőbe.
   ```

   Itt is látszólag szerkeszthető a tartalom, de egy kis villanás után visszaíródik az eredeti érték. Előnye viszont a kiemelt háttérszín.
3. A legegyszerűbb megoldás, ha a nem engedélyezést választjuk. Hátránya a szürke háttér. A háttérszín módosításához a rajzoló fejezet ismeretei szükségesek. Információk a CWnd::OnCtlColor tagfüggvénynél.

   GetDlgItem(IDC_FIZET_EDIT)->EnableWindow(FALSE);
4. Választhattunk volna azt a megoldást is, hogy a fizetendő összeget statikus szövegként jelenítettük volna meg, akkor viszont nem tudtunk volna float típusú változót hozzárendelni, csak CStringet (ami nem olyan nagy baj). A vezérlő háttérszíne az előbbiek szerint itt is módosítható, sőt betűszíne is beállítható.

3.1.3. Új párbeszédablakok

➢ A gyümölcs árát ne változtathassuk ilyen egyszerűen!

Most a gyümölcsök ára mező is szerkeszthető. Módosítás esetén jelenjen meg egy üzenetablak, mely visszakérdez, valóban módosítani kívánom-e az aktuális gyümölcs árát, s az csak igen válasz esetén módosuljon!

Előredefiniált (Predefined) párbeszédablakok

Már találkoztunk ilyen ún. **üzenetablak**kal (**message box**) a Hello programnál, nézzük meg, hogyan paraméterezhetőek!

Az osztályvarázsló segítségével az ID_BANAN_EDIT EN_KILLFOCUS üzenetéhez csatolt függvényben: (Add Function / Edit Code)

```
void CEladDlg::OnKillfocusBananEdit()
{
if (MessageBox( "Kívánja módosítani a Banán árát?",
   "Árváltozás",  MB_YESNO+MB_ICONEXCLAMATION)==IDYES)
{
    UpdateData(TRUE);
    OnButtonClicked(IDC_BANAN_EDIT);
}
else UpdateData(FALSE);
}
```

A MessageBox-ra állva üsse le az F1 gombot, válassza a Class Library Reference CWnd::MessageBox témát! Nézze végig a szöveget, majd a Message-Box Styles-t!

Megjegyzés:

Ha kitöröljük a banán árát, és ezután helyezzük át a fókuszt egy másik vezérlőre, és a kérdésre, hogy "Kívánja módosítani a banán árát?" igennel válaszolunk, a következő hibaüzenetet kapjuk: "Please enter an integer!" Ha az egyetlen gombot, az OK-t választjuk, újra a "Kívánja módosítani..." kérdés következik. Mindaddig váltogathatjuk a két ablakot, amíg azt nem választjuk, hogy nem kívánjuk módosítani a banán árát, mert ekkor visszakerül az eredeti ár az ablakba. A magyarázat, hogy az OnKillFocusBananEdit()-ben az UpdateData(TRUE) függvény az üres szerkesztőmezőből próbál egy számot beolvasni. Ez okozza a hibaüzenetet. A megoldás, hogy az üres mezőt a beolvasás előtt fel kell töltenünk 0-val, ami egy valódi számérték. A CString osztály ismertetésekor a 4.9.1. elméleti szakasz végén foglalkozunk a problémával.

Szerkesztett párbeszédablak csatolása

A körte árának módosítása (másképp mint a banáné).

Workspace / ResourceView / Dialog / Jobb egér / Insert / Dialog / New

Properties / IDD_KORTE_DIALOG,

Caption: Körte árváltozás;

Új vezérlők a felületre:

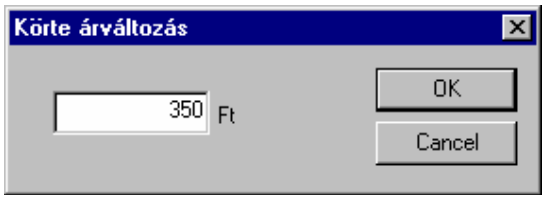
Szerkesztőmező:

IDC_KORTEAR_EDIT

Right alignd text

Multiline

Static text: Ft



3. Gyakorlat 6. ábra Új szerkesztett párbeszédablak

A tabulátor sorrendnél 1. a szövegmező legyen, majd az OK gomb, hogy az ablak megjelenésekor azonnal módosíthassuk a billentyűzetről az árat.

Rendeljünk osztályt az új párbeszédablakhoz! Legyen az új párbeszédablak aktuális, s nyissuk ki az osztályvarázslót!

Üzenet: Ez egy új erőforrás. Rendeljünk új osztályt hozzá! / OK.

Neve: CKorteArDlg, őse a CDialog, erőforrása az IDD_KORTE_DIALOG / OK

Létrehozza a KorteArDlg.h és KorteArDlg.cpp állományokat.

Rendeljünk változót az IDD_KORTEAR_EDIT-hez m_BeKorteAr néven!

Típusa: int. / OK Osztályvarázsló: OK.

Tartalmazzon a CEladDlg osztály egy CKorteArDlg típusú adatmezőt!

Az EladDlg.h-ban :

```
#include „KorteArDlg.h”
...
int m_Ar[4];
CKorteArDlg m_KorteArDlg;
```

Azt akarjuk, hogy ha elkezdjük módosítani a körte árát, jelenjen meg egy párbeszédablak, és abba írjuk a módosítást! Csatoljunk a CEladDlg osztály IDC_KORTE_EDIT azonosítójához tartozó EN_CHANGE üzenethez kódot:

```
void CEladDlg::OnChangeKorteEdit()
{

 m_KorteArDlg.m_BeKorteAr = m_Ar[1];
// m_Ar[1] Az eredeti ár amit módosítani akarunk.

 if (m_KorteArDlg.DoModal() == IDOK) // A módosított árat
elfogadjuk.
 {
    m_Ar[1]=m_KorteArDlg.m_BeKorteAr;
    // Kiírja a képernyőre az új árat
    UpdateData(FALSE);
```

```
      // Megjeleníti az új árral a fizetendőt.
      OnButtonClicked(IDC_KORTE);
   }
   else
      // Megjeleníti a régi értéket.
      UpdateData(FALSE);
}
```

A módosításnál leütött első számjegy elveszett. Újra be kell ütni a Körte árváltozás párbeszédablakban. Ha azt akarjuk, hogy ne dolgozzunk fölöslegesen, be kell olvasnunk a módosítást még a párbeszédablak megjelenítése előtt! (UpdateData()) Így azonban Cancel választása esetén nem tudjuk visszaállítani az eredeti állapotot, mert az az első szám beütésekor felülíródott. Tehát egy segédváltozóban el kell menteni az eredeti árat. A végső kód a következő:

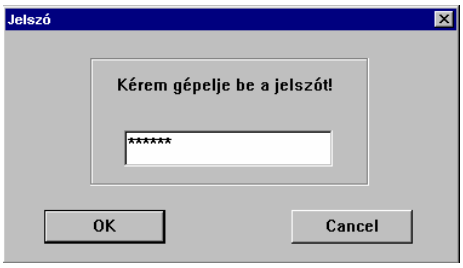
```
void CEladDlg::OnChangeKorteEdit()
{
 int oldAr = m_Ar[1];
 // Az első módosítást is megjelenítjük.
 UpdateData();
 // Feltölti az ármódosítás párbeszédablak adattagját.
 // melyet a DoModal hívás automatikusan megjelenít.
 m_KorteArDlg.m_BeKorteAr = m_Ar[1];
 if (m_KorteArDlg.DoModal() == IDOK)
 {
    m_Ar[1]=m_KorteArDlg.m_BeKorteAr;
    // Kiírja a képernyőre az új árat.
    UpdateData(FALSE);
    // Megjeleníti az új árral a fizetendőt.
    OnButtonClicked(IDC_KORTE);
 }
 else
 {
    m_Ar[1] = oldAr;
    // Megjeleníti a régi értéket.
    UpdateData(FALSE);
 }
}
```

➢ Ha módosítani akarja a körte árát, kérjünk jelszót!

Workspace / ResourceView / Dialog / Jobb egér / Insert / Dialog

Készítsük el a jelszót bekérő ablak felületét! (3.Gyakorlat 7. ábra)

A címsort (Caption: Jelszó), a kurzor a jelszóra várjon! (Tab Order)



3. Gyakorlat 7. ábra Jelszót bekérő ablak

Az IDC_JELSZO_EDIT Properties / Styles fülben jelöljük ki a **Password** jelölőnégyzetet!

Rendeljünk CString típusú változót a **jelszó** szerkesztőmezőhöz (m_beirtJelszo)! Ehhez nyissuk meg az osztályvarázslót!

Megjelenik egy üzenetablak, a JelszoDlg egy új resource, valószínűleg új osztályt akarunk rendelni hozzá. Választhatunk egy meglevő osztályt is.

Create a new class OK

Name: CJelszo OK

Class Wizard / Member Variables fül / IDC_JELSZO_EDIT

Add Variable

Member variable name: m_beirtJelszo (CString)

OK

Hozzunk létre a jelszó tárolására egy CString objektumot a CJelszo osztályban! (Jelszo.h)

```
// Dialog Data
 //{{AFX_DATA(CJelszo)
 enum { IDD = IDD_JELSZO_DIALOG };
 CString       m_beirtJelszo;
 //}}AFX_DATA
 CString m_Jelszo;
```

A konstruktorban adjunk neki kezdőértéket! m_Jelszo = "AAA";

Ha az OK gombra kattint, akkor ellenőrizzük a beírt jelszó helyességét! Ez az OnOK felülírásával történik.

Osztályvarázsló / IDOK / BN_CLICKED

Add Function / OnOK()

```
void CJelszo::OnOK()
{
 // TODO: Add extra validation here
 UpdateData();
 if (m_beirtJelszo==m_Jelszo)
 {
    // Ne maradjon ott a következő jelszó bekéréshez!
    // Ha ott marad a helyes jelszó, annak beütése nélkül,
    // egyszerű OK-ra engedi a módosítást.
    m_beirtJelszo="";
    // Ha nem írom a képernyőre, mivel kilépéskor OK-val lépek
    // ki, az frissíti a képernyőről az m_beirtJelszo-t.
    UpdateData(FALSE);
    CDialog::OnOK();
 }
 else
 {
    MessageBox("Hibás jelszó!!!");
    m_beirtJelszo="";
    CDialog::OnCancel();
 }
}
```

A körte szerkesztőmező EN_CHANGE üzenetére ezt a párbeszédablakot hívjuk, s csak OK visszatérés esetén hívjuk meg az egységár módosítását lehetővé tevő párbeszédablakot.

Ehhez előbb létre kell hoznunk a CEladDlg osztályban egy CJelszoDlg típusú adatmezőt!

Az EladDlg.h ban :

```
#include „Jelszo.h”
...
CKorteArDlg m_KorteArDlg;
CJelszo m_JelszoDlg;
```

Az OnChangeKorteEdit –ben:

```
void CEladDlg::OnChangeKorteEdit()
{
 if (m_JelszoDlg.DoModal() == IDOK)
 {
  int oldAr = m_Ar[1];
  // Az első módosítást is megjelenítjük
  …
  UpdateData(FALSE);
 }
 //Ne maradjon ott a leütött karakter!
 else UpdateData(FALSE);
}
```

Ha a felhasználó előbb kijelöli a körte árát, majd elkezdi beütni a kívánt új árat, az EN_CHANGE üzenet kétszer hívódik meg. Először amikor a kijelölés hatására üres lesz a szövegmező, másodszor amikor leütöttük a módosított ár gombját. Ráadásul az üres mező után a program visszaállítja az eredeti árat. Megoldás:

Üres szövegmező esetén ne történjen semmi, tehát térjen vissza a függvény!

```
void CEladDlg::OnChangeKorteEdit()
{
// Ha kijelöléssel kezdi a módosítást az üres érték után ne
// kérjen jelszót!
 CString beAr;
 GetDlgItem(IDC_KORTE_EDIT)->GetWindowText(beAr);
 if (beAr=="") return;

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

(Ezzel még mindig nem kezeltük azt az esetet, amikor csak az ár egy részét jelöli ki a felhasználó. Ilyenkor továbbra is kétszer kéri a jelszót és a módosítást az alkalmazás. A CString osztályról és műveleteiről a 4. fejezetben olvashatunk.)

Feladat:

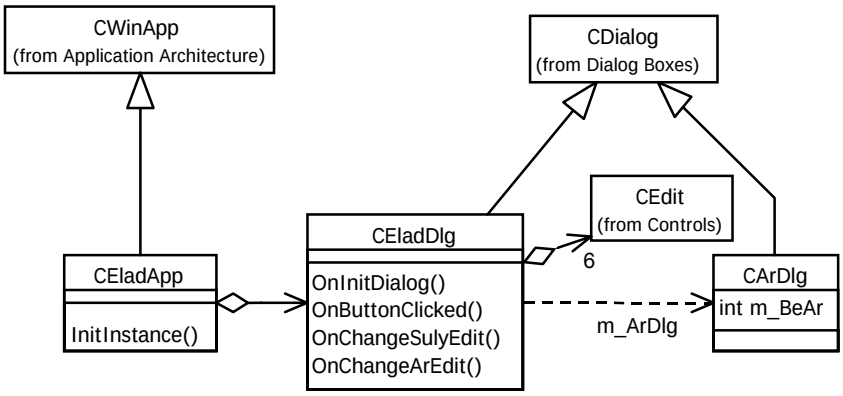
1. Kezelje az összes ármódosítást egységesen a jelszóbekéréses módszerrel! A módosító párbeszédablak tájékoztasson arról is, most épp melyik gyümölcs árát módosítjuk!
2. Az ár módosítását egy külön ármódosító gomb választása tegye lehetővé, természetesen jelszó megadása után! Ezzel a módszerrel elkerülhetjük a jelszó dupla bekérését.
3. Legyen az automatánkon a jelszó módosítását lehetővé tevő gomb!
4. Tárolja a gyümölcsök árait fájlba! A következő indításkor a módosított ár legyen már érvényes! A kódolt jelszavakat is tárolja fájlban!
5. A Spy++ segítségével vizsgálja meg a vezérlők által a párbeszédablaknak küldött üzeneteket! (Elmélet 2.11. ábra.)

3.2. Második megoldás: mindez vezérlőobjektum változókkal

➢ Az osztálydiagram vezérlőobjektumokkal

Ha a vezérlőkhöz vezérlőobjektumokat rendelünk, ezek az objektumok az őket tartalmazó ablakkal együtt létrejönnek, és vele együtt meg is szűnnek, vagyis

tartalmazási kapcsolat van a párbeszédosztály és a vezérlőosztályok között. (3.Gyakorlat 8. ábra)



3. Gyakorlat 8. ábra Az Elad alkalmazás osztálydiagramja vezérlő objektumokkal

➢ Az új projekt generálása

Ha az előző elad projektünk alkönyvtárát átnevezzük Elad1-re, akkor újra Elad néven generálhatjuk a párbeszédalapú alkalmazásunkat.

Másoljuk át az előző Elad program Elad.rc és Resource.h fájljait az új alkalmazásba! (Írjuk felül a meglevő fájlokat!) Más feladatnál, ha saját erőforrásokkal is rendelkezik az alkalmazás (pl. saját kurzor vagy ikon), akkor a res alkönyvtárat is át kell másolni!

➢ Változók hozzárendelése a vezérlőkhöz

Rendeljünk vezérlő objektumokat a szövegmezőkhöz!

Osztályvarázsló / Member Variables fül / ID: IDC_BANAN_EDIT

Add Variable

Adattag neve: m_Banan, ..., m_Suly, m_Fizet

Kategóriája Control

Típusa: CEdit. //Automatikusan

➢ Kezdeti beállítások

Az inicializálás során be kell állítanunk a megjelenített egységárakat. Ezt a főablakosztály (CEladDlg) **WM_INITDIALOG** üzenetét kezelő ***OnInitDialog*** metódusában tehetjük meg. Az alkalmazásvarázsló már elkészítette számunkra az üzenetkezelőt, melynek kódját ki fogjuk egészíteni. A metódust a ClassView-ban duplakattintással a nevén (vagy a Wizard Bar CEladDlg osztály, majd a metódus nevének kiválasztásával, vagy az osztályvarázslóból Massage Maps fül, az osztály és függvénynév választása után az Edit Code gomb segítségével érhetjük el). A "`// TODO: Add extra initialization here`" szöveg mögé célszerű írnunk.

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

 // Set the icon for this dialog.  The framework does this
automatically
 //  when the application's main window is not a dialog
 SetIcon(m_hIcon, TRUE);                // Set big icon
 SetIcon(m_hIcon, FALSE);         // Set small icon

 // TODO: Add extra initialization here
 m_Banan.SetWindowText("150");
 m_Korte.SetWindowText("300");
 m_Alma.SetWindowText("200");
 m_Narancs.SetWindowText("250");
 m_Suly.SetWindowText("0");
return TRUE; //return TRUE unless you set the focus to a control
}
```

A ***CWnd::SetWindowText*** metódusa szöveges adatot vár, így árainkat szövegként kell megadnunk.

➢ Események kezelése

Súlymódosítás

Gyümölcsválasztás

Egységár módosítás

Az utóbbi kettő eseménycsoport kódja az OnChangeSulyEdit tagfüggvény meghívását tartalmazza.

Súlymódosítás:

Osztályvarázsló (CEladDlg)

IDC_SULY_EDIT

EN_CHANGE

Add Function

OnChangeSulyEdit

Edit Code

```
void CEladDlg::OnChangeSulyEdit()
{
 CString Suly;
 CString ForintPerKilo;
 m_Suly.GetWindowText(Suly);
 int nSuly = atoi(Suly);
 int Melyik = GetCheckedRadioButton(IDC_BANAN, IDC_NARANCS);
 switch (Melyik)
 {
  case IDC_BANAN:m_Banan.GetWindowText(ForintPerKilo);break;
  case IDC_KORTE:m_Korte.GetWindowText(ForintPerKilo);break;
  case IDC_ALMA: m_Alma.GetWindowText(ForintPerKilo);break;
  case IDC_NARANCS: m_Narancs.GetWindowText(ForintPerKilo);
                                                    break;
 }
 int nForintPerKilo = atoi(ForintPerKilo);
 int nForint = (nForintPerKilo * nSuly);
 char Forint[10];
 itoa(nForint, Forint, 10);
 CString TeljesAr(Forint);
 m_Fizet.SetWindowText(TeljesAr);
}
```

A CWnd osztály ***int GetCheckedRadioButton(int*** *nIDFirstButton****, int*** *nIDLastButton* ***);*** metódusa választógomb esetén visszaadja a választott gomb azonosítóját.

Az atoi, itoa függvények alakítják át a szöveget egésszé, illetve az egész számot szöveggé.

Megjegyzés:

Ha átírjuk a súly szövegmező tartalmát, az kétszer hívja meg az EN_CHANGE üzenetet, egyszer amikor kitöröljük "" tartalommal és egyszer a beíráskor "szám" tartalommal.

Gyümölcsválasztás:

Be kell állítanunk, hogy a megfelelő választógomb legyen kiválasztva! Most tegyük meg ezt egyenként! A beállítás után csak meg kell hívnunk a súly módosítását kezelő metódust.

Osztályvarázsló (CEladDlg)

IDC_BANAN

BN_CLICKED

Add Function

OnBanan

Edit Code

```
void CEladDlg::OnBanan()
{
 CheckRadioButton(IDC_BANAN, IDC_NARANCS, IDC_BANAN);
 OnChangeSulyEdit();
}
```

A CWnd osztály ***void CheckRadioButton(int*** *nIDFirstButton****, int*** *nIDLastButton****, int*** *nIDCheckButton* **);** tagfüggvénye az első paraméterben megadott gombtól a második paraméterben megadott gombig kiszedi a csoportból a kijelölést, és a harmadik paraméterben megadott gombot jelöli ki. Itt is feltételezzük, hogy a gombok azonosítói a banántól a narancsig egymás után következő számokat jelentenek.

A többi gombra is megtehetjük, a kód csak a CheckRadioButton 3. paraméterében változik.

Az egységár módosítása legyen az olvasó feladata.

Azt láthatjuk az eddigiekből, hogy a feladat megoldása vezérlő objektum változók használata esetén egészen más problémákat vet fel. Ne feledjük, hogy párhuzamosan is használhatjuk a változókat, tehát egy szövegmezőhöz rendelhetünk egy CEdit és egy int típusú adattagot is! A programozó dönti el a konkrét feladatban, hogy mikor melyik megoldást választja. A nyelv a lehetőségeket biztosítja.

3.3. Feladatok:

3.3.1. Köszöntés hanggal *

Tegyünk hangot a Hello alkalmazás köszöntés gomb párbeszédablaka alá! A Köszöntés gomb választása után ne csak az üzenetablak jelenjen meg, hanem hallgassuk is meg a köszöntést! (Ha módunkban áll fölvenni a szöveget, tegyük meg, ha nem, csatoljunk egy egyszerű zenét!) Próbáljuk ki a különbséget, hogy szinkron és aszinkron hang esetén be tudjuk-e zárni az ablakot a hangfájl vége előtt!

3.3.2. Számológép

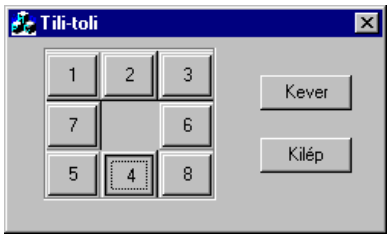
Készítsünk a négy alapműveletet elvégezni tudó számológép alkalmazást! A számokat jelző gombokat a 3.1. gyakorlaton ismertetett módon közösen kezelhetjük.

3.3.3. Fényképtár *

Készítsünk alkalmazást, melyben egy legördülő listából választva megjelenik a kiválasztott személy képe! Ha nincsenek készenlétben képek, egyszerűsítsük a feladatot! A listából betűket válasszunk, s az általunk elkészített képen is betűk legyenek!

3.3.4. Tili-toli játék *

Egy 3X3-as négyzetben 1-8-ig számok helyezkednek el, egy "kocka" üres. A játékos az üressel szomszédos mezők valamelyikére kattintva, az ott elhelyezkedő számot az üres helyre cseréli. Ilyen módon a számok sorrendje megváltoztatható. Cél a számok sorba rendezése. (Ötlet: Baga, 1998, 88.old.)



3. Gyakorlat 9. ábra Tili-toli játék

3.3.5. Telefon

Az alkalmazás egy jelszóbekérő ablak megnyitásával indul. Az egyes felhasználók különböző szintű jogosultságokkal rendelkeznek. Akinek a felhasználói neve eggyel kezdődik, kezdeményezhet nemzetközi távhívást, a kettessel kezdődőek hívhatnak mobiltelefont vagy belföldi távolsági hívást (06), a hármassal kezdődőek helyi hívásra jogosultak, a négyes csak a házon belüli melléket hívhatja.

A készülék a hívott szám beütése után jelezze vissza a hívott számot, és adjon telefonhangot abban az esetben, ha a felhasználónak joga van hívni az adott számot! Egyébként küldjön hibaüzenetet!

3.3.6. Nyomógomb képpel *

Készítsünk a párbeszédablakunkra két nyomógombot! Az egyik színes háttéren az U (Up), D (Down), F (Focused), X (Disabled) betűket mutassa, a másikon kép található! Kezeljük a **duplakattintás eseményt** is!

3. Gyakorlat 10. ábra Gomb képpel

3.3.7. Szókirakó játék

Készítsen egy programot, mely egy bizonyos szót keres betűnként!

A játékot maximum négy játékos játszhatja, akik az elején megadják a nevüket. A képernyőn a játék során annyi játékos név szerepeljen, amennyi nevet megadtak!

Az aktuális játékos leüt egy betűt, ami ha szerepel a szóban, megjelenik a saját helyén a szó ablakában. Ha a szóban ez a betű nem szerepel, a következő játékosé a lehetőség. Aki az egész szót eltalálta, az a győztes.

A választott betű megadását kísérje hangjelzés attól függően, hogy talált-e a betű!

3.3.8. Memória játék (memory)

A képernyőn arccal lefelé fordított kártyák találhatók, melyek nem látható előlapja kettesével megegyezik. A játékos rákattint az egérrel kettőre, melyek megfordulnak, s ha egyformák, akkor "levesszük őket az asztalról". A találatot elérő játékos pontszáma növelése után folytathatja a játékot addig, míg össze nem illő párt választ. Ekkor a következő játékos próbálkozhat.

A játékot lehet más, pl. angol-magyar szópárokkal is játszani.

3.4. Ötletek a feladatok megoldásához:

A sorszámok itt a feladat szövegének sorszámát jelzik.

3.2.1. Köszöntés hanggal

Tegyünk hangot a Hello alkalmazás köszöntés gomb párbeszédablaka alá!

Beállítások az elmélet 2.4. szakaszában.

```
void CHelloDlg::OnKoszonButton()
{
 sndPlaySound("C:\\…\\hello.wav",SND_SYNC);
 MessageBox("Hello! ");
 sndPlaySound(NULL,SND_ASYNC);    // Ne feledjük!!!
}
```

3.3.3. Fényképtár

Egyszerűsített változat, nem szükségesek hozzá képek. Készítsünk alkalmazást, melyben egy legördülő listából betűket választva megjelenik a kiválasztott betű képe!

Készítsünk üres párbeszédalapú alkalmazást!

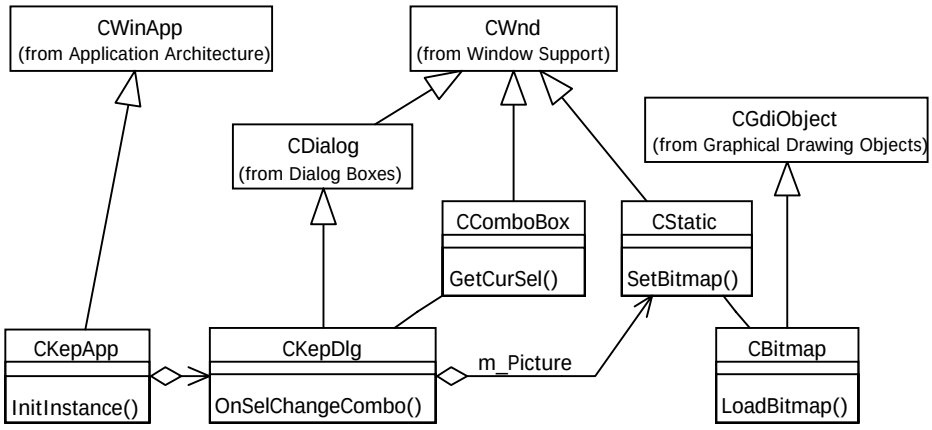
Vegyünk föl egymás után **néhány új Bitmap erőforrást**! A Graphics eszköztárban az A betűt választva a megnyíló szövegszerkesztő ablakba gépeljük be a kívánt betűt! Majd a téglalap kiválasztó ikon segítségével kijelölve a betűt a bitmap-en a kívánt helyre mozgathatjuk, a kívánt méretre nagyíthatjuk. Figyeljünk rá, hogy **az első Bitmap** (IDB_BITMAP1) **maradjon üres, a többibe egyenként vigyük föl az ABC betűit!**

Tegyünk a párbeszédablakunkra egy Combo Box és egy Picture vezérlőt! A Combo Box tulajdonságablakának Data fülében vegyük fel egymás alá a betűket! (Ctrl Enter a sorváltás**.) A Picture vezérlőnek adjunk saját azonosítót (IDC_PICTURE)**, típusa (type) legyen Bitmap, és az Image legördülő listájából válasszuk ki az üres képet! (Más megoldás, ha a Visible jelölőnégyzetből vesszük ki a pipát. Ekkor nem kell üres Bitmap, és figyelni kell a kép láthatóvá tételére is.)

Rendeljünk a Picture vezérlőhöz egy CStatic m_Picture vezérlő típusú változót! Kezeljük a választást a legördülő listában!

```
void CKepDlg::OnSelchangeCombo()
{
 int ind=((CComboBox*)GetDlgItem(IDC_COMBO))->GetCurSel();
             // Visszaadja a kiválasztott cella sorszámát.
 CBitmap bitmap;
 bitmap.LoadBitmap(IDB_BITMAP1+ind+1);
 m_Picture.SetBitmap(bitmap);
 m_Picture.RedrawWindow(); // NT-nél nem kell!
}                          // A vezérlő képét frissíti.
```

A program működésének feltétele, hogy a Bitmap-ek azonosítói egymás után következő számok legyenek.



3. Gyakorlat 11. ábra A Fényképtár osztályai

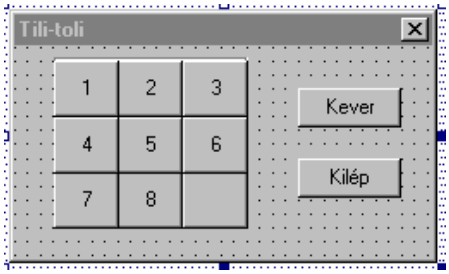
Ha a képeket fájlokból szeretnénk feltölteni, akkor nem lesz szükségünk a Bitmap erőforrásokra, és a CBitmap objektumra sem, viszont meg kell oldanunk a fájlnevek módosítását. A LoadImage API függvény betölti nekünk a képeket a fájlból:

```
CString forras;
forras.Format("res\\bitmap%d.bmp", ind+1);
HANDLE h = LoadImage(NULL,forras, IMAGE_BITMAP, 200, 200,
                                          LR_LOADFROMFILE);
m_Picture.SetBitmap(HBITMAP(h));
```

3.3.4. Tili-toli játék

A játék célja, az összekevert gombok sorba rakása. A játék összerakásához arra a gombra kell kattintani az egérrel, ami szomszédos az üres hellyel, és át akarjuk tolni az üres helyére.

Szerkesszük meg a felületet! A gombok közül elég, ha egyet állítunk be, a többi másolható, egy sor elkészítése után a másik két sor is másolható. A gombokat nevezzük el a számuknak megfelelően IDC_BUTTONX-nek, a 9-es gomb Visible jelölőnégyzete ne legyen kijelölve! Ahhoz, hogy a nem látható gomb helye látható legyen, vegyük körül egy szöveg nélküli csoportosító vezérlővel a gombokat!



3. Gyakorlat 12. ábra A tili-toli játék a szerkesztőben

Tároljuk az üres helyet A CTiliDlg osztályban egy int m_Ures[2] tömbben, mely a konstruktorban kapja meg a kezdőértéket (jobb alsó sarok). m_Ures[0]=2; m_Ures[1]=2;

Mivel a gombok azonosítói egymás után követik egymást, megoldható, hogy egy kezelőt rendeljünk hozzájuk, a gyümölcsautomata feladathoz hasonlóan. (OnButtonClicked)

```
afx_msg void CTiliDlg::OnButtonClicked(UINT nID)
{
 //Az azonosítóból kiszámolja a gomb helyét
 int sor=(nID-IDC_BUTTON1)/3;
 int oszlop=(nID-IDC_BUTTON1)%3;
 // Ha szomszédosak, csere!
 if (abs(sor-m_Ures[0])+abs(oszlop-m_Ures[1])==1)
 {
  CWnd* gomb=GetDlgItem(nID);
  CWnd* ures=GetDlgItem(IDC_BUTTON1+3*m_Ures[0]+m_Ures[1]);
  // A feliratok cseréje
  CString szam;
  gomb->GetWindowText(szam);
  ures->SetWindowText(szam);
  gomb->SetWindowText("");
  // A láthatóságok cseréje
  ures->ShowWindow(SW_SHOW);
  gomb->ShowWindow(SW_HIDE);
  // Az új üres sor beállítása
  m_Ures[0]=sor;
  m_Ures[1]=oszlop;
 }
}
```

A játék a keverés kivételével már működik is. A keverés nem jelent mást, mint véletlen szám alkalommal meghívni a cserét.

```
void CTiliDlg::OnKeverButton()
{
 // A véletlengenerátor kezdőértéke
 srand( (unsigned)time( NULL ) );
 int hanyszor=rand()/100;
 for (int i=1;i<hanyszor;i++)
 {
    // A 9 gombból véletlenül választ egyet
        int mire=rand()*9/RAND_MAX;
    OnButtonClicked(IDC_BUTTON1+mire);
 }
}
```

Természetesen az OnInitDialog()-ban meg kell hívnunk a keverést.

3.3.6. Nyomógomb képpel

Készítsünk nyomógombot, melyen kép található!

A megoldáshoz használjunk **CBitmapButton** objektumot!

A párbeszédalapú alkalmazásra helyezzünk el két gombot! A CBitmapButton használatához a stílusának **owner draw**-nak kell lennie! A nyomógomb tulajdonságlapján (Property page) a Styles fülben válasszuk ki az Owner draw jelölőnégyzetet! (**Ez az a tulajdonság mely esetén a duplakattintás esemény kezelhető.**) Az **Extended Styles** fülben legyen a **Modal frame** jelölőnégyzet kiválasztva, ha azt akarjuk, hogy gombunk kiemelkedjen a felületből!

Készítsük el a Bitmap erőforrásokat! Szúrjunk be az alkalmazás projektjébe négy új .bmp-t! (ResourceView / resources, jobb egér / Insert / Bitmap / New) Készítsük el a képeket az erőforrás-szerkesztővel! Hátterük legyen színes és szerepeljen rajtuk a négy betű! **A CBitmapButton-nak maximum 4 állapota lehet: U (up), D (down), F (focused), X (disabled).** (További információk a Help-ben a CBitmapButton leírásában.) Ennek megfelelően az azonosítója a képeknek legyen a funkciónak megfelelő betű, tehát IMAGEU(, IMAGED, IMAGEF, IMAGEX).

3. Gyakorlat 13. ábra A képgombok a szerkesztőben

Rendeljünk a Dlg osztályunkban egy CBitmapButton típusú m_BitButton nevű adattagot az első gombhoz! Mivel az osztályvarázsló nem kínálja föl a Control

kategóriában a CBitmapButtont, ezért rendeljünk hozzá CButton-t, majd az osztályban a deklarációt írjuk át CBitmapButton-ra!

Owner draw stílus esetén meg kell adnunk a képeket is, amit a gomb helyén akarunk megjeleníteni, még az ablak kirajzolása előtt. A Dlg osztály InitInstance tagfüggvényében hívjuk meg az m_BitButton adattagra a ***LoadBitmaps*** tagfüggvényt!

```
m_BitButton.LoadBitmaps(IMAGEU,IMAGED,IMAGEF,IMAGEX);
```

Próbáljuk ki, ha a gombot lenyomva tartjuk, megjelenik az U, rákattintva pedig az F betű, hisz fókuszban van. Az ablakra (a gombon kívül) kattintva eltűnik az F. A másik gomb nem látszik, mert nem adtunk meg képet a megjelenítéshez, s ez owner draw esetben kötelező.

Vegyünk fel új képeket a másik gombhoz, melynek azonosítóját nevezzük m_KepButton-nak! Végezzük el a hozzárendelést az InitInstance-ban! (Választhatjuk a meglevő képeket is.)

Rendeljük az első gomb BN_CLICKED üzenetéhez a másik gomb tiltását / engedélyesését! A duplakattintáshoz **BN_DOUBLECLICKED** – egy üzenetablakot, hogy duplán kattintottunk.

```
void CKepGombDlg::OnButton1()
{
 m_KepButton.EnableWindow(!m_KepButton.IsWindowEnabled());
}

void CKepGombDlg::OnDoubleclickedButton1()
{
 MessageBox("Dupla kattintás");
}
```

Az ***IsWindowEnabled()*** tagfüggvény visszaadja, hogy az ablak, amire hívtuk, engedélyezett-e. A második gomb duplakattintásával az első gomb tiltását / engedélyezését szabályozhatjuk.

```
void CKepGombDlg::OnDoubleclickedButton2()
{
 m_BitButton.EnableWindow(!m_BitButton.IsWindowEnabled());
}
```

Figyeljük meg, hogy **a duplakattintás először meghívja a kattintást, s csak azután hajtódik végre!**

Megjegyzés:

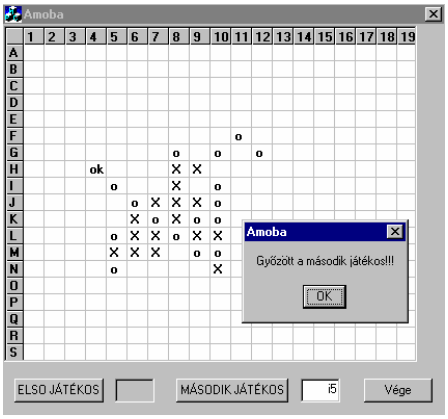
Az AutoLoad függvény segítségével is feltölthetjük a BitmapButton képeit. Lásd bővebben a súgóban!

Az owner draw stílus azt jelenti, magunknak kell megrajzolni a gomb képét. Ezt nem csak bitmap magjelenítésével, hanem valóban a gomb kirajzolásával is megtehetjük. Erről bővebben a 5. fejezet feladatai közt lesz szó.

4. Gyakorlat MFC osztályok

4.1. Amőba játék

A játékosok felváltva választhatnak kockát, az a nyertes, aki egymás mellé (fölé, vagy keresztben) egy vonalban 5 egyforma jelet tud tenni. Győzelem esetén jelenjen meg a 4. Gyakorlat 1. ábrán látható ablak!

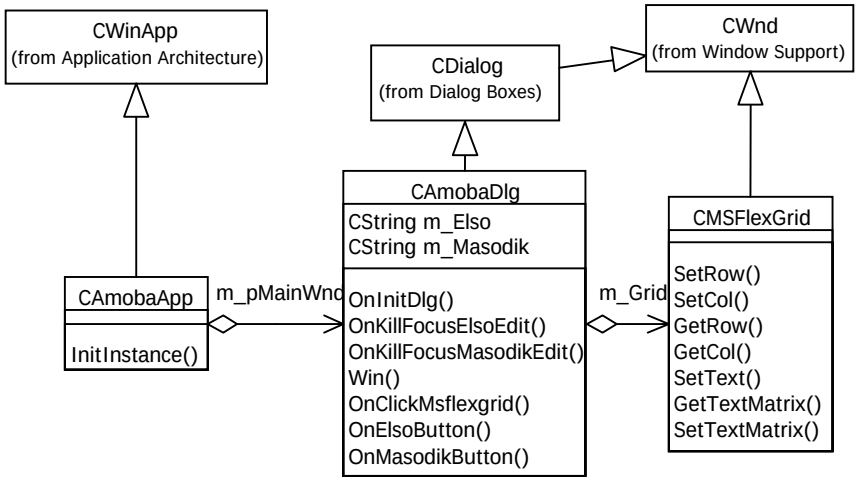


4. Gyakorlat 1. ábra Amőba játék

Mindkét játékos játszhatja billentyűzetről és egér segítségével is!

4.1.1. Terv

A játékot párbeszédalapú alkalmazással valósítjuk meg. A párbeszédfelületen, a nyomógombokon és a szerkesztőmezőkön kívül szükségünk lesz egy táblázatvezérlőre is. A győzelmet egy üzenetablakkal íratjuk ki.



4. Gyakorlat 2. ábra Az amőba osztálydiagramja

4.1.2. Előkészítő lépések

Ebben és a következő bekezdésben leírt tevékenységek elvégzésére nincs szükség, mert telepítéskor a telepítőlemez elvégezte helyettünk, de érdemes e lépéseket követnünk, mert amennyiben a telepítőlemezen nem szereplő ActiveX vezérlővel szeretnénk dolgozni, munkánk feltétele e lépések elvégzése.

Nézzük meg, hogy a Windows \ System32 könyvtárban megtalálható-e, az általunk csatolni kívánt Msflxgrd.ocx! (Ez egy 32 bites ActiveX control.)

Ha nincs ott, telepítsük a telepítőlemezről!

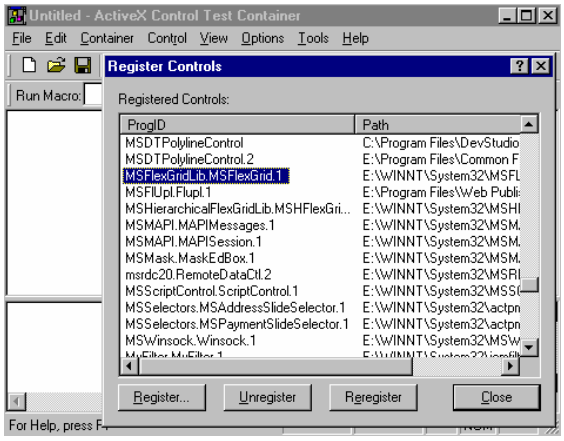
- **Vegyük nyilvántartásba a file-t!**

Ellenőrizzük, regisztrálta-e a rendszer!

Válasszuk a Tools menü Register Control vagy ActiveX Control Test Container menüpontját. (Az eszköz önállóan is használható a Start menü / Programs / Visual Studio / Microsoft Visual Studio Tools / ActiveX Control Test Container címszó alatt.

File menü **Register Controls** menüpont.

A listában most megtalálható a MsFlexGridLib.MSFlexGrid1.



4. Gyakorlat 3. ábra Regisztrálás

Ha nem szerepel: Regiszter nyomógomb.

A file neve : …System32 \ Msflxgrd.ocx.

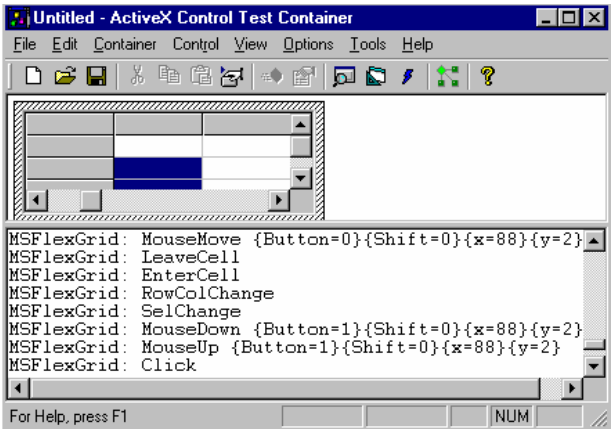
Open

A regisztráció megtörténtét a **Controls Registry** ablakban ellenőrizhetjük.

Close

Tesztelhetjük is a vezérlőnket.

Edit menü / Insert new Control / A listából választ.



4. Gyakorlat 4. ábra Az ActiveX Control Test Container

4.1.3. Kódolás

- **A projekt fájljainak generálása**

MFC AppWizard. exe **név: Amoba**

Dialog Based

A 2. lépésben **ActiveX controls** jelölőnégyzet (check box) bekapcsolás ellenőrzése (default).

A többi default.

- **Az Amőba program vizuális felülete**

Resources / Dialog / IDD_AMOBA_DIALOG

Jobb egérrel kattintsunk a párbeszédablak szabad felületére!

Insert ActiveX Control

Microsoft FlexGrid Control

OK

Készítsük el a felületet a szokott módon!

A Grid ablak ugyanúgy szerkeszthető, mint az eddigi vezérlők. Állítható a mérete, helye. Beállíthatók a jellemzők a properties menüpont segítségével.

Properties

General / ID: IDC_MSFLEXGRID

Control / Sorok, Oszlopok száma 21, ScrollBars / None

Font / Bold

Az oszlopok szélességét majd a programból állíthatjuk be.

- **Osztályok létrehozása**

Osztályvarázsló

Member Variables

Osztály neve (Class name): CAmobaDlg

Object IDs: IDC_MSFLEXGRID

Változó hozzáadása (Add Variable) gomb.

A Grid vezérlő nincs beillesztve a project-be… / OK.

Az osztályok érvényesítése (Confirm Classes) OK

A Visual C++ létre fogja hozni a CMSFlexGrid (...) osztályt, a msflexgrid.h és msflexgrid.cpp állományokban.

Ez után csatolhatjuk a változókat a vezérlőkhöz. A CMSFlexGrid típusú m_Grid adattagot a táblázathoz. A két szerkesztődobozhoz a CString típusú m_Elso, m_Masodik adattagokat. Méretük legyen maximálisan 3 karakter (osztályvarázsló).

➢ Kezdőérték beállítások

Az osztályvarázslóban a CAmobaDlg azonosítóhoz tartozó WM_INITDIALOG üzenethez csatolt OnInitDialog függvény kódját kell módosítanunk az Edit Code nyomógomb választásával. A sorok és oszlopok száma eggyel nagyobb a szükségesnél, ez a széleken levő elemek vizsgálatát fogja megkönnyíteni, nem kell vizsgálni, hogy mikor hivatkozunk határon kívüli elemekre.

A következő kódrészleteknél a kód megírásán kívül az is cél volt, hogy használjuk a CString osztály néhány metódusát.

```
BOOL CAmobaDlg::OnInitDialog()
{
CDialog::OnInitDialog();
...
// TODO: Add extra initialization here
short i;
 CString SorNev("ABCDEFGHIJKLMNOPQRS"); // A fejléc betűi
 m_Grid.SetRow(0);          //Oszlopfejléc feltöltése
m_Grid.SetColWidth(0,250);
//Az első oszlop négyzet alakú cellákkal
m_Grid.SetFixedAlignment(0,2);
//Az írás középre pozicionálása.
char oszlop[2];
for (i=1;i<20;i++)                //Csak 19 cellában játszunk!
 {
// 250-re lenne négyzet alakú, de a számok nem férnének el.
m_Grid.SetColWidth(i,300);
m_Grid.SetFixedAlignment(i,2);
//Az írás középre pozicionálása.
    m_Grid.SetCol(i);             //Oszlopfejléc feltöltése
    itoa(i,oszlop,10);
    m_Grid.SetText(oszlop);
 }
m_Grid.SetCol(0);                 //Sorfejléc feltöltése.
for (i=1;i<20;i++)
 {
m_Grid.SetRow(i);
m_Grid.SetText(CString(SorNev[i-1]));
 }
 m_Grid.SetColWidth(20,1);        //Szélső oszlop
 m_Grid.SetRowHeight(20,1);       //Szélső sor
```

```
m_Grid.MoveWindow(0,0,400,325);
//Az ablakméret hozzáigazítása
//End my Code
return TRUE;  // return unless you set the focus to a control
}
```

- **Beviteli dobozba írás esemény**

Akkor vizsgáljuk a bevitt szöveget, ha már végeztünk a beírással. Ezt a másik játékosnak átadva a játék jogát (lenyomva a gombját) jelezhetjük.

Osztályvarázsló / IDC_ELSO_EDIT / EN_KILLFOCUS

Add Function / OnKillfocusElsoEdit / Edit Code

```
void CAmobaDlg::OnKillfocusElsoEdit()
{
 UpdateData(TRUE);          // Beolvassa a cella nevét.
    //Ha egérre vált ne dolgozzon végig, és ne legyen hiba!
 if (m_Elso.IsEmpty()) return;
 CString SorNev("ABCDEFGHIJKLMNOPQRS"); CString Sor;
 //Csak ezeket a betűket fogadjuk el!
 CString OszlopNev("0123456789"); CString Oszlop;
 Sor=m_Elso.Left(1);        //Az első karakter a sor betűje.
 Sor.MakeUpper();
 Sor=Sor.SpanIncluding(SorNev);
 //Ha nincs a megadott betűk között, üres lesz.
 Oszlop=m_Elso.Mid(1,2);
 //Az 1. sorszámú, tehát a második karaktertől kezdődően vesz,
 //ha van kettőt.
 Oszlop=Oszlop.SpanIncluding(OszlopNev);
 //Csak számjegyeket fogad el.
 int nOszlop = atoi(Oszlop);             // String->int
 if (Sor.IsEmpty() || Oszlop.IsEmpty())
 {
         m_Elso="";  UpdateData(FALSE);
 //Ha bármelyik üres, nem hivatkozik rá, törli az adatot.
 }
 else
 {
  m_Grid.SetRow(SorNev.Find(Sor)+1);
  m_Grid.SetCol(nOszlop);
  m_Grid.SetText("X");
  if (Win(m_Grid.GetRow(),m_Grid.GetCol(),"X"))
  MessageBox("Győzött az első játékos!!!");
 }
}
```

➢ A győzelem függvény

Az OnKillFocus egyelőre nem vizsgálja a határokat és a bevitel helyességét (számjegy, megfelelő betű), emiatt tesztelni csak helyes értékekkel érdemes.

A győzelem függvény a CAmobaDlg osztály tagfüggvénye, így annak header file-jában deklarálni kell, majd: a győzelem függvény megvizsgálja az összes szomszédos elemet, hogy megegyezik-e a Mark paraméterben megadottal. Ha talál egyezőt, elmegy abba az irányba (x és y koordinátákkal megadva) a sor végéig, majd visszafelé számol. Ha 5 vagy annál több egyforma van egymás mellett, akkor a talált sorozat végére írja, hogy „ok”, és ezután igaz, egyébként hamis értékkel tér vissza.

BOOL CAmobaDlg::Win(long RowC, long ColC, const CString Mark);

```
BOOL CAmobaDlg::Win(long RowC, long ColC, const CString Mark)
{
short Row=short (RowC); short Col= short (ColC);
short Count=1; short x=1; short y=1;
while (y!=-1)
  {
while (m_Grid.GetTextMatrix(Row+y,Col+x)==Mark)
      { //Ha megegyezőt talált, ugyanabba az irányba lép tovább.
          Row+=y;Col+=x;
      } //Elmegy a végéig, majd visszafelé számol,
        //mert lehet, hogy a közepéről indult.
      while (m_Grid.GetTextMatrix(Row-y,Col-x)==Mark)
      {
          Row-=y;Col-=x;
          Count++;
      }
      if (Count>=5)
      { //5 vagy több elemnél a végére írja: ok.
          m_Grid.SetTextMatrix(Row-y,Col-x,”ok”);
          return TRUE;
      }
      else                  // Nincs meg az 5, irányt vált.
      {
          if (x>-1) x-=1;   // Új x irány
          else y-=1;        // Új y irány
          // Kiindulási ponból indul.
          Row=short (RowC); Col= short (ColC);
          Count=1;
      }
  }
return Count==5;
}
```

Ha látni akarjuk az elemeket, ahogy keresi és megtalálja, használjuk a GetTextMatrix metódus helyett a SetRow, SetCol, és GetText metódusokat! Sleep(néhányszáz) függvénnyel lassíthatjuk a lépéseket.

Elegendő a szomszédos elemeket csak az egyik oldalról ellenőrizni, mert a visszafelé menetben az ellenkező irányt úgyis végignézzük.

Mivel a program még nem vizsgálja a határokat, tesztelés közben csak középső pontokat adjunk meg!

➢ A tesztelés során felmerülő problémák megoldása

Győzelem esetén legyen vége:

```
if (Win(…
{
MessageBox( "Győzött az első játékos!!!" );
OnOK( );
}
```

A szélekre ne hivatkozzunk!!

A táblázat mérete eggyel nagyobb a szükségesnél (21). Ezt már eleve így állítottuk be.

Ne engedjünk a határon kívüli értéket bevinni! Az OnKillFocus-ban:

```
if (Sor.IsEmpty()||Oszlop.IsEmpty() || nOszlop<1 || nOszlop>19)
{
m_Elso="";  UpdateData(FALSE);
 }
```

Írjuk meg a másik játékos kódját is! Figyelem! A második játékos nem tehet 'O' betűt a képernyőre, mert ekkor az N / O / P sor 1. cella választása esetén hibásan működik, mivel a fejléc O karakterét is figyelembe veszi. Válasszuk pl. a kis 'o' betűt!

A játékosok felváltva játsszanak!

Amíg a másik játékos játszik, legyen letiltva a bevitel, az aktív játékos beviteli ablaka legyen csak engedélyezett! Kezdetben az első játékos beviteli ablaka legyen csak engedélyezett! Ezt az erőforrás-szerkesztőben beállíthatjuk. Properties / IDC_MASODIK_EDIT / General / Disabled

A váltást az OnKillFocus eseménykezelőjébe tegyük, az elágazás helyes adatok esetén megvalósuló részébe, a győzelem vizsgálat mögé!

```
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(FALSE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(TRUE);
```

Ha váltunk, az előző ablakban ne legyen kiírva az előző választás! Ezt tehetjük a KillFocus eseménykezelőjébe. Ha nem figyel az ellenfél, utólag már nem tudja megnézni, mi volt az utolsó lépés. Nem is kell beírnunk semmit, hisz a kód az

elágazás igaz ágában már bent van, ez azt jelzi, kivehetjük az elágazásból. Így viszont az igaz ág üres lesz, tehát fogalmazzuk át a feltételt épp az ellenkezőjére! A módosított kód:

```
int nOszlop = atoi(Oszlop);
if (!Sor.IsEmpty() && !Oszlop.IsEmpty() && nOszlop >0 && nOszlop
< 20)
  {
m_Grid.SetRow(SorNev.Find(Sor)+1);
m_Grid.SetCol(nOszlop);
m_Grid.SetText("X");
if (Win(m_Grid.GetRow(),m_Grid.GetCol(),"X"))
MessageBox("Győzött az első játékos!!!");
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(FALSE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(TRUE);
  }
m_Elso="";
UpdateData(FALSE);
```

4.1.4. Egér használata

ClassWizard

Message Maps fül

CAmobaDlg osztály

IDC_MSFLEXGRID

Messages: Click Add Function

```
void CAmobaDlg::OnClickMsflexgrid()
{
if (GetDlgItem(IDC_ELSO_EDIT)->EnableWindow())
  {
m_Grid.SetText("o");
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(TRUE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(FALSE);
m_Elso ="";
UpdateData(FALSE);
  }
else
  {
m_Grid.SetText("X");
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(FALSE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(TRUE);
m_Masodik ="";
UpdateData(FALSE);
  }
}
```

Az UpdateData(FALSE) mindkét elágazásban szerepel, emeljük ki az elágazáson kívülre!

```
m_Masodik = "";
    }
UpdateData(FALSE);
}
```

Az OnClickMsflexgrid() tagfüggvényből is hívjuk meg a győzelem függvényt!

A már bejelölt cellákat ne engedjük felülírni! Tehát csak üres cella választása esetén vegyük figyelembe a kattintást az egérrel!

```
void CAmobaDlg::OnClickMsflexgrid()
{
if (m_Grid.GetText()=="")
 {
if (GetDlgItem(IDC_ELSO_EDIT)->EnableWindow())
    {
m_Grid.SetText("o");
if (Win(m_Grid.GetRow(),m_Grid.GetCol(),"o"))
        {
MessageBox("Győzött a második játékos!!!");
OnOK();
        }
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(TRUE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(FALSE);
m_Elso = "";
    }
else
    {
m_Grid.SetText("X");
if (Win(m_Grid.GetRow(),m_Grid.GetCol(),"X"))
        {
MessageBox("Győzött az első játékos!!!");
OnOK();
        }
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(FALSE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(TRUE);
m_Masodik = "";
    }
UpdateData(FALSE);
 }
}
```

A jelek fölülírását a billentyűzetről is tiltsuk meg!

```
if (!Sor.IsEmpty() && !Oszlop.IsEmpty() && nOszlop >0 && nOszlop
< 20)
 {
m_Grid.SetRow(SorNev.Find(Sor)+1);
m_Grid.SetCol(nOszlop);
if (m_Grid.GetText()=="")
    {
m_Grid.SetText("X");
```

```
if (Win(m_Grid.GetRow(),m_Grid.GetCol(),"X"))
        {
MessageBox("Győzött az első játékos!!!");
OnOK();
        }
GetDlgItem(IDC_ELSO_EDIT)->EnableWindow(FALSE);
GetDlgItem(IDC_MASODIK_EDIT)->EnableWindow(TRUE);
    }
  }
m_Elso="";
UpdateData(FALSE);
```

Ha az új játékos nyomógombját választottuk (BN_CLICKED), kapja meg a hozzá tartozó beviteli mező a fókuszt, ne kelljen még oda is rákattintani!

```
void CAmobaDlg::OnElsoButton()
{
GetDlgItem(IDC_ELSO_EDIT)->SetFocus( );
}
```

Vigyázat, ha nem szerepel az

```
UpdateData(TRUE);
if (m_Elso.IsEmpty()) return;
```

sor az OnKillFocus függvényeinkben, akkor, ha a nyomógomb választása után mégis úgy döntenénk, nincs kedvünk gépelni, egérrel folytatjuk, akkor debug assertion hibát kapunk! (A Mid függvény nem szereti, ha üres sztringre hívják.) Erről a hibáról nem szerzünk tudomást, ha Release módon fordítjuk le a programot. A hiba akkor is jelentkezik, ha beállítjuk a kezdőfókuszt az első játékos beviteli ablakára (SetFocus vagy Tab Order segítségével), s mi mégis egérrel szeretnénk kezdeni.

4.1.5. Ablakok a Spy++-ban

Vizsgáljuk meg az Amőba alkalmazás ablakait, ablakkezelőit a Spy++ segítségével! (Elmélet 4.8.3. szakasz.)

Tools menü **Spy++.**

4.2. Feladatok

4.2.1. Időmérés az Amőba feladatban

Korlátozzuk a játékosok egy lépésre szánt idejét, s a hátralevő értéket jelezzük egy ProgressBar vezérlő segítségével! Ha elfogyott a játékos ideje, a másik játékos kapja meg a választás lehetőségét, de ő is csak korlátozott ideig léphet.

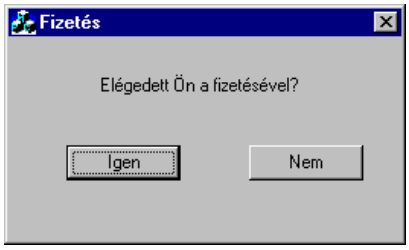
4.2.2. Kapj el gomb!*

Jelenjen meg egy nyomógomb az ablak tetszőleges pontjában, de bizonyos időközönként mindig új helyen! Ha az egérrel rákattintunk, álljon le a mozgás! Újabb kattintásra induljon újra a játék! (Baga, 1998. 95. old.) A feladat több, különböző időközönként mozgó gombbal is megvalósítható. Két továbbfejlesztett változata a 5.5.7. és az 6.6.4. feladat.

4. Gyakorlat 5. ábra 'Kapj el!' gomb

4.2.3. Elégedett Ön a fizetésével?*

A párbeszédablakon a kérdés mellett két nyomógombot találunk, Igen Nem felirattal. Ha egerünket az igen gomb fölé visszük, az elmozdul, így soha nem lesz kiválasztható. Ha a nem gombot választjuk: A főnöke elégedett Önnel! feliratú párbeszédablak OK gombjának leütése után vége az alkalmazásnak. (A feladat a számítógépes viccek kategóriájába tartozik, több helyről kaptam e-mailben.)



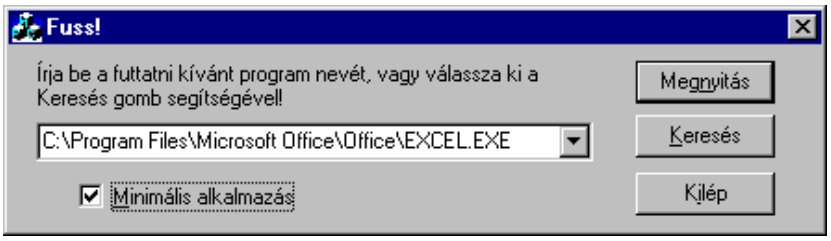
4. Gyakorlat 6. ábra Fizetés

4.2.4. Szerencsekerék

Készítsünk a szókirakóból (3.3.7. feladat) Szerencsekerék játékot! Tehát ne csak egy szót, hanem egy vagy több mondatot is ki lehessen találni! A helyes betű megadása után egy nyomógomb hatására véletlen számmal megadott ideig, vagy egy csúszkával megadott ideig pontszámok, dupláz, csőd felirat váltakozzanak a képernyőn! A versenyző pontszáma az idő letelte után mutatott értéknek megfelelően változzon, és nyeremény esetén folytathassa a játékot! Ha a választott betű nem szerepel a feliratban, a következő játékos következzen!

4.2.5. Fuss!*

Indítsunk újabb alkalmazásokat programunk segítségével! Az indításra váró alkalmazás útvonalát a kombipanelbe gépelve vagy a Keresés gomb segítségével kiválasztva, a megnyitás gomb hatására indítsuk el az alkalmazást! Ha minimalizálva akarjuk indítani, jelöljük a jelölőnégyzetet!



4. Gyakorlat 7. ábra A Fuss alkalmazás futás közben

4.2.6. DragList*

Készítsünk egy ablakot, melyen két lista látható! A listadobozok elemeit az egérrel, drag & drop technikával tetszőleges sorrendűre állíthatjuk. Egyik listából a másikba is tehessünk át elemeket!

4. Gyakorlat 8. ábra DragList

4.2.7. Feladatütemező

Alkalmazásunk egy táblázatot tartalmaz, melynek sorai és oszlopai egy-egy elvégzendő feladatot jelentenek. Ha az egyik feladat végrehajtásának feltétele a másik elvégzése, tegyünk egy X jelet a feltételfeladat sorába és a feltételt szabó feladat oszlopába. Ha van üres oszlopunk, ahhoz a feladathoz nem rendeltünk előzményt, így ő lehet az első. Sorát és oszlopát elhagyva megint vizsgálhatjuk az üres oszlopot, mint következő feladatot. Ha nincs üres oszlop, akkor egymást kölcsönösen feltételező feladataink vannak. A probléma megoldása nem várható az ütemezőtől. Vonjuk össze e feladatokat egy egységgé!

4. Gyakorlat 9. ábra A feladatütemező kezdetben és munka közben

4.2.8. Többfülű alkalmazás

Készítsünk alkalmazást, mely a tulajdonságablakhoz hasonlóan fülek segítségével lehetővé tesz beállításokat, majd az OK választására a beállításoknak megfelelő adatokat kiírja a képernyőre! (Vagy a megadott adatokkal ablakot készít.)

4.2.9. Varázsló

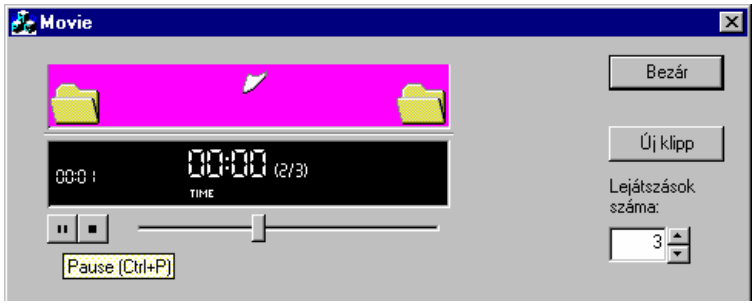
Alakítsuk át az előző feladatot többfülű ablak helyett varázslóra!

4.2.10. Fényújság

Készítsünk alkalmazást, mely valamilyen reklámszöveget görget jobbról balra! Ha véget ér a szöveg, kezdje újra kiírását elölről, mintha egy szalag két végét összeragasztottuk volna!

4.2.11. Videólejátszó*

Készítsünk alkalmazást, mely mozgóképeket mutat be!



4. Gyakorlat 10. ábra A videó lejátszó

4.3. Ötletek a feladatok megoldásához:

4.2.2. Kapj el gomb!

Azt, hogy mikor kell a gombnak arrébb ugrani, egy időzítő segítségével adhatjuk meg. Az időzítőt a Dlg osztály OnInitDlg metódusában inicializáljuk.

```
#define MOVE 1
 //...
SetTimer(MOVE,1000,NULL);
```

Az inicializált időzítőt a ***DestroyWindow()*** virtuális tagfüggvény felülírásával állíthatjuk le. (Az osztályvarázsló a Messages között sorolja fel a virtuális függvényeket is.)

```
BOOL CKapjElDlg::DestroyWindow()
{
// TODO: Add your specialized code here and/or
//call the base class
KillTimer(MOVE);
return CDialog::DestroyWindow();
}
```

A gombot mozgassuk a **WM_TIMER** üzenethez rendelt kezelő függvénnyel: ***OnTimer()***.

```
void CKapjElDlg::OnTimer(UINT nIDEvent)
{
CRect buttonRect;
GetDlgItem(IDC_BUTTON)->GetClientRect(&buttonRect);
CRect dlgRect;
GetClientRect(dlgRect);
int cx=dlgRect.right-buttonRect.right;
int cy=dlgRect.bottom-buttonRect.bottom;
//A gomb méretét kivonjuk, hogy ne lógjon ki a képből.
cx =rand()*cx/RAND_MAX;
cy =rand()*cy/RAND_MAX;
GetDlgItem(IDC_BUTTON)->
MoveWindow(cx,cy,buttonRect.right,buttonRect.bottom);
CDialog::OnTimer(nIDEvent);
}
```

A gombot a BN_CLICKED üzenetkezelőjével állíthatjuk le, illetve indíthatjuk újra. Ehhez szükség lesz egy adattagra, mely tárolja, hogy a gomb mozog-e. BOOL m_Move, melynek TRUE kezdőértékét a konstruktorban állítjuk be.

```
void CKapjElDlg::OnButton()
{
m_Move=!m_Move;
}
```

Az OnTimer-ben m_Move értékétől függően mozog a gomb

```
void CKapjElDlg::OnTimer(UINT nIDEvent)
{
if (m_Move)
  {
CRect buttonRect;
GetDlgItem(IDC_BUTTON)->GetClientRect(&buttonRect);
...
CDialog::OnTimer(nIDEvent);
  }
}
```

A feladat két időzítő kezelésével két vagy több, különböző időközönként elmozduló gombot is tartalmazhat. Ehhez a különböző időzítők üzeneteit külön-külön kell kezelnünk. Mivel a gombok egymás fölé is elmozdulhatnak, így újrafestésükről gondoskodnunk kell.

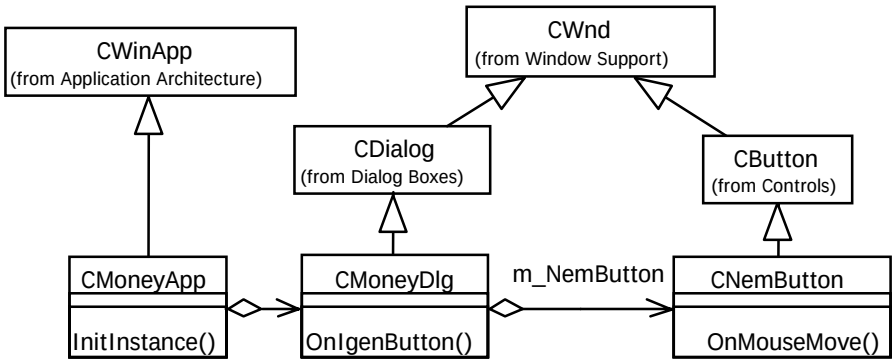
```
void CKapjElDlg::OnTimer(UINT nIDEvent)
{
CRect dlgRect;
GetClientRect(dlgRect);
switch (nIDEvent)
 {
case GYORS:
if (m_Move)
        {
CRect buttonRect;
GetDlgItem(IDC_BUTTON)->GetClientRect(&buttonRect);
int cx=dlgRect.right-buttonRect.right;
int cy=dlgRect.bottom-buttonRect.bottom;
//A gomb méretét kivonjuk, hogy ne lógjon ki a képből!
cx =rand()*cx/RAND_MAX;
cy =rand()*cy/RAND_MAX;
GetDlgItem(IDC_BUTTON)->
MoveWindow(cx,cy,buttonRect.right,buttonRect.bottom);
        }
break ;
case LASSU:
if (m_MoveLassu)
        {
CRect buttonRect;
GetDlgItem(IDC_LASSU_BUTTON)->
GetClientRect(&buttonRect);
int cx=dlgRect.right-buttonRect.right;
int cy=dlgRect.bottom-buttonRect.bottom;
//A gomb méretét kivonjuk, hogy ne lógjon ki a képből!
cx =rand()*cx/RAND_MAX;
cy =rand()*cy/RAND_MAX;
GetDlgItem(IDC_LASSU_BUTTON)->
MoveWindow(cx,cy,buttonRect.right,buttonRect.bottom);
        }
    }
// A gombok újrarajzolása.
GetDlgItem(IDC_BUTTON)->RedrawWindow();
GetDlgItem(IDC_LASSU_BUTTON)->RedrawWindow();
CDialog::OnTimer(nIDEvent);
}
```

4. Gyakorlat 11. ábra Kétgombos 'Kapj el!'.

4.2.3. Elégedett Ön a fizetésével?

A feladat megoldása azért érdekes, mert az Igen gombon a **WM_MOUSEMOVE** üzenetet kell feldolgoznunk. Bár tudjuk, hogy **a CButton a CWnd utódosztálya**, az osztályvarázsló mégsem kínálja föl ezt az üzenetet a Dlg osztályban a feldolgozásra. Tehát létre kell hoznunk egy saját CNemButton osztályt, melyben kezelni tudjuk a WM_MOUSEMOVE üzenetet. Ezután a Nem gombhoz egy CNemButton típusú adattagot rendelünk. Az objektumon keresztül kezeljük az egér mozgása eseményt.

A megoldás osztálydiagramja:



4. Gyakorlat 12. ábra A vezérlő objektum osztálya a CButton utódosztálya

Az Igen gomb kezelése az egyszerűbb:

```
void CMoneyDlg::OnIgenButton()
{
MessageBox("A főnöke elégedett Önnel!");
OnOK();
}
```

Készítsük el az osztályvarázslóval a CNemButton osztályt!

Osztályvarázsló

Add Class **New**

Class name: CNemButton

Base class: CButton **OK**

Member Variables fül

IDC_NEM_BUTTON **Add Variable**

Name: m_NemButton

Category: Control

Variable type: CNemButton **OK**

Be kell szerkesztenünk a Dlg osztály fejlécfájljába az új osztály fejlécfájlját.

Rendeljünk a CNemButton osztályhoz egy WM_MOUSEMOVE üzenetkezelőt, és írjuk meg a kódját az előző feladat OnTimer kódjához hasonlóan!

```
void CNemButton::OnMouseMove(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or
//call default
CRect buttonRect;
GetClientRect(&buttonRect);
CRect dlgRect;
GetParent()->GetClientRect(dlgRect);
int cx=dlgRect.right-buttonRect.right;
int cy=dlgRect.bottom-buttonRect.bottom;
//A gomb méretét kivonjuk, hogy ne lógjon ki a képből.
cx =rand()*cx/RAND_MAX;
cy =rand()*cy/RAND_MAX;
MoveWindow(cx,cy,buttonRect.right,buttonRect.bottom);
CButton::OnMouseMove(nFlags, point);
}
```

Ha azt akarjuk, hogy az Igen gomb fölött is mozduljon el a Nem gomb az egér alól, akkor a Nem gombot helyezzük fókuszba úgy, hogy a tabulátor-sorrendben az 1. helyet adjuk neki. (ResourceView / Layout / Tab Order)

4.2.5. Fuss!

Készítsük el a 4. Gyakorlat 7. ábra alapján az alkalmazás főablakát!

A kombipanelünkhöz most kétféle kategóriájú változót is rendelünk. Az egyik a beírt szöveget tartalmazó CString m_CommandLine; a másik a vezérlő típusú **CComboBox** m_Combo. A jelölőnégyzethez rendeljük a BOOL m_bMinimize; adattagot.

A Megnyitás gombhoz rendelt ***OnOpen()*** metódus a feladat lelke, mely a ShellExecuteEx függvény segítségével indítja a kiválasztott alkalmazást:

```
void CRunDlg::OnOpen()
{
UpdateData();
SHELLEXECUTEINFO sei;
 sei.cbSize          = sizeof(sei);
 sei.fMask           = 0;
 sei.hwnd            = GetSafeHwnd();
 sei.lpVerb          = NULL;
 sei.lpFile          = m_CommandLine;
sei.lpParameters= NULL;
 sei.lpDirectory     = NULL;
 sei.nShow           = m_bMinimize? SW_SHOWMINIMIZED :
SW_SHOWNORMAL;
 sei.hInstApp = AfxGetInstanceHandle();

if(ShellExecuteEx(&sei))
 {
if(CB_ERR == m_Combo.FindStringExact(0, m_CommandLine))
// Ha sikerült végrehajtani, akkor előbb ellenőrizzük, nincs-e
// az üzenetsorban, s ha nincs, tegyük be a kombipanel sorába!
    {
m_Combo.AddString(m_CommandLine);
    }

m_Combo.SetWindowText("”); //Törli a kombipanel szövegét.
 }
}
```

Ha nem futtatható alkalmazást választunk, de a fájlhoz hozzá van rendelve az az alkalmazás, mely adatait értelmezni tudja, úgy az is elindul e kód hatására.

A sei.nShow beállításánál figyelembe vettük a min. jelölőnégyzet állapotát.

A Keresés gomb hatására a **CFileDialog** osztály felhasználásával tudjuk egyszerűen és más alkalmazásokban megszokott módon megtalálni az indításra váró fájlt.

```
void CRunDlg::OnBrowse()
{
static char BASED_CODE szFilter[] = "All Files (*.*) |*.*||”;
```

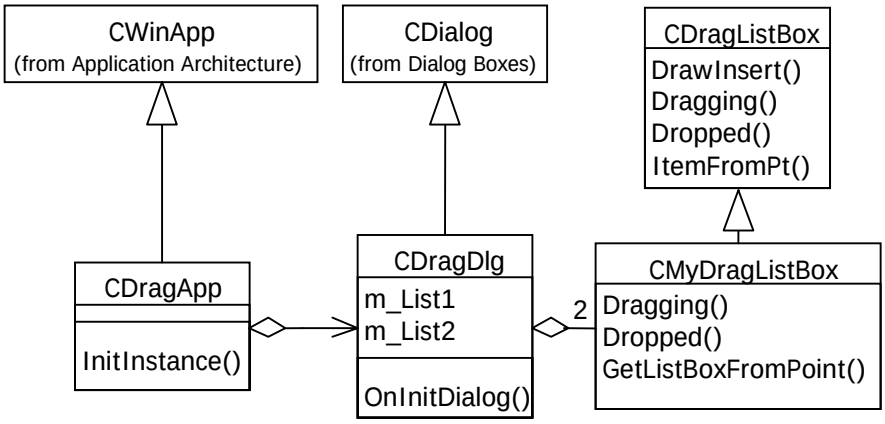
```
 CFileDialog Dlg(TRUE,      // Megnyitja a File Dialogot
                 NULL,      // Alapértelmezett kiterjesztés
                 NULL,      // Kezdő fájlnév
                 OFN_HIDEREADONLY,       // Flags
                 szFilter );             // Filter

 if (IDOK == Dlg.DoModal())              // Megjelenít
 {
 m_CommandLine = Dlg.GetPathName();      // Kiszedi az útvonalat
a párbeszédablakból, s tárolja az ablak adatmezőjében.
 UpdateData (FALSE);                     // Átírja az
adatmezőket, a változóból a képernyőre írja a beolvasott adatot.
 }
}
```

Próbáljuk ki a kezdő fájlnév megadását pl. _T(„C:\\Program Files\\Microsoft Office\\Office\\Excel.exe”), az adott könyvtár adott fájlját kínálja föl kezdésnek.

Nézzük meg Spy++ segítségével az elindított folyamatokat!

4.2.6. DragList

Ha csak egy listán belül akarunk vonszolni elemeket, akkor elegendő a listadobozunkhoz egy CDragListBox típusú vezérlő objektumot rendelni, s a művelet működik. Ha azonban egyik listából a másikba is szeretnénk vonszolni, fölül kell írnunk a **CDragListBox** osztály ***Dragging()*** és ***Dropped()*** metódusait. A Dragging feladata a vonszolás során annak jelzése, hogy a másik ablak is elfogadja a vonszolt elemet. A Dropped metódusban adjuk meg, mi történjen, ha az ablak fölött elengedjük a vonszolt elemet! A metódusok fölülírása egy utódosztályban történik (CMyDragListBox).



4. Gyakorlat 13. ábra A DragList alkalmazás osztálydiagramja

A feladat nemcsak azért érdekes, mert egy új vezérlő kezelésével találkozunk, hanem azért is, mert példát látunk arra, hogyan érhetjük el az egyik vezérlő metódusából a szülőablakon keresztül a másik vezérlőt.

Készítsük el a párbeszédalapú projektet az alkalmazásvarázsló segítségével! Helyezzük el a listadobozokat a felületen! Ügyeljünk rá, hogy a **CDragListBox** elemei nem lehetnek rendezettek (Properties / Styles / Sort ne legyen kijelölve)!

Rendeljünk Control kategóriájú objektumot az erőforrásokhoz (m_List1, m_List2)! Az osztályvarázsló csak CListBox-ot enged kiválasztani, de a CDragDlg osztály deklarációjában az adattagok típusát átírhatjuk CDragListBox-ra.

Az OnInitDialog tagfüggvényben feltölthetjük elemekkel a listákat (***AddString***). Az egyiket számokkal, a másikat betűkkel.

```
BOOL CDragDlg::OnInitDialog()
{
...
// TODO: Add extra initialization here
for (int i=0; i<6; i++)
  {
m_List1.AddString(CString('1'+i));
m_List2.AddString(CString('A'+i));
  }
return TRUE;
}
```

Egy listán belül már megy a vonszolás, feltéve, hogy nem felejtettük el a rendezettség kikapcsolását.

Hozzuk létre az osztályvarázslóval az új CMyDragListBox osztályt, mely a CDragListBox utóda legyen! Ahhoz, hogy a CDragDlg tudjon az új osztályról, a fejlécfájlt be kell szerkesztenünk! Ezután legyen a két lista vezérlőnk osztálya a CMyDragListBox! Még nem változott semmi a futó alkalmazásban! Az egyik listából a másikba nem tudunk vonszolni.

Először meg kell tudnunk a vonszolás alatt a forrás listadobozban, hogy milyen ablak fölött van a kurzor. Ehhez készítünk egy függvényt GetListBoxFromPoint(), mely NULL értékkel tér vissza, ha a kurzor nem listadoboz fölött van, egyébként pedig a listadoboz mutatójával. A ***CWnd::WindowFromPoint()*** (statikus) metódusa visszaadja az aktuális kurzorpozíción található ablak mutatóját. Figyeljük meg, hogyan használhatjuk a futásidejű információkat annak eldöntésére, hogy egy ablak CDragListBox típusú-e!

```
CDragListBox* CMyDragListBox::GetListBoxFromPoint(CPoint point)
{
CWnd* pWnd= WindowFromPoint(point);
if (pWnd == NULL ||
!pWnd->IsKindOf(RUNTIME_CLASS(CDragListBox)))
return NULL;
return (CDragListBox*)pWnd;
}
```

A vonszolás alatt mutatnia kell a másik listának, hogy hova kerülne az elem, ha elengednénk! Ezt a ***Dragging*** metódus valósítja meg.

```
UINT CMyDragListBox::Dragging(CPoint pt)
{
CDragListBox* pListBox=GetListBoxFromPoint(pt);
// Ha nem listadoboz fölött van a kurzor:
if (pListBox== NULL)
return DL_STOPCURSOR;
// Ha listadoboz fölött van a kurzor:
return pListBox->CDragListBox::Dragging(pt);
}
```

Most már jelzi a vonszolás fogadását, de még nem fogadja a másik lista. Ez a ***Dropped*** metódus felülírásával érhető el. Ha a cél és a forrás DragListBox megegyezik, akkor hívhatjuk az ősosztály azonos metódusát. Nekünk csak különböző listadobozok esetén kell megvalósítanunk a vonszolást.

A ***CDargListBox::ItemFromPt()***tagfüggvénye visszaadja a paraméterben megadott pont alatti elem indexét a listában.

```
void CMyDragListBox::Dropped(int nSrcIndex, CPoint pt)
{
ASSERT(!(GetStyle() &
(LBS_OWNERDRAWFIXED|LBS_OWNERDRAWVARIABLE)) ||
(GetStyle() & LBS_HASSTRINGS));
// Lekéri a kurzor alatti ListBox mutatóját.
CDragListBox* pListBox=GetListBoxFromPoint(pt);

// Nem ListBox van a kurzor alatt:
if (pListBox==NULL)
return;
// Ha a cél és a forrás azonos:
if (pListBox==this)
  {
CDragListBox::Dropped(nSrcIndex, pt);
  }
```

```
else
  {
// Másik LisBoxba vonszoltunk:
int nDestIndex=pListBox->ItemFromPt(pt);
if (nDestIndex >=0 )
    {
// Forrásvezérlőből olvasunk és törlünk:
CString str;
GetText(nSrcIndex, str);
DeleteString(nSrcIndex);

// A célvezérlőbe beszúrjuk az új elemet:
nDestIndex=pListBox->InsertString(nDestIndex,str);
pListBox->SetCurSel(nDestIndex);
    }
  }
}
```

4.2.11. Videólejátszó

Használjuk az ActiveMovieControlt az **ActiveX** vezérlők közül!

A lejátszásra váró fájl a **CFileDialog** segítségével kereshető meg. Ha beállítottuk az ActiveX vezérlőhöz rendelt m_Movie objektumra a lejátszandó fájl nevét, a vezérlő gombjaival kezdődhet a lejátszás.

```
void CMovieDlg::OnUjButton()
{
CFileDialog dlg(TRUE, "avi", "C:\\Program Files\\
Visual Studio\\Common\\Graphics\\Videos\\Filecopy.avi",
OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
"Avi files (*.avi)|*.avi|All files (*.*)|*.*||");
if (dlg.DoModal())
  {
m_Movie.SetFileName(dlg.GetFileName());
  }
}
```

Beállíthatjuk a lejátszások számát is léptetőgomb segítségével (Styles: Right Alignment, Auto buddy, Set buddy integer). Lásd még 5.3. gyakorlat 'A vonalvastagság beállítása'!

A szerkesztőmezőhöz rendeljük az **int** típusú m_Count változót! Állítsuk be értékhatárait! Adjuk meg kezdőértékét a konstruktorban!

Az ablak inicializálásakor állítsuk be a léptetőgomb értékhatárait!

```
BOOL CMovieDlg::OnInitDialog()
{
  ...
// TODO: Add extra initialization here
// m_Movie.SetFileName("...");
((CSpinButtonCtrl*)GetDlgItem(IDC_SPIN))->SetRange32(0,50);
return TRUE;  // return TRUE unless you set the focus to a
control
}
```

A szövegdoboz értékének módosításához fókuszba kerül. Ha akkor állítjuk be a videolejátszón a lejátszások számát, amikor elveszíti a fókuszt, a függvény mind a léptetőgombos, mind pedig a szerkesztőmezőbe beírt változtatások esetén meghívódik.

```
void CMovieDlg::OnKillfocusNumEdit()
{
UpdateData();
m_Movie.SetPlayCount(m_Count);
}
```

5. Gyakorlat A grafikus felület kezelése

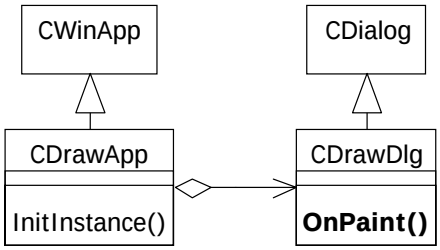
5.1. Rajz készítése

Készítsünk alkalmazást, mely egy előre elkészített ábrát rajzol ki minden frissítés során!

Készítsünk egy párbeszédalapú alkalmazást Draw néven! Vegyük ki az ablakból a 'TODO: Place...' feliratot!

5.1.1. Rajzoljunk ablakunkba egy ábrát!

Ha azt akarjuk, hogy a rajz a képernyőn frissüljön, pl. az ablak minimalizálása és újbóli megnyitása illetve a fölé kerülő ablak bezárása után is látsszon, akkor a rajzolás kódját az ***OnPaint()*** metódusba tegyük. Az OnPaint a WM_PAINT üzenet kezelője. A Windows a **WM_PAINT** üzenetet küldi, ha frissíteni kell az ablakot. Természetesen írhatjuk a kódot az OnPaint által hívott metódusba is. Ez történik Document / View architektúra esetén is, amikor a View osztály OnDraw metódusa tartalmazza a megjelenítés kódját (6.1. gyakorlat).



5. Gyakorlat 1. ábra Rajzolás párbeszédalapú alkalmazásban. A frissítést az ablak OnPaint metódusa biztosítja.

Az CDrawDlg::OnPaint metódust az alkalmazásvarázsló már elkészítette nekünk. Az ***IsIconic()*** függvény visszatérési értéke igaz, ha ablakunkat minimalizáljuk. A kód ebben az esetben rajzolja ki az alkalmazás ikonját. A **CPaintDC** objektum létrehozását vegyük ki az IsIconic blokkból, mi is rajzolni szeretnénk rá az elágazás else ágában!

```
void CDrawDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 if (IsIconic())
 {
    ...
 }
 else
 {
    dc.Rectangle(100,100, 200,200);            // Ház.
    dc.MoveTo(90, 110);                        // Tető.
    dc.LineTo(150, 50);
    dc.LineTo(210, 110);

    CDialog::OnPaint();
 }
}
```

Ha kedvünk van, rajzoljunk tovább!

Teszteljük a rajzot, hogy valóban frissül-e az ablak újrafestésekor! Mozgassuk az ablak egy részét a képernyőn kívülre, nyissunk fölé másik ablakot, majd tegyük újra a többi ablak fölé! Ha az ablak tulajdonságlapján beállítottuk a Minimize box tulajdonságot, akkor minimalizálni is tudjuk, majd visszaállíthatjuk az eredeti méretét.

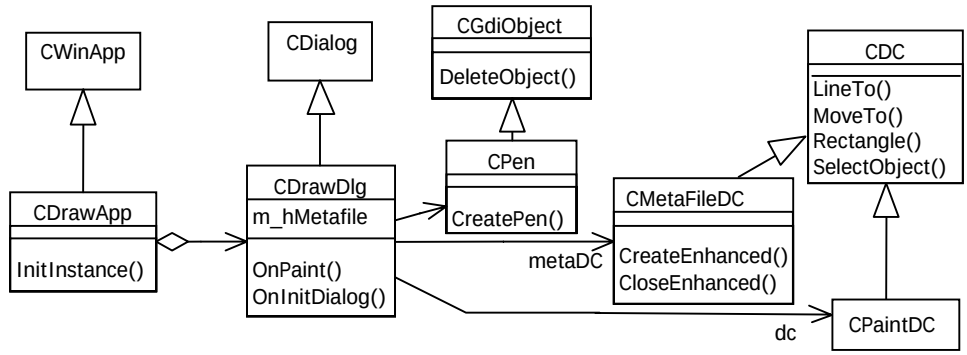
➢ Színezzük a rajzot!

Legyenek a vonalak vastagabbak, a tető piros!

```
else
{
   CPen newPen;
   newPen.CreatePen(PS_SOLID,3,RGB(128,128,128));
   CPen* pOriginalPen=dc.SelectObject(&newPen);
   dc.Rectangle(100,100, 200,200);            // Ház.
   CPen redPen;
   redPen.CreatePen(PS_SOLID,5,RGB(255,0,0));
   dc.SelectObject(&redPen);
   dc.MoveTo(90, 110);                        // Tető.
   dc.LineTo(150, 50);
   dc.LineTo(210, 110);

   dc.SelectObject(pOriginalPen);
   newPen.DeleteObject();
   redPen.DeleteObject();
   CDialog::OnPaint();
}
```

5.1.2. Tároljuk a rajzot metafájlban!

Ha a rajzot metafájlban tároljuk, a metafájlt az OnInitDialog metódusban hozzuk létre, és egy kezelőjét (m_hMetafile) tároljuk az osztályban. Az OnPaint metódus a kezelő ismeretében tudja lejátszani a metafájlt.



5. Gyakorlat 2. ábra Rajzolás metafájl segítségével.

Ehhez fel kell vennünk egy `CMetaFileDC metaDC;` lokális változót az OnInitDialog-ban, majd inicializálnunk kell. Ha paraméterként nem adunk meg fájlnevet (alapértelmezés: NULL), akkor a memóriában tároljuk a leíró fájl tartalmát.

Erre kell elkészítenünk a rajzot!

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

 // TODO: Add extra initialization here
 CMetaFileDC metaDC;
 metaDC.CreateEnhanced(NULL,NULL,NULL,NULL);
 CPen newPen;
 newPen.CreatePen(PS_SOLID,3,RGB(128,128,128));
 CPen* pOriginalPen=metaDC.SelectObject(&newPen);
 metaDC.Rectangle(100,100, 200,200);           // Ház.
 CPen redPen;
 redPen.CreatePen(PS_SOLID,5,RGB(255,0,0));
 metaDC.SelectObject(&redPen);
 metaDC.MoveTo(90, 110);                       // Tető.
 metaDC.LineTo(150, 50);
 metaDC.LineTo(210, 110);
 metaDC.SelectObject(pOriginalPen);
 newPen.DeleteObject();
 redPen.DeleteObject();

 return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

➢ A metafájl lezárása

A lezárásnál rendeljünk a metafájlhoz egy **HENHMETAFILE** típusú, m_hMetaFile nevű kezelőt. Ehhez fel kell vennünk a CDrawDlg osztályban egy `private: HENHMETAFILE m_hMetafile;` adattagot! (A Class Viewban CDrawDlg osztály nevén jobb egérrel kattintva válasszuk az Add Member Variable menüpontot!) Ezen keresztül érhetjük el a későbbiekben a metafájlt. Azért kell az osztály tagjaként deklarálnunk, hogy az alkalmazás futása során többször és más tagfüggvényekből is lássuk. A **CMetaFileDC** objektum bezárása után visszakapjuk a kezelőt, mely a metafájlban tárolt műveletek leírására mutat. A zárás után a metaDC objektum m_hDC adattagja üres lesz, így az objektumba tovább rajzolni nem lehet.

A bezárás kódját még az OnInitDialog függvénybe írjuk!

```
m_hMetafile=metaDC.CloseEnhanced();
```

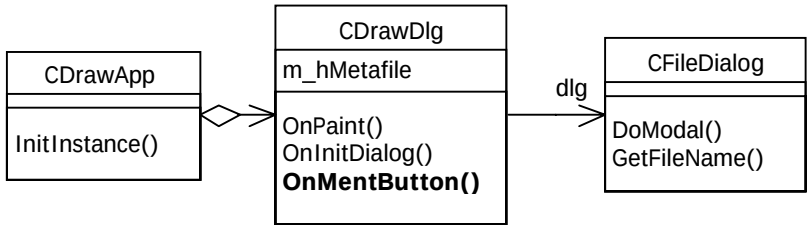
➢ A metafájl lejátszása

Mivel továbbra is azt akarjuk, hogy ablakunkban frissüljön a rajz, tehát a metafájlt az OnPaint metódusban kell lejátszani!

```
else
{
   PlayEnhMetaFile(dc.m_hDC,m_hMetafile,CRect(0,20,200,220));
   CDialog::OnPaint();
}
```

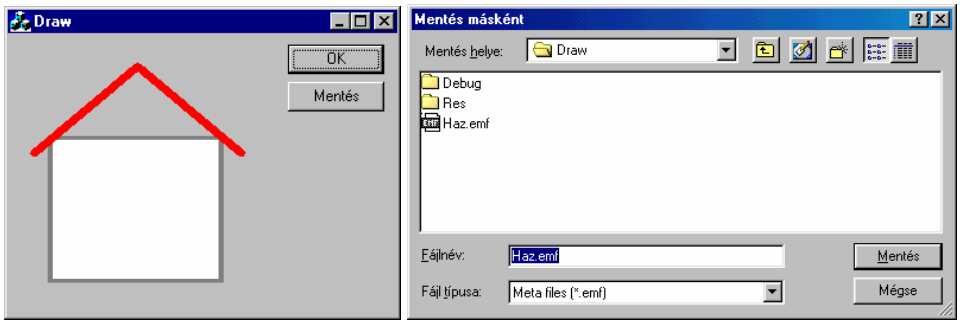
➢ Metafájl mentése a háttértárba

Ha fájlba akarjuk menteni rajzunkat, akkor a ***CopyEnhMetaFile()*** API függvény segít ebben. A fájl nevét és helyét a merevlemezen egy FileDlg ablak segítségével választhatjuk ki. A mentést egy 'Mentés' feliratú gomb kattintás eseményéhez csatoljuk.



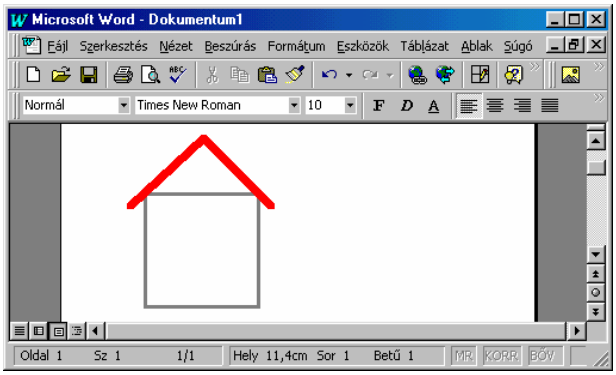
5. Gyakorlat 3. ábra A metafájl mentése gomb választásra

```
void CDrawDlg::OnMentButton()
{
 CFileDialog dlg(FALSE, "emf", "Haz.emf",
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "Meta files (*.emf)|*.emf |All files (*.*)|*.*||");
 dlg.DoModal();
 CopyEnhMetaFile(m_hMetafile, dlg.GetFileName());
}
```



5. Gyakorlat 4. ábra A mentés gomb választása

A mentés eredményességét kipróbálhatjuk, ha pl. egy Word dokumentumba beszúrjuk képként a **emf** fájlt.



5. Gyakorlat 5. ábra A Haz.emf kép a Wordben

Feladat:

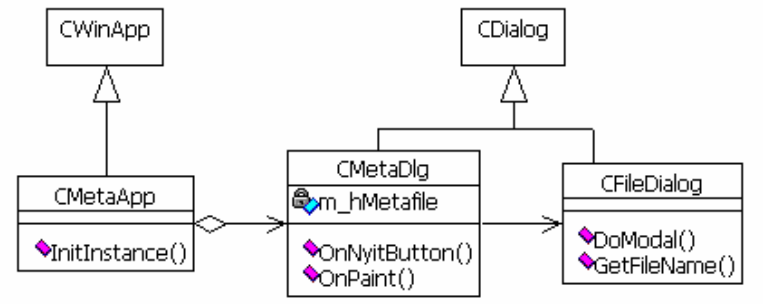
Készítsük el a rajzoló alkalmazásunkat úgy, hogy az alkalmazás .wmf kiterjesztésű fájlba legyen menthető!

Megjegyzés:

Ha .wmf-el dolgozunk és Wordben nem látnánk a képet, kattintsunk rá kétszer a Wordben, hogy megnyissuk, s egy egész kicsi ábrát várjuk a bal fölső sarokban! A bezárás után már a fődokumentum is mutatja a képet.

5.1.3. Metafált kirajzoló alkalmazás!

Készítsünk egy párbeszéd alapú alkalmazást Meta néven!



5. Gyakorlat 6. ábra A metafájlt kirajzoló alkalmazás

Az ablaka legyen minimalizálható, maximalizálható! (Properties / Styles / Minimize box) Tegyünk rá egy megnyit feliratú gombot! (IDC_NYIT_BUTTON)

A CMetaDlg-ban vegyünk fel egy HENHMETAFILE m_hMetafile adattagot! A konstruktorban kapjon NULL kezdőértéket! A metafájlt ***GetEnhMetaFile()*** API függvény segítségével olvashatjuk be!

Készítsük el a 'Megnyitás' gomb kezelőjét!

```
void CMetaDlg::OnNyitButton()
{
 CFileDialog dlg(TRUE, "emf", "",
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "Meta files (*.emf)|*.emf|All files (*.*)|*.*||");
  if (dlg.DoModal())
  {
   m_hMetafile=GetEnhMetaFile(dlg.GetFileName());
   Invalidate();
  }
}
```

Rajzoljuk ki a képet az OnPaint-ben! Ehhez megint tegyük az elágazáson kívülre a CPaintDC dc(this) objektum létrehozását! Az else ág kódja itt következik:

```
 else
 {
    PlayEnhMetaFile(dc.m_hDC,m_hMetafile, CRect(0,20,200,220));
    CDialog::OnPaint();
 }
```

Próbáljuk ki alkalmazásunkat különböző emf fájlok kirajzolására!

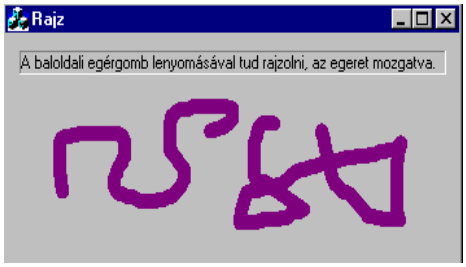
Feladat:

Készítsük el alkalmazásunkat úgy, hogy segítségével wmf kiterjesztésű fájlok is megnyithatóak legyenek!

5.2. Rajzolás egérrel

A program, ha a bal egérgombot lenyomva tartjuk és húzzuk az egeret, rajzolni kezd. A gombot fölengedve a vonal húzása megszűnik, újra lenyomva az új kezdőponttól rajzolunk tovább.

A feladat végén átállítjuk az egérkurzort általunk rajzoltra. A pihenőidőben animált kurzorral jelezzük: a program unatkozik.

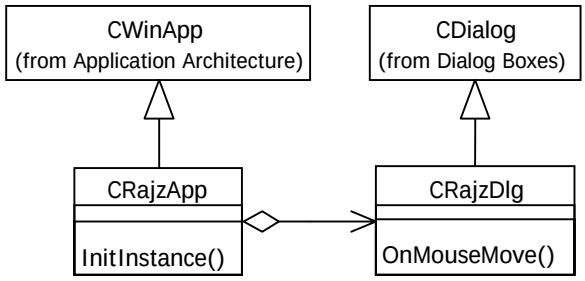


5. Gyakorlat 7. ábra A rajzolóprogram működés közben

A feladat továbbfejlesztése egy a jobb egérgomb hatására felbukkanó menüvel, melyben kiválaszthatjuk a rajzoló színt, a vonalvastagságot és a vonal stílusát, a menükezelés után az 6. fejezet feladatai közt szerepel.

5.2.1. Terv

Ha mozgatjuk az egeret, a WM_MOUSEMOVE üzenethez csatolt OnMouseMove kezelőmetódusban lekérdezzük, hogy a bal oldali egérgombot lenyomva tartjuk-e a mozgatás közben. Ha igen, pontokat teszünk ki a képernyőre.



5. Gyakorlat 8. ábra A kezdeti elképzelés terve

5.2.2. Rajzolás pontokkal

Az alkalmazásvarázsló segítségével készítsünk egy párbeszédalapú keretprogramot Rajz néven!

A vizuális felület mindössze egy tájékoztató sor legyen, mely kiírja, hogyan rajzolhatunk az alkalmazásban!

Először próbáljuk meg pontokból kirajzolni a vonalat! Újabb pontot az egér mozgatásakor kell a képernyőre tenni, tehát ezt az eseményt kell kezelnünk.

WM_MOUSEMOVE eseményhez tartozó ***OnMouseMove()*** metódus kódjában (az osztályvarázslóval hozzuk létre a kezelő metódust!) a DC-re pontokat rajzolunk:

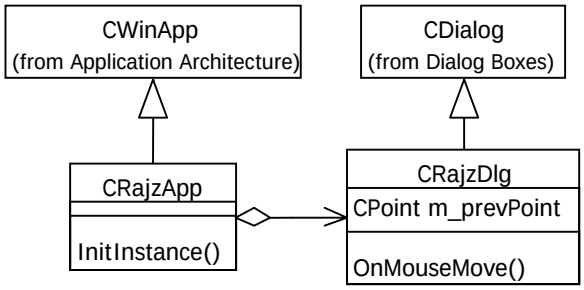
```
void CRajzDlg::OnMouseMove(UINT nFlags, CPoint point)
{
 if ((nFlags & MK_LBUTTON) == MK_LBUTTON )
 {
    CClientDC dc(this);
    dc.SetPixel(point, RGB(0,0,0));
 }
 CDialog::OnMouseMove(nFlags, point);
}
```

Teszteljük! Próbáljuk ki! Ha gyorsan mozgatjuk az egeret, ritkán kerülnek a pontok a képernyőre! Láthatjuk, hogy a rendszer milyen időközönként küldte el a WM_MOUSEMOVE üzenetet.

5.2.3. Rajzolás vonalakkal

- **Folytonos rajz**

Ha folytonos vonalat akarunk rajzolni, nem elég az egér aktuális pozíciójának ismerete, meg kell őriznünk az előző pontot is, hogy összeköthessük őket!



5. Gyakorlat 9. ábra Rajzolás vonalhúzással

A RajzDlg.h-ban a CRajzDlg osztályban hozzunk létre egy m_prevPoint nevű adatmezőt! A legegyszerűbben az osztály nevén jobb egérgombbal kattintva, s az Add Member Variable menüpontot választva tehetjük (5. Gyakorlat 10. ábra):

5. Gyakorlat 10. ábra Új adattag felvétele

```
private:
  CPoint m_prevPoint;
```

Az előző kódba dc.SetPixel helyett:

```
dc.MoveTo( m_prevPoint);
dc.LineTo( point);
m_prevPoint= point;
```

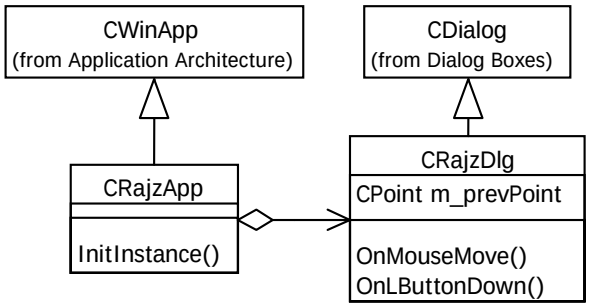
A kezdőpont a rajzolásnál véletlen pont, hisz nem inicializáltuk, és ha felengedjük az egérgombot, akkor ugyan nem rajzol a program, de ha újrakezdjük a rajzolást, az új kezdőpontot összeköti az előző végponttal.

- **Ne csak összefüggő rajz legyen!**

Az előző rajz végét ne kösse össze a következő elejével!

A **WM_LBUTTONDOWN** eseményhez rendeljünk kezdő értékadást! (A kezelőfüggvényt az osztályvarázslóval készítsük el!)

```
void CRajzDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
 m_prevPoint=point;
 CDialog::OnLButtonDown(nFlags, point);
}
```



5. Gyakorlat 11. ábra A vonal ne legyen összefüggő!

➢ Színes vonalat szeretnénk

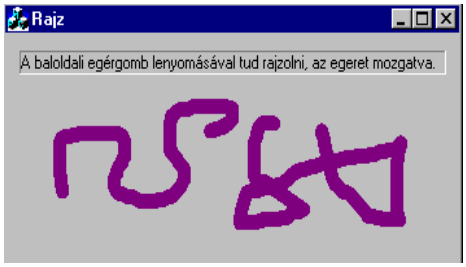
Az OnMouseMove tagfüggvénybe a rajzolás előtt:

```
CPen newPen(PS_SOLID, 10, RGB( 255, 0, 0 ));
CPen* pOriginalPen;
pOriginalPen = dc.SelectObject(&newPen);
```

A rajz után: `dc.SelectObject(pOriginalPen);`

A teljes kód:

```
void CRajzDlg::OnMouseMove(UINT nFlags, CPoint point)
{
 if ((nFlags & MK_LBUTTON) == MK_LBUTTON )
 {
    CClientDC dc(this);
    CPen newPen(PS_SOLID,10,RGB(100,0,100));
    CPen* pOriginalPen = dc.SelectObject(&newPen);
//      dc.SetPixel(point, RGB(0,0,0));
    dc.MoveTo(m_prevPoint);
    dc.LineTo(point);
    m_prevPoint=point;
    dc. SelectObject(pOriginalPen);
    newPen.DeleteObject();
 }

 CDialog::OnMouseMove(nFlags, point);
}
```



5. Gyakorlat 12. ábra A rajzolóprogram működés közben

➢ Tároljuk a rajzolóobjektumot, ne hozzuk mindig létre!

Hozzunk létre egy `CPen m_newPen`; adatmezőt a Dlg osztályban!

Inicializáljuk az ***OnInitDialog()***-ban!

```
m_newPen.CreatePen(PS_SOLID,10,RGB(100,0,100))
```

Felszámoljuk az ablakkal együtt a ***DestroyWindow()***-ban! (A DestroyWindow felülírásához az osztályvarázsló Messages ablakában válasszuk ki a DestroyWindow virtuális függvényt!)

```
m_newPen.DeleteObject();
```

Ezt a tollat válasszuk az OnMouseMove tagfüggvényben!

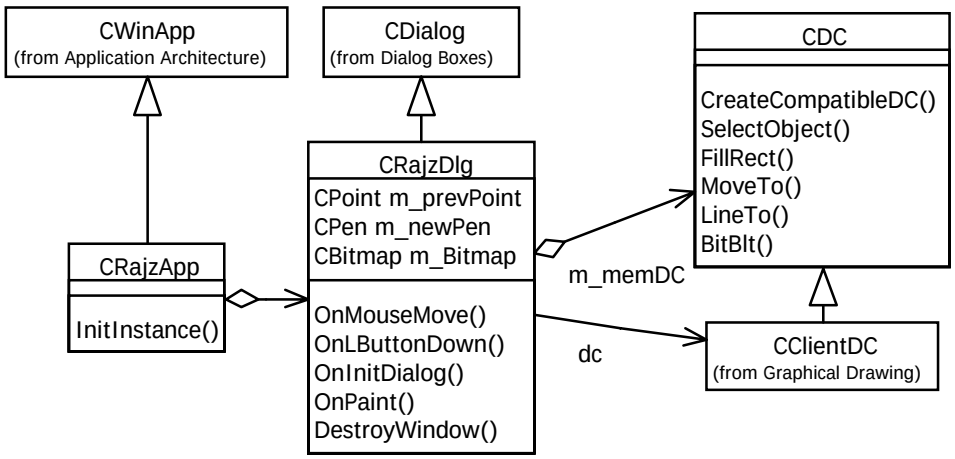
➢ A rajz ne takarja el a feliratot, haladjon mögötte!

A rajzunk a feliratra is rárajzol. Ha azt akarjuk, hogy a felirat szövege mögé kerüljön, akkor állítsuk be a párbeszédablak tulajdonságlapján a **Styles** fülben a Clip children tulajdonságot! Ennek hatására az ablak **WS_CLIPCHILDREN** stílusú lesz, így a gyermekablak(ai) (vezérlő) által takart felületet nem írja újra az ablak kirajzolásakor. Lásd még 6.4.1. szakasz Írás átlapolt gyermekablakokba.

5.2.4. Tároljuk a rajzot a memóriában!

Ha a rajzunk ablakát minimalizáljuk, majd újra megnyitjuk, vagy fölé mozgatunk egy másik ablakot, eltűnik a kép. Ahhoz, hogy vissza tudjuk állítani a rajzot, tárolnunk kell a memóriában.

Az egyik lehetőség a tárolásra, ha a memóriában létrehozunk egy DC-t és csatoljuk hozzá a képernyővel azonos méretű **bitmap**-et, mely a rajzunkat pixelenként tárolja. Újrafestéskor ezt a bitmap-et tesszük ki a képernyőre. Ha csak a kliensterülettel azonos méretű bitmap-et tárolunk, akkor maximalizálva az ablakot és rajzolva rá, az eredeti kliens méretén kívüli részek eltűnnének. (Próbáljuk ki ezt az esetet!)



5. Gyakorlat 13. ábra A 'Rajz' program osztályai

Először vegyük fel a két új adattagot, a CBitmap m_Bitmap és a CDC m_memDC-t!

Az m_memDC és az m_Bitmap inicializálása az OnInitDialog() tagfüggvényben, míg megszüntetésük a ***DestroyWindow()*** tagfüggvényben történik.

```
BOOL CRajzDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
 ...
 // TODO: Add extra initialization here
 m_newPen.CreatePen(PS_SOLID,10,RGB(100,0,100));
 //A maximalizált ablak kliensterületének mérete
 CRect rect(0,0,GetSystemMetrics(SM_CXFULLSCREEN),
                  GetSystemMetrics(SM_CYFULLSCREEN));
 CClientDC dc(this);
 m_Bitmap.CreateCompatibleBitmap(&dc,rect.Width(),
                                      rect.Height());
 m_memDC.CreateCompatibleDC(&dc);
 m_memDC.SelectObject(&m_Bitmap);
 // A memóriában is ez a toll rajzoljon!
 m_memDC.SelectObject(&m_newPen);
 // A háttér legyen szürke!
 CBrush brush;
 brush.CreateStockObject(LTGRAY_BRUSH);
 m_memDC.FillRect(&rect,&brush);

 return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

```
BOOL CRajzDlg::DestroyWindow()
{
 ReleaseDC(&m_memDC);
 m_newPen.DeleteObject();
 m_Bitmap.DeleteObject();
 return CDialog::DestroyWindow();
}
```

A kliensterületre mutató dc helyett most már az m_memDC-re rajzolunk az OnMouseMove-ban! És a WM_PAINT-tel tesszük a képernyőre.

```
void CRajzDlg::OnMouseMove(UINT nFlags, CPoint point)
{
 if ((nFlags & MK_LBUTTON) == MK_LBUTTON )
 {
//      CPen newPen(PS_SOLID,10,RGB(100,0,100));
    CPen* pOriginalPen;
    pOriginalPen=dc.SelectObject(&m_newPen);
//      dc.SetPixel(point, RGB(0,0,0));
    m_memDC.MoveTo(m_prevPoint);
    m_memDC.LineTo(point);
    Invalidate();
    m_prevPoint=point;
    dc. SelectObject(pOriginalPen);
//      newPen.DeleteObject();
 }

 CDialog::OnMouseMove(nFlags, point);
}
```

Frissítéskor a háttérből ki kell rajzolni az elmentett képet! Ez az ***OnPaint*** feladata! Az OnPaint metódus már szerepel a függvényeink között, de csak az ikont rajzolja ki a minimalizált esetben. Az if (IsIconic()) elágazásban már szerepel a **CPaintDC** és a kliens téglalp lekérdezése, mivel ezekre nekünk is szükségünk lesz, vegyük ki őket az elágazásból az elágazás elé!

A képet az 'else' ágba a ***BitBlt*** függvény segítségével rajzoljuk ki.

```
void CRajzDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 CRect rect;
 GetClientRect(&rect);
 if (IsIconic())
 {
    SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

    // Center icon in client rectangle
    int cxIcon = GetSystemMetrics(SM_CXICON);
    int cyIcon = GetSystemMetrics(SM_CYICON);
    int x = (rect.Width() - cxIcon + 1) / 2;
    int y = (rect.Height() - cyIcon + 1) / 2;
```

```
      // Draw the icon
      dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
      // A kép másolása a képernyőre.
      dc.BitBlt(0,0, rect.Width(), rect.Height(),&m_memDC,
                                                  0,0,SRCCOPY);
      CDialog::OnPaint();
    }
  }
```

A kép a teljes ablak gyakori frissítése miatt villog. A villogást megszüntethetjük, ha nem a teljes kliensterületet, hanem csak az (m_prevPoint, point) által megadott téglalapot frissítjük. Figyelembe véve a toll vastagságát az Invalidate()-ből a következő sorok lesznek:

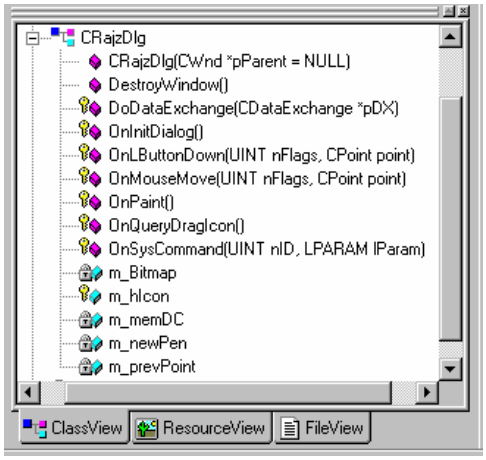
```
      m_memDC.MoveTo(m_prevPoint);
      m_memDC.LineTo(point);
      LOGPEN lgPen;
      m_newPen.GetLogPen(&lgPen);
      CSize penWidth(lgPen.lopnWidth.x,lgPen.lopnWidth.x);
      CRect rect(m_prevPoint-penWidth,point+penWidth);
      InvalidateRect(rect);
```

Mivel az újrafestés pihenőidős tevékenység, ha gyorsan húzzuk a vonalat, egyes részei nem frissülnek. Az m_memDC természetesen az egész rajzot tartalmazza. Ha nem akarjuk a villogást, vagy amikor a vonal húzását befejeztük kiadunk egy teljes ablakra vonatkozó frissítést (Az OnLButtonUp()-ban Invalidate()), vagy mégis kirajzoljuk a CClientDC dc-re is a képet.

Az ***OnMouseMove*** kódja tehát:

```
void CRajzDlg::OnMouseMove(UINT nFlags, CPoint point)
{
  if ((nFlags & MK_LBUTTON) == MK_LBUTTON )
  {
     SetCursor(m_hCursor);
     CClientDC dc(this);
//      CPen newPen(PS_SOLID,10,RGB(100,0,100));
     CPen* pOriginalPen=dc.SelectObject(&m_newPen);
//      dc.SetPixel(point, RGB(0,0,0));
     dc.MoveTo(m_prevPoint);
     dc.LineTo(point);
     m_memDC.MoveTo(m_prevPoint);
     m_memDC.LineTo(point);
     LOGPEN lgPen;
     m_newPen.GetLogPen(&lgPen);
     CSize penWidth(lgPen.lopnWidth.x,lgPen.lopnWidth.x);
     CRect rect(m_prevPoint-penWidth,point+penWidth);
     InvalidateRect(rect);
```

```
    m_prevPoint=point;
    dc. SelectObject(pOriginalPen);
//      newPen.DeleteObject();
 }
 CDialog::OnMouseMove(nFlags, point);
}
```



5. Gyakorlat 14. ábra A CRajzDlg osztály a ClassView-ban

5.2.5. Módosítsuk az egérkurzort!

- **Homokórakurzor**

A módosítás leggyakrabban használt esete: ha valami hosszantartó művelet közben jelzünk a felhasználónak, minden rendben van, csak dolgozik a program. Ezt a homokórakurzorral tehetjük meg a legkönnyebben. Szimuláljuk ezt az esetet egy várakoztatással! Ne készítsünk hozzá külön eseménykezelőt, hisz úgyis ki fogjuk törölni! Mondjuk az OnLButtonDown tagfüggvénybe írjuk be:

```
void CRajzDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
// TODO: Add your message handler code here and/or call default
 m_prevPoint=point;

 CWaitCursor wCursor;
 Sleep(5000);

 CDialog::OnLButtonDown(nFlags, point);
}
```

Ezután a kurzor minden bal egérgomb leütésnél 5 másodpercig várakozik, ezalatt megjeleníti a homokórakurzort, majd folytatja az üzenetek feldolgozását.

Tesztelés után töröljük ki e fölösleges sorokat a programból!

➢ Standard kurzor

Ha a rendszer által kínált kurzorokra szeretnénk váltani, válasszuk például a kereszt alakút, s a rajzolás alatt legyen ilyen a kurzorunk!

Vegyünk föl egy **HCURSOR** típusú m_hCursor nevű adattagot a CRajzDlg osztályunkban! A konstruktorban adjunk neki értéket!

```
m_hCursor = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
```

Majd az OnMouseMove tagfüggvényben hívjuk meg ezt a kurzort!

```
SetCursor(m_hCursor);
```

Ha azt akarjuk, hogy már az egérgomb lenyomása jelezze: "Rajzolni fogunk!", akkor az OnLButtonDown függvényben is állítsuk keresztre az egérkurzort!

Rajzolás közben villog a kurzorunk, mert a rendszer mindig visszaállítja az eredeti kurzort (nyíl). Ennek megakadályozására a **WM_SETCURSOR** üzenethez kezelőt rendelve felülírhatjuk az ***OnSetCursor*** metódust, hogyha a kliensterületen mozgatjuk az egeret, akkor ne hívja meg az ősosztály OnSetCursor metódusát, mely a kurzor visszaállítását végzi.

```
BOOL CRajzDlg::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT
message)
{
  // TODO: Add your message handler code here and/or call
default
  if (message == WM_MOUSEMOVE && nHitTest == HTCLIENT)
     return TRUE;
  return CDialog::OnSetCursor(pWnd, nHitTest, message);
}
```

Próbáljuk ki, hogyan módosul az alkalmazás, ha elhagyjuk a feltétel valamelyik ágát, vagy egyáltalán nem hívjuk meg a CDialog::OnSetCursor függvényét!

Ha az egérkurzor épp az ablak fölött van amikor indítjuk az alkalmazást, a homokórakurzor jelenik meg indulásképp. Ha az eredeti nyílkurzort akarjuk látni (amit akkor is látunk, ha egerünket az ablakon kívülről mozgatással hozunk be), akkor a konstruktorunkban kezdőértéket kell adni kurzorunknak.

```
CRajzDlg::CRajzDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CRajzDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CRajzDlg)
     // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent
DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
  m_hCursor=AfxGetApp()->LoadStandardCursor(IDC_CROSS);
  SetCursor(AfxGetApp()->LoadStandardCursor(IDC_ARROW));
}
```

➢ Saját kurzor használata

Ha saját kurzort szeretnénk használni, azt előbb el kell készítenünk az erőforrás-szerkesztővel.

Workspace

ResourceView fül

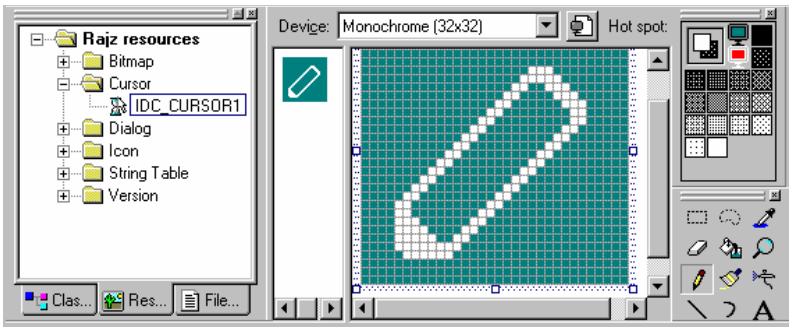
Rajz Resources jobb egérgomb

Insert

Cursor

New / Import (.cur) kiterjesztésűt válasszunk!

Ha a New-t választottuk, nekünk kell megrajzolnunk az új kurzort. Az erőforrások között megjelenik egy Cursor nevű, és generálódik egy IDC_CURSOR1 nevű új kurzor. Ezt kiválasztva a már ismerős erőforrás-szerkesztőben találjuk magunkat. Ide a vágólapon keresztül bármilyen ábrát beszúrhatunk, de mi is készíthetünk rajzot. Színként a zöld monitort választva a kurzorunk azon a helyen átlátszó lesz, a narancsszínű a háttértől elütő színt ad. Készítsünk egy rajzot! (pl. egy átlátszó ceruzát)



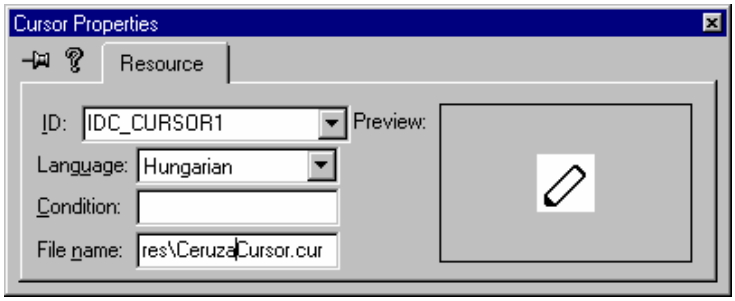
5. Gyakorlat 15. ábra A kurzor képének szerkesztése

Állítsuk a forró pontját a ceruza hegyére!

Legyen a mi kurzorunk a kiválasztott! A CRajzDlg osztály m_hCursor kurzorkezelőjéhez ezt a kurzort rendeljük. A CRajzDlg konstruktorába írjuk a következő sort, a standard kurzor betöltés helyére!

```
m_hCursor = AfxGetApp()->LoadCursor(IDC_CURSOR1);
```

A kurzorunkat leíró fájlt a projekt res alkönyvtárában Cursor1.cur néven találjuk. Ha a későbbi felhasználás érdekében "beszédesebb" nevet kívánunk adni neki, az IDC_CURSOR1 azonosítón jobb egér hatására legördülő menüből a Properties-t választva, átírhatjuk a fájl nevét.



5. Gyakorlat 16. ábra A kurzorfájl átnevezése

Természetesen az azonosítót is átnevezhettük volna, sőt még most is megtehetjük, de akkor mindenütt, ahol hivatkozunk rá (LoadCursor), ezt kell tennünk.

A konstruktorban feltöltött kezelőket szabadítsuk fel a destruktorban!

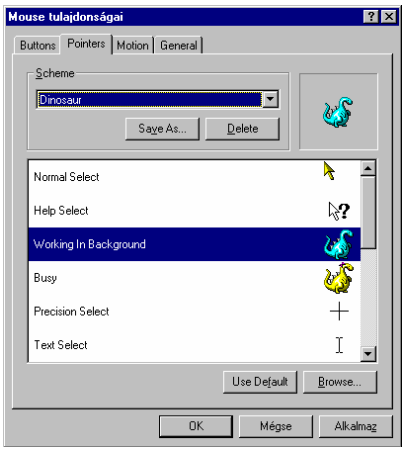
```
CRajzDlg::~CRajzDlg()
{
  DestroyIcon(m_hIcon);
  DestroyCursor(m_hCursor);
}
```

➢ Animált kurzor

Ha nem rajzolunk az egérrel, unatkozzon a programunk! Ennek jelzésére épp alkalmas egy animált kurzor.

Nézzük meg, milyen animált kurzoraink vannak! A Vezérlőpult / Egér / Mutatók fül / Séma ablakában a legördülő listából kiválaszthatjuk, és mozgás közben nézhetjük meg a rendelkezésünkre álló kurzorokat.

A kurzorfájlokat a Windows könyvtár alatti Cursors alkönyvtárban találjuk. A .cur kiterjesztésűek a hagyományos, míg a .ani kiterjesztésűek az animált kurzorok.

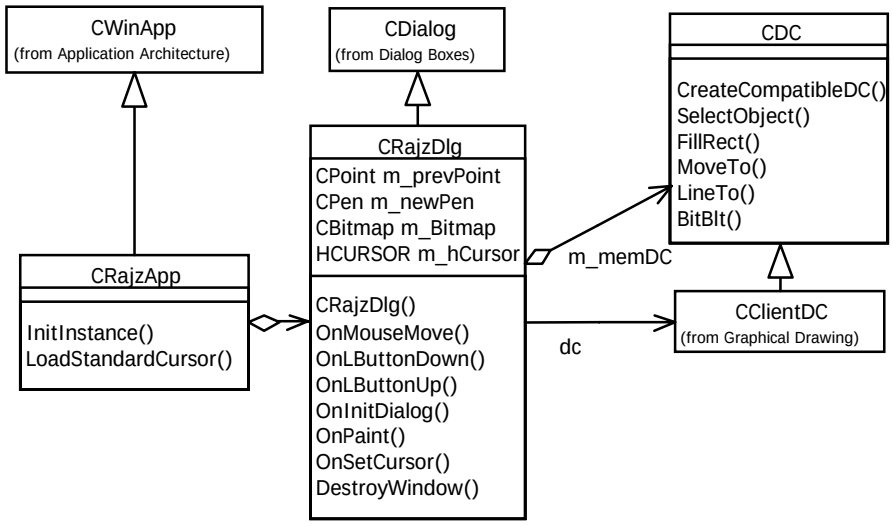


5. Gyakorlat 17. ábra Animált kurzorok működés közben

Mivel a rajzolást az egérgomb fölengedésével fejeztük be, ehhez az eseményhez rendeljük az animált kurzor indítását.

```
void CRajzDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
SetCursor(LoadCursorFromFile("E:\\WINNT\\Cursors\\banana.ani"));
  Invalidate();//Ha ezt választottuk.
  CDialog::OnLButtonUp(nFlags, point);
}
```

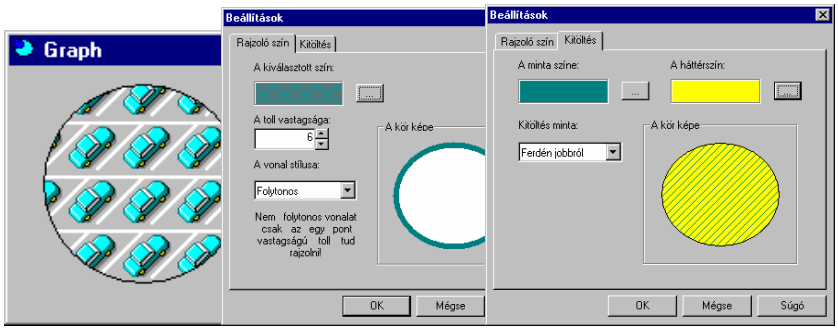
Természetesen a saját animált kurzorunk útvonalát írjuk a fájl neve elé!



5. Gyakorlat 18. ábra A 'Rajz' alkalmazás osztálydiagramja

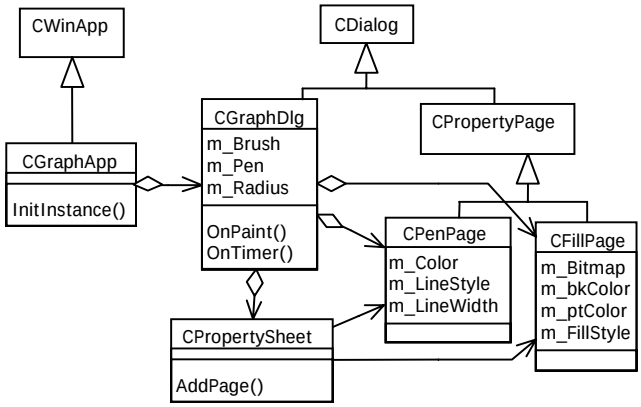
5.3. A toll és az ecset beállítása a párbeszédfelületen

Készítsünk egy alkalmazást, mely egy pulzáló kört rajzol ablakunkba! A kör mérete egy darabig növekszik, majd csökken. Legyen lehetőség a vonal színét, vastagságát és stílusát, valamint a kör belsejének kitöltő színét, mintáját beállítani!



5. Gyakorlat 19. ábra A kész alkalmazás futás közben

5.3.1. A terv



5. Gyakorlat 20. ábra A Graph alkalmazás kezdő terve

5.3.2. A rajz

Generáljunk Graph néven egy egyszerű párbeszédablakos alkalmazást! A párbeszédfelületre összesen egy 'Színek' feliratú gombot helyezzünk el! Bezárni, pl. az ablak bezáró ikonjával lehet az alkalmazást.

➢ Időzítő létrehozása

Időzítő (Timer) beállítása: CGraphDlg osztály inicializálása során történjen!

A **WM_INITDIALOG** üzenethez tartozó ***OnInitDialog()*** metódusban:

```
#define PULSE 1
//...
if (SetTimer(PULSE, 500, NULL) = = 0)
{ MessageBox( „Nem tudtam létrehozni az időzítőt”); }
```

Az időzítő leállítása az ablak lezárásakor szükséges.

A CGraphDlg osztály ***DestroyWindow()*** virtuális függvényét felülírjuk:

```
KillTimer(PULSE);
```

Ha az időzítő létrehozásakor a ***SetTimer()*** harmadik paramétere NULL volt, akkor CGraphDlg osztály **WM_TIMER** eseményéhez csatolt ***OnTimer()*** függvényt hívjuk meg az időzítő által megadott időközönként. Az időzítő tesztelése:

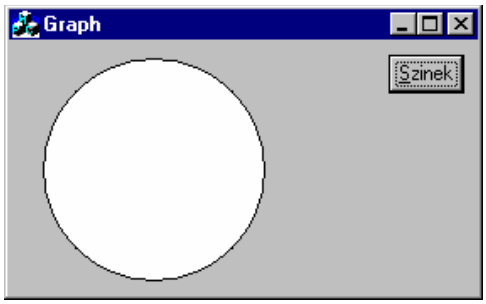
```
MessageBeep(MB_OK);
```

//Ezzel a módszerrel az időzítő sebességét érzékelhetjük.

➢ Rajzoljunk kört!

A kör kirajzolása a **WM_PAINT** eseményhez tartozó OnPaint() metódus feladata. Ezzel azt is elérhetjük, hogy az ablak újrarajzolásakor is megjelenjen a kép az ablakban. A **CPaintDC** objektum létrehozását most is vegyük ki az `IsIconic()` blokkból!

```
void CGraphDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 if (IsIconic())
 {
 //...
 }
 else
 {
 CDialog::OnPaint();
 CPen myNewPen;
 myNewPen.CreatePen(PS_SOLID,3, RGB(255,0,0));
 CPen* pOriginalPen=dc.SelectObject(&m_Pen);
 dc.Ellipse (20,10,120,110);
 dc.SelectObject(pOriginalPen);
 }
}
```



5. Gyakorlat 21. ábra Kört rajzol

➢ A kör mérete változzon!

Definiáljunk két változót a CGaphDlg osztályban:

```
int m_Radius;
int m_Direction;
```

Inicializáljuk őket az OnInitDialog-ban:

```
m_Radius=50;
m_Direction=1;
```

Kezeljük az időzítő eseményt az OnTimerben:

```
OnTimer( UINT nIDEvent)
{
 m_Radius += m_Direction;
 if (m_Radius>=100) {m_Direction= -1;}
 if (m_Radius<=3) {m_Direction= 1;}
 Invalidate();  // Az OnPaint() függvényhívás kikényszerítése.
}
```

Az OnPaint függvényben a téglalap koordinátáit javítsuk ki!

```
dc.Ellipse(20,10, 20+m_Radius*2, 10+m_Radius*2);
```

➢ A villogás megszüntetése

Ha Invalidate()-tel az egész ablak újrafestését kikényszerítjük, az ablakot először letörli a CDialog::OnPaint() hívás, majd kirajzolja a képet a gombbal együtt. Ezt a helyzetet jól megfigyelhetjük, ha az időzítőnket 2 másodpercenkénti üzenetküldésre állítjuk. De vele együtt nagyobb legyen a növekedés üteme is!

A párbeszédablak OnInitDialog-ban:

```
if (SetTimer(PULSE, 500, NULL) = = 0)
{ MessageBox( „Nem tudtam létrehozni az időzítőt”); }
m_Direction=10;
```

Majd az OnPaint-ben:

```
CDialog::OnPaint();
::Sleep(1000);
```

A tesztelés után állítsuk vissza az eredeti állapotot!

A villogás csökkenthető, ha nem rajzoljuk újra a teljes ablakot, csak a kívánt részt.

De a legjobb megoldás, ha a kör rajzolását és az előző törlését az OnTimer függvényünkre bízzuk. (Természetesen az OnPaint()-ben is ott maradhat a kód, ez a különösen ritka időzítőüzenet beállítás és a képernyő újrafestés együttes bekövetkezése esetén lehet hasznos.)

A téglalap belsejének újrafestéséhez válasszunk egy üres tollat, így a határvonalat nem rajzoljuk ki. Mivel ezt a tollat minden OnTimer híváskor használjuk, vegyük föl az osztály adatai között! Majd az OnInitDialogban kapjon kezdőértéket!

```
 m_nullPen.CreateStockObject(NULL_PEN);
```

Az új OnTimer függvényünk tehát:

```
void CGraphDlg::OnTimer(UINT nIDEvent)
{
 // TODO: Add your message handler code here and/or call
default
 //MessageBeep(MB_OK);
 CClientDC dc(this);
 CPen* pOriginalPen;
 // Háttér törlése.
 // Ne rajzoljon vonalat!
 pOriginalPen=dc.SelectObject(&m_nullPen);
 CRect myRectangle(20,10,20+m_Radius*2,10+m_Radius*2);
 CBrush brush; // Háttér színű ecset.
 brush.CreateStockObject(LTGRAY_BRUSH);
 dc.FillRect(&myRectangle, &brush);

 // Az új méret beállítása
 m_Radius = m_Radius + m_Direction;

 if (m_Radius>=100) { m_Direction = -1;}
 if (m_Radius<=3)  { m_Direction =  1;}
//  Invalidate();// A WM_PAINT kikényszerítése.
```

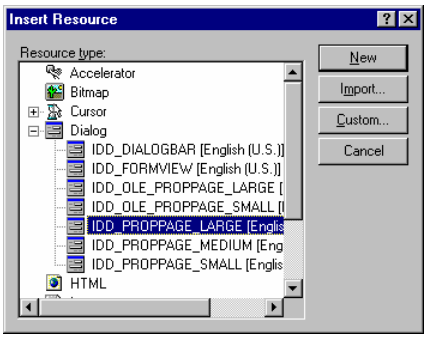
```
// Az új kör kirajzolása.
CPen myNewPen;
myNewPen.CreatePen(PS_SOLID,3, RGB(255,0,0));
dc.SelectObject(&myNewPen);
 myRectangle=CRect(20,10,20+m_Radius*2,10+m_Radius*2);
 dc.Ellipse (&myRectangle);
 dc.SelectObject(pOriginalPen);

 CDialog::OnTimer(nIDEvent);
}
```

5.3.3. A toll és az ecset tulajdonságlapja

A színek nyomógomb hatására egy kétfülű tulajdonságablak megnyitásával szeretnénk kényelmessé tenni a rajzolóobjektumok beállítását. Az egyik fül a toll, míg a másik az ecset beállításait támogatja. A feladatot property page és property sheet segítségével oldjuk meg.

➢ Az üres párbeszédfelületek és meghívásuk

Az erőforrás-szerkesztőben vegyünk fel egy új Dialog erőforrást, melynek az alapja egy IDD_PROPPAGE_LARGE erőforrás lesz, amit az Insert Resource ablakban választhatunk ki! (ResorceView / Dialog / Jobb egérgomb / Insert.)



5. Gyakorlat 22. ábra A property page erőforrás beszúrása

Nevezzük ezt az erőforrást **IDD_PROPPAGE**_PEN-nek! Majd ismételjük meg az azonos méretű (LARGE) tulajdonságlap beszúrásával IDD_PROPPAGE_FILL néven!

Rendeljünk **CPropertyPage** utódosztályokat CPenPage és CFillPage néven hozzájuk! (CtrlW / Adding Class / Create a new class / OK / Name: CPenPage / Base Class: CPropertyPage / Dialog ID: IDD_PROPPAGE_PEN)

Vegyük fel az objektumokat a CGraphDlg osztályban m_penPage és m_fillPage néven! (CGraphDlg / Jobb egérgomb / Add Member Variable / Variable Type:

CPenPage / Variable name: m_penPage / Private.) Vegyünk fel egy CPropertySheet típusú m_szinek nevű adattagot is az osztályban!

A CGraphDlg::OnInitDialog metódusban fűzzük föl a tulajdonságlapokat a **CPropertySheet** osztály ***AddPage()*** tagfüggvénye segítségével!

```
BOOL CGraphDlg::OnInitDialog()
{
// ...
 // A tulajdonságlapok felfűzése.
 m_szinek.SetTitle("Beállítások");//A cím beállítása.
 m_szinek.AddPage(&m_penPage);
 m_szinek.AddPage(&m_fillPage);
 return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

A párbeszédfelületet a Színek gomb választás hatására jelenítjük meg. Tehát a gomb megnyomásához rendelt eseménykezelőben nyissuk ki ***DoModal()***-lal a CPropertySheet ablakot!

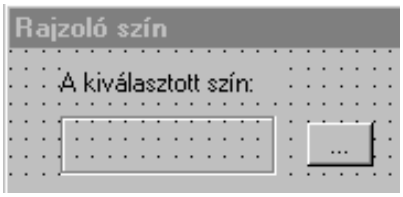
```
void CGraphDlg::OnSzinekButton()
{
 m_szinek.DoModal();
}
```

Itt lehet majd a párbeszédfelület bezárása után lekérdezni a beállított értékeket. Ez a rész egyenlőre azért kellett, hogy futás közben tesztelni tudjuk a tulajdonságlapok vezérlőit. Ha ezt nem írtuk volna meg, a tulajdonságlapok futtatáskor nem jelennének meg a képernyőn.

5.3.4. A toll beállításai

➢ A toll színe

Először is adjunk az IDD_PROPPAGE_PEN tulajdonságlapnak címet! A property page Caption mezőjébe írjuk be: Rajzoló szín.



5. Gyakorlat 23. ábra A szín vezérlők az erőforrás-szerkesztőben

A színt, amikor kirajzoljuk a vezérlőre, szeretnénk egy keretben látni a rajzot, ezért "A kiválasztott szín:" szöveg alá egy szöveg nélküli csoportablak vezérlőt teszünk IDC_SZIN_STATIC néven. Hozzárendelhetünk egy **CStatic** típusú m_Frame nevű változót. (Ha az osztályvarázsló nem engedi a hozzárendelést, a változó típusát módosíthatjuk kézzel is, de mivel csak a vezérlő helyének lekérdezésére használjuk más osztály objektuma is megfelelő.) S mellé elhelyezünk egy a szín kiválasztására szolgáló három pontot tartalmazó nyomógombot, IDC_CSERE_BUTTON néven.

A színt a következő megjelenítésig tárolni is kell, ezért vegyünk fel a CPenPage osztályban egy m_Color **COLORREF** típusú public adattagot! Ez kezdetben legyen 0! Mivel az m_Color-t a főablak körének kirajzolásakor is látni kell, ezért legyen public láthatóságú! (Vagy írjunk egy public GetColor tagfüggvényt hozzá!)

```
CPenPage::CPenPage() : CPropertyPage(CPenPage::IDD)
{
 m_Color=0;
}
```

Ha a színt kezdéskor is és az ablak frissítésekor is ki akarjuk rajzolni, célszerű azt az osztály OnPaint metódusába tenni.

```
void CPenPage::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 // A szín kirajzolása.
 CRect rect;
 CBrush brush;
 brush.CreateSolidBrush(m_Color);
 m_Frame.GetWindowRect(&rect);
 ScreenToClient(&rect);
 rect.top+=6; // A fejléc miatt
 dc.FillRect(&rect,&brush);
 // Do not call CPropertyPage::OnPaint() for painting messages
}
```

A három pontot tartalmazó nyomógomb a szín cseréjét végzi a **CColorDialog** segítségével. Ehhez egy CColorDialog m_ColorDlg adattagra lesz szükségünk a CPenPage osztályunkban. A gomb kiválasztása egy m_ColorDlg.DoModal hívást vált ki. A színek párbeszédablak bezárása után le lehet kérdezni az osztályban beállított színt és betenni az m_Color adattagba. Ahhoz, hogy az újrafestés bekövetkezzen, meg kell hívnunk az ***Invalidate()*** metódust is.

```
void CPenPage::OnCsereButton()
{
 if (m_ColorDlg.DoModal())
 {
    m_Color=m_ColorDlg.GetColor();
    Invalidate(); // Fesse újra az ablakot!
 }
}
```

Nézzük meg a Custom Colors részben a színt kevert színként és tiszta színként hogyan látjuk majd!

➢ Rajzoljuk ki a kört a kiválasztott színnel!

Ehhez vegyünk fel egy újabb csoportablakot, immár "A kör képe" szöveggel és IDC_RAJZ_STATIC egyedi azonosítóval! A kör kirajzolását a vezérlők adatainak beállításához köthetjük, de minden kör azonos méretű lesz. Ehhez vegyünk fel egy m_rajzRect nevű adattagot a CPenPage osztályban, s az OnInitDialog-ban inicializáljuk a keret adatai segítségével!

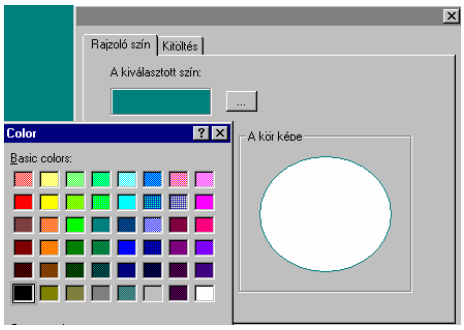
```
BOOL CPenPage::OnInitDialog()
{
 CPropertyPage::OnInitDialog();
 GetDlgItem(IDC_RAJZ_STATIC)->GetWindowRect(&m_rajzRect);
 ScreenToClient(&m_rajzRect);
 // A címsorral lejjebb.
 m_rajzRect.top+=26;
 m_rajzRect.left+=20;
 m_rajzRect.bottom-=20;
 m_rajzRect.right-=20;

 return TRUE;  // return TRUE unless you set the focus to a
//control EXCEPTION: OCX Property Pages should return FALSE
}
```

A 20 egység méretcsökkentés azért kell, hogy a toll vastagságának növelése közben se lógjon ki a kör képe a keretből. A rajzoláskor mindig elkészíthettük volna a téglalap méretét, de fölösleges minden rajzoláskor újra lekérdezni és kiszámolni a méretet! Mivel a rajzolást több helyről is hívni fogjuk, készítsünk e célra egy Rajz nevű private tagfüggvényt! (CPenPage / Jobb egérgomb / Add Member Function...)

```
void CPenPage::Rajz()
{
 CClientDC dc(this);
 // Először letöröljük az előző kört
 CBrush brush;
 brush.CreateStockObject(LTGRAY_BRUSH);
 CRect rect;
 // A törlés téglalapmérete a toll vastagsága miatt nagyobb.
 rect.left=m_rajzRect.left-15;
 rect.top=m_rajzRect.top-15;
 rect.right=m_rajzRect.right+15;
 rect.bottom=m_rajzRect.bottom+15;
 dc.FillRect(&rect, &brush);
```

```
    CPen pen, *pOriginalPen;
    // Ezt írjuk majd át a többi paraméter beállításakor!
    pen.CreatePen(PS_SOLID, 1, m_Color);
    pOriginalPen=dc.SelectObject(&pen);
    dc.Ellipse(&m_rajzRect);
    dc.SelectObject(pOriginalPen);
    pen.DeleteObject();
}
```

A Rajz tagfüggvényt hívjuk meg az OnPaint-ből! Az OnCsereButton-ból is meg kellene hívni, de az Invalidate() az OnPaint-et, az pedig a Rajz metódust úgyis meghívja.



5. Gyakorlat 24. ábra A szín vezérlők az erőforrás-szerkesztőben

➢ A vonalvastagság beállítása

A tollszélesség beállításához egy szerkesztőmezőt (IDC_LINEWIDTH_EDIT) jobbra rendezett szöveggel és egy léptetőgombot (spin) (IDC_LINEWIDTH_SPIN) használunk.

5. Gyakorlat 25. ábra A tollszélesség vezérlői az erőforrás-szerkesztőben, és futásidőben

A szövegmezőt rendezzük a kívánt szélességűre, a jobbról mellé helyezett léptetőgomb annak jobb szélét fogja elfoglalni (5. Gyakorlat 25. ábra). A léptetőgomb tulajdonságlapján a Styles fülben állítsuk az Alignment értékét Right-re! Most határoztuk meg, hogy léptetőgombunk a hozzá tartozó ablak jobb oldalán fog elhelyezkedni. Az 'Auto buddy' és a 'Set buddy integer' stílust is válasszuk ki! Így elértük, hogy a splitter a szövegmező ablakához kapcsolódik és a benne tárolt szám értékét változtatja. A szövegmezőhöz csatolást a Test gomb segítségével ellenőrizhetjük (5. Gyakorlat 25. ábra).

Ha nem a megfelelő mezőhöz csatoltuk, a tabulátorsorrenden kell változtatnunk (Layout / Tab Order)! Ha a tabulátorsorrendben a kívánt ablak előzi meg a léptetőgombot, a munkaablak automatikusan szándékunk szerint csatolódik.

A szövegmezőhöz rendeljük hozzá az m_LineWidth integer, public adattagot! Értékhatárait állítsuk 0 és 30 közé! Kezdőértékét a konstruktorban írjuk át 1-re!

```
CPenPage::CPenPage() : CPropertyPage(CPenPage::IDD)
{
 //{{AFX_DATA_INIT(CPenPage)
 m_LineWidth = 1;
 //}}AFX_DATA_INIT
 m_Color=0;
}
```

Ha megfigyeljük az értékek változását, láthatjuk hogy a léptetőgomb alapértelmezésének megfelelően felfelé nyílra csökken míg lefelé nyíl választásakor növekszik az érték. Az értékhatárok 0 és 100 között változnak. Az értékhatárok módosításához használjuk a SetRange() függvényt!

```
BOOL CPenPage::OnInitDialog()
{...
 CSpinButtonCtrl* pSpin =
              (CSpinButtonCtrl*)GetDlgItem(IDC_LINEWIDTH_SPIN);
 pSpin->SetRange32(0,30);
...
}
```

A kör képe még nem jelenik meg 'A kör képe' ablakban! Ehhez a szövegmező **EN_CHANGE** üzenetét kell kezelnünk. A feltételt azért kell a kódba betennünk, hogy kezdetben (és később is) üres szövegmezőből ne próbáljon számot kiolvasni az UpdateData()

```
void CPenPage::OnChangeLinewidthEdit()
{
 CString strLineWidth;
 GetDlgItem(IDC_LINEWIDTH_EDIT)->GetWindowText(strLineWidth);
 if (strLineWidth!="")      //Kezdetben ne fagyjon le!
 {
    UpdateData();
    Rajz();
 }
```

A Rajz függvényben vegyük figyelembe a toll szélességét!

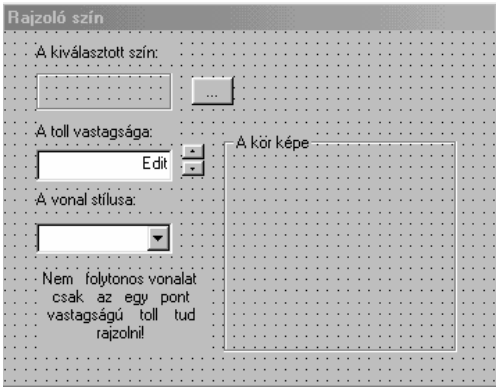
```
void CPenPage::Rajz()
{
 //...
 dc.FillRect(&rect, &brush);
 if (m_LineWidth) // Nincs toll, nem rajzolunk.
 {
    CPen pen, *pOriginalPen;
    pen.CreatePen(PS_SOLID, m_LineWidth, m_Color);
    pOriginalPen=dc.SelectObject(&pen);
    dc.Ellipse(&m_rajzRect);
    dc.SelectObject(pOriginalPen);
    pen.DeleteObject();
 }
}
```

Az OnChangeLinewidthEdit() függvény egyben a billentyűzetről begépelt számok esetén is meghívódik. A gond csak az, hogy bár ellenőrzi az osztályvarázslóban az m_LineWidth-hez beállított értékhatárt, és nem fogadja el a rajta kívül eső számokat, a rajz mégis elkészül pl. 100 szélességű tollal is. Tehát újabb feltételre van szükségünk!

```
void CPenPage::OnChangeLinewidthEdit()
{
 CString strLineWidth;
 GetDlgItem(IDC_LINEWIDTH_EDIT)->GetWindowText(strLineWidth);
 if (strLineWidth!="")      //Kezdetben ne fagyjon le!
 {
    UpdateData();
    if ((m_LineWidth>=0) && (m_LineWidth<31))
    {
        Rajz();
    }
 }
}
```

➢ A toll stílusa

Egy egyszerű kombipanellel beállíthatjuk a toll stílusát.



5. Gyakorlat 26. ábra A toll tulajdonságlapja az erőforrás-szerkesztőben

A **kombipanel** adatai közé írjuk fel sorba a toll stílusait, s rendeljünk integer típusú tagváltozót a kombipanelhez (m_LineStyle)! (Ha szükséges kézzel!) A konstruktorban -1 kezdőértéket találunk, aminek hatására nem jelenik meg stílus a körhöz. Írjuk át ezt 0-ra, ami a folytonos vonal stílusát jelenti!

```
CPenPage::CPenPage() : CPropertyPage(CPenPage::IDD)
{
 //{{AFX_DATA_INIT(CPenPage)
 m_LineWidth = 1;
 m_LineStyle = 0;//Folytonos vonal
 //}}AFX_DATA_INIT
 m_Color=0;
}
```

A felhasználó által kiválasztott értéket a **CBN_SELCHANGE** üzenetet kezelve olvashatjuk be.

```
void CPenPage::OnSelchangeLinestyleCombo()
{
 UpdateData();
 Rajz();
}
```

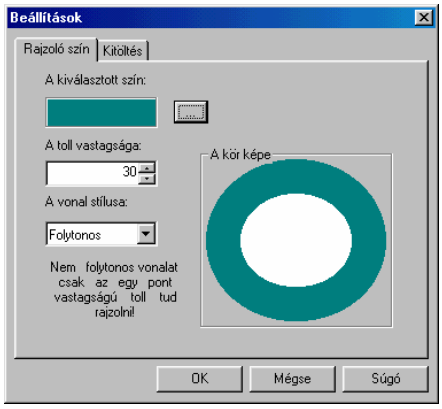
A Rajz metódusban a stílust is vegyük figyelembe!

```
pen.CreatePen(m_LineStyle, m_LineWidth,m_Color);
```

Mivel egynél nagyobb tollszélesség esetén mindenképp folytonos vonalat kapunk, erre figyelmeztessük alkalmazásunk felhasználóját is!

Megtehetnénk, hogy egynél vastagabb toll esetén nem engedünk stílust váltani, de akkor az üres toll és a kitöltő színű toll sem lenne kiválasztható.

! Figyeljük meg, hogy bizonyos színek választása esetén jelentős eltérés van a kiválasztott szín és a rajz színe között. A magyarázat, hogy a toll tiszta színnel rajzol, míg az ecset kevert színt használ. Ha a toll színét kitöltő szín stílusra választjuk, az eltérés megszűnik.



5. Gyakorlat 27. ábra A toll tulajdonságlapja működés közben

➢ A főablak rajzának módosítása a tollbeállítások figyelembevételével

Mivel többször rajzoljuk ki azonos rajzobjektumok segítségével a köröket, célszerű lenne ezeket nem minden alkalommal létrehozni, hanem tárolni kellene az osztályban.

Vegyünk fel a CGraphDlg osztályban két **CBrush** objektumot, m_Brush és m_bkBrush néven a rajzoláshoz és a törléshez, és két **CPen** objektumot m_Pen és m_nullPen néven! Inicializáljuk őket az OnInitDialog-ban!

```
BOOL CGraphDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
//...
 m_Pen.CreateStockObject(BLACK_PEN);
 m_Brush.CreateStockObject(WHITE_BRUSH);
 m_bkBrush.CreateStockObject(LTGRAY_BRUSH);
 m_nullPen.CreateStockObject(NULL_PEN);
```

```
 // A tulajdonságlapok felfűzése.
 m_szinek.SetTitle("Beállítások");
 m_szinek.m_psh.dwFlags |= PSH_NOAPPLYNOW;
 m_szinek.AddPage(&m_penPage);
 m_szinek.AddPage(&m_fillPage);
 return TRUE;  // return TRUE  unless you set the focus to a
control
}
```

Az objektumokat a DestroyWindow()-ban töröljük.

```
BOOL CGraphDlg::DestroyWindow()
{
 KillTimer(PULSE);
 m_bkBrush.DeleteObject();
 m_Brush.DeleteObject();
 m_Pen.DeleteObject();
 m_nullPen.DeleteObject();

 return CDialog::DestroyWindow();
}
```

Az OnPaint-ben ezeket az objektumokat használva rajzoljuk ki a kört!

```
void CGraphDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 //...
 else
 {
    CDialog::OnPaint();
    CPen* pOriginalPen;
    pOriginalPen=dc.SelectObject(&m_Pen);
    CRect MyRectangle(20,10,20+m_Radius*2,10+m_Radius*2);
    dc.Ellipse (&MyRectangle);
    dc.SelectObject(pOriginalPen);
 }
}
```

Az OnTimer-ben ezekkel töröljük le az előző kört és rajzoljuk ki az újat!

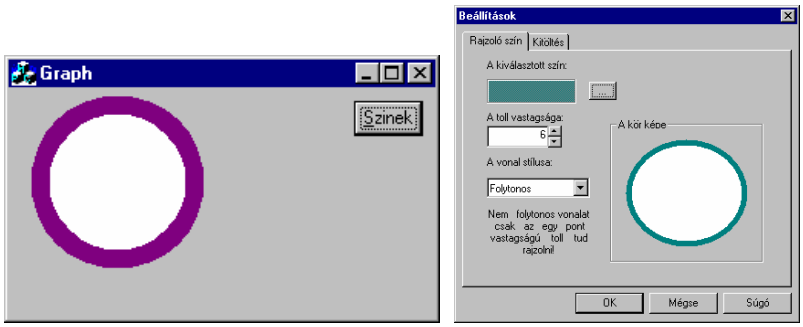
```
void CGraphDlg::OnTimer(UINT nIDEvent)
{
 // TODO: Add your message handler code here and/or call
default
 //MessageBeep(MB_OK);
 CClientDC dc(this);
 CPen* pOriginalPen;
 // A háttér törlése.
 //Ne rajzoljon vonalat!
 pOriginalPen=dc.SelectObject(&m_nullPen);
 // A toll vastagságával növelni kell a törölt téglalapot!
 CRect myRectangle(20,10,20+m_Radius*2,10+m_Radius*2);
 dc.FillRect(&myRectangle, &m_bkBrush);
```

```
 // A méret változtatása.
 m_Radius = m_Radius + m_Direction;

 if (m_Radius>=100) { m_Direction = -1;}
 if (m_Radius<=3)  { m_Direction =  1;}

//  Invalidate();// A WM_PAINT kikényszerítése.
 // Az új kör kirajzolása.
 dc.SelectObject(&m_Pen);
 CBrush* pOriginalBrush;
 pOriginalBrush=dc.SelectObject(&m_Brush);
 myRectangle=CRect(20,10,20+m_Radius*2,10+m_Radius*2);
 dc.Ellipse (&myRectangle);
 dc.SelectObject(pOriginalPen);
 dc.SelectObject(pOriginalBrush);

 CDialog::OnTimer(nIDEvent);
}
```

Az OnSzinekButton-ban, ha OK-ra lép ki, akkor a rajzolótollat (később az ecsetet is) módosítni kell a beállításoknak megfelelően. A módosítás azt jelenti, hogy töröljük az előző tollat és létrehozunk egy másikat.

```
void CGraphDlg::OnSzinekButton()
{
 if (m_szinek.DoModal()==IDOK)
 {
    m_Pen.DeleteObject();
    m_Pen.CreatePen(m_penPage.m_LineStyle,
                    m_penPage.m_LineWidth,m_penPage.m_Color);
 }
}
```

Ha a színek beállítása ablakot elhúzzuk a kép elől, jól látható, hogy a rajzolás a Szinek ablak aktívvá tétele alatt is folytatódik, a szín váltása azonban csak az OK gomb leütése után valósul meg. Ez természetes, hisz a CGraphDlg osztályban adatok frissítése csak ekkor történik meg.

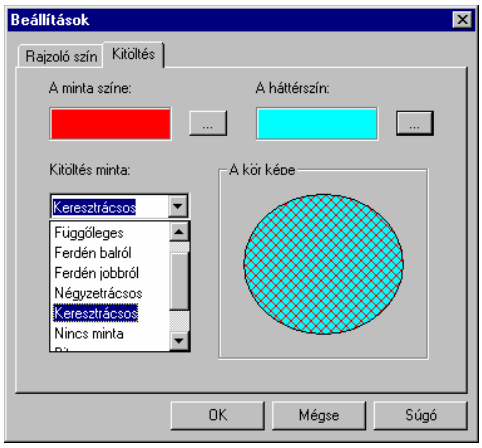


5. Gyakorlat 28. ábra A 'Graph' alkalmazás működés közben

5.3.5. Az ecset tulajdonságlapja

➢ A felület

Az ecset beállításait az előzőek alapján az olvasóra bízzuk. Itt csak egy lehetséges kódot közlünk. A megoldás során mind az erőforrás készítésekor, mind pedig a kód írásakor használhatjuk a már meglevő kódunk részleteit.



5. Gyakorlat 29. ábra A kitöltés tulajdonságlap

A tulajdonságlap erőforrásfelülete a tolléhoz hasonló, a kombipanel szövegei a következők: Vízszintes, Függőleges, Ferdén balról, Ferdén jobbról, Négyzetrácsos, Keresztrácsos, Nincs minta, Bitmap, Üres ecset. Ennek azért van jelentősége, mert a CreateHatchBrush függvény első paramétere az ecset stílusának felel meg, s a kombipanel által kiválasztott sor szövege így lesz azonos a megadott stílussal. A 0. helyen a 'Vízszintes' szöveg áll, s a **HS_HORIZONTAL**=0.

➢ A CFillPage osztály

A CFillPage osztály deklarációja:

```
class CFillPage : public CPropertyPage
{
 DECLARE_DYNCREATE(CFillPage)

// Construction
public:
 CBitmap m_Bitmap;
 void Rajz();
 COLORREF m_bkColor;// Háttérszín.
 COLORREF m_ptColor;// Minta színe.
 CFillPage();
 ~CFillPage();

// Dialog Data
 //{{AFX_DATA(CFillPage)
 enum { IDD = IDD_PROPPAGE_FILL };
 CButton      m_Frame;
 CButton      m_ptFrame;
 int          m_FillStyle;
 //}}AFX_DATA

//...
// Implementation
protected:
 // Generated message map functions
 //{{AFX_MSG(CFillPage)
 afx_msg void OnSzincsereButton();
 afx_msg void OnPtSzincsereButton();
 virtual BOOL OnInitDialog();
 afx_msg void OnPaint();
 afx_msg void OnSelchangeComboboxex();
 //}}AFX_MSG
 DECLARE_MESSAGE_MAP()

private:
 CRect m_rajzRect;
 CColorDialog m_ColorDlg;
};
```

A függvények

```
CFillPage::CFillPage() : CPropertyPage(CFillPage::IDD)
{
 //{{AFX_DATA_INIT(CFillPage)
 m_FillStyle = 6;
 //}}AFX_DATA_INIT
 m_bkColor=RGB(255,255,255);
 m_ptColor=RGB(192,192,192);
}
```

```
CFillPage::~CFillPage()
{
 m_Bitmap.DeleteObject();
}

void CFillPage::DoDataExchange(CDataExchange* pDX)
{
 CPropertyPage::DoDataExchange(pDX);
 //{{AFX_DATA_MAP(CFillPage)
 DDX_Control(pDX, IDC_BKCOLOR_STATIC, m_Frame);
 DDX_Control(pDX, IDC_PTCOLOR_STATIC, m_ptFrame);
 DDX_CBIndex(pDX, IDC_COMBOBOXEX, m_FillStyle);
 //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CFillPage, CPropertyPage)
 //{{AFX_MSG_MAP(CFillPage)
 ON_BN_CLICKED(IDC_SZINCSERE_BUTTON, OnSzincsereButton)
 ON_BN_CLICKED(IDC_PTSZINCSERE_BUTTON, OnPtSzincsereButton)
 ON_WM_PAINT()
 ON_CBN_SELCHANGE(IDC_COMBOBOXEX, OnSelchangeComboboxex)
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

➢ A színek és a stílus módosítása

Ha az ecset színét módosítottuk:

```
void CFillPage::OnSzincsereButton()
{
 if (m_ColorDlg.DoModal())
 {
    m_bkColor=m_ColorDlg.GetColor();
    Invalidate(); // Fesse újra az ablakot!
 }
}

void CFillPage::OnPtSzincsereButton()
{
 if (m_ColorDlg.DoModal())
 {
    m_ptColor=m_ColorDlg.GetColor();
    Invalidate(); // Fesse újra az ablakot!
 }
}
```

Megjegyzés:

A mintás (hatch) ecset színeinél a minta színe (m_ptColor) ugyanúgy viselkedik, mint a toll színe. Tehát a beállítástól függően a legközelebbi **tiszta szín**nel rajzol.

Ha elemet választottunk a kombipanelből:

```
void CFillPage::OnSelchangeComboboxex()
{
 UpdateData();
 Rajz();
}
```

➢ Ábra a tulajdonságlapon

A rajzolótéglalap beállítása:

```
BOOL CFillPage::OnInitDialog()
{
 CPropertyPage::OnInitDialog();

 GetDlgItem(IDC_FILLRAJZ_STATIC)->GetWindowRect(&m_rajzRect);
 ScreenToClient(&m_rajzRect);
 // A címsorral lejjebb.
 m_rajzRect.top+=26;
 m_rajzRect.left+=20;
 m_rajzRect.bottom-=20;
 m_rajzRect.right-=20;

 return TRUE;  // return TRUE unless you set the focus to a
control
               // EXCEPTION: OCX Property Pages should return
FALSE
}
```

Ha az ablakot újra kell festeni:

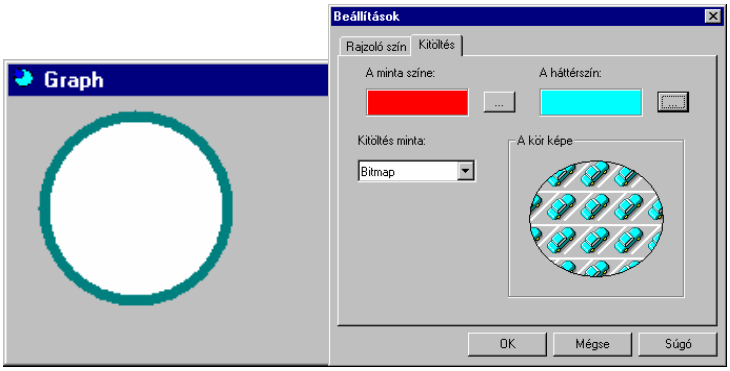
```
void CFillPage::OnPaint()
{
 CPaintDC dc(this); // device context for painting

 // A szín kirajzolása.
 CRect rect, ptRect;
 CBrush brush, ptBrush;
 // A háttérszín kirajzolása
 brush.CreateSolidBrush(m_bkColor);
 m_Frame.GetWindowRect(&rect);
 ScreenToClient(&rect);
 rect.top+=6; // A fejléc miatt
 dc.FillRect(&rect,&brush);
```

```
 // A minta színének kirajzolása
 ptBrush.CreateSolidBrush(m_ptColor);
 m_ptFrame.GetWindowRect(&ptRrect);
 ScreenToClient(&ptRect);
 ptRect.top+=6; // A fejléc miatt
 dc.FillRect(&ptRect,&ptBrush);
 // A háttéradatok frissítése.
 UpdateData(FALSE);
 // A rajz elkészítése.
 Rajz();

 // Do not call CPropertyPage::OnPaint() for painting messages
}
```

A keretbe kirajzoljuk a beállításnak megfelelő kört:

```
void CFillPage::Rajz()
{
 CClientDC dc(this);
 // Először letöröljük az előző kört.
 CBrush brush;
 brush.CreateStockObject(LTGRAY_BRUSH);
 CRect rect;
 // A törlés téglalapmérete a toll vastagsága miatt nagyobb.
 rect.left=m_rajzRect.left-15;
 rect.top=m_rajzRect.top-15;
 rect.right=m_rajzRect.right+15;
 rect.bottom=m_rajzRect.bottom+15;
 dc.FillRect(&rect, &brush);
 // Az ecset kiválasztása
 CBrush *pOriginalBrush;
 brush.DeleteObject();
 switch (m_FillStyle)
 {
    case 6:   brush.CreateSolidBrush(m_bkColor); break;
    case 7:   m_Bitmap.DeleteObject();
              m_Bitmap.LoadBitmap(IDB_BITMAP1);
              brush.CreatePatternBrush(&m_Bitmap);
              break;
    case 8:   brush.CreateStockObject(NULL_BRUSH);break;
    default:  brush.CreateHatchBrush(m_FillStyle, m_ptColor);
                                                    break;
 }
    pOriginalBrush=dc.SelectObject(&brush);
    COLORREF oldColor=dc.SetBkColor(m_bkColor);

    dc.Ellipse(&m_rajzRect);
    dc.SelectObject(pOriginalBrush);
    dc.SetBkColor(oldColor);
    brush.DeleteObject();
}
```

A 'Kitöltés' ablak már jól működik, de a főablakban is a választott ecsetet kell használnunk!



5. Gyakorlat 30. ábra A 'Kitöltés' ablak már működik

➢ A főablak is vegye figyelembe az ecsetválasztást!

Az új ecsetet a 'Beállítások' ablak bezárása után készítjük el.

```
void CGraphDlg::OnSzinekButton()
{
  if (m_szinek.DoModal()==IDOK)
  {
     m_Pen.DeleteObject();
     m_Pen.CreatePen(m_penPage.m_LineStyle,
                     m_penPage.m_LineWidth,m_penPage.m_Color);
     m_Brush.DeleteObject();
     switch (m_fillPage.m_FillStyle)
     {
         case 6:m_Brush.CreateSolidBrush(m_fillPage.m_bkColor);
                break;
         case 7:m_Brush.CreatePatternBrush(&m_fillPage.m_Bitmap);
                break;
         case 8:m_Brush.CreateStockObject(NULL_BRUSH);
                break;
         default:m_Brush.CreateHatchBrush(m_fillPage.m_FillStyle,
                     m_fillPage.m_ptColor); break;
     }
  }
}
```

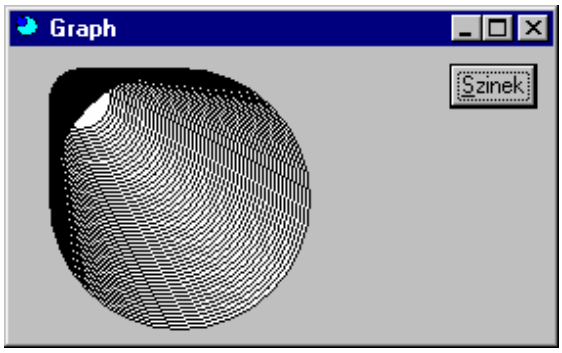
Próbáljuk ki, hogy ne töröljük le az előző képet, ha üres ecsetet választunk! Ehhez a CGraphDlg osztály OnPaint és OnTimer metódusát kell módosítanunk.

```
void CGraphDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 if (IsIconic())
 {
    //...
 }
 else
 {
    if (!(m_fillPage.m_FillStyle==8))
    {
        CDialog::OnPaint();
    }

    CPen* pOriginalPen;
    pOriginalPen=dc.SelectObject(&m_Pen);
    COLORREF oldColor=dc.SetBkColor(m_fillPage.m_bkColor);

    CRect MyRectangle(20,10,20+m_Radius*2,10+m_Radius*2);
    dc.Ellipse (&MyRectangle);
    dc.SetBkColor(oldColor);
    dc.SelectObject(pOriginalPen);
 }
}

void CGraphDlg::OnTimer(UINT nIDEvent)
{
 //...
 pOriginalPen=dc.SelectObject(&m_nullPen);
 if (!(m_fillPage.m_FillStyle==8))
 {
    // A háttér törlése.
    //Ne rajzoljon vonalat!
    dc.FillRect(&myRectangle, &m_bkBrush);
 }
 // A méret változtatása.
 m_Radius = m_Radius + m_Direction;

 if (m_Radius>=100) { m_Direction = -1;}
 if (m_Radius<=3)  { m_Direction =  1;}
```

A főablak is vegye figyelembe az ecsetválasztást!

```
//  Invalidate();// A WM_PAINT kikényszerítése.
 // Az új kör kirajzolása.
 dc.SelectObject(&m_Pen);
 CBrush* pOriginalBrush;
 pOriginalBrush=dc.SelectObject(&m_Brush);
 COLORREF oldColor=dc.SetBkColor(m_fillPage.m_bkColor);
 myRectangle=CRect(20,10,20+m_Radius*2,10+m_Radius*2);
 dc.Ellipse (&myRectangle);
 dc.SetBkColor(oldColor);
 dc.SelectObject(pOriginalPen);
 dc.SelectObject(pOriginalBrush);

 CDialog::OnTimer(nIDEvent);
}
```

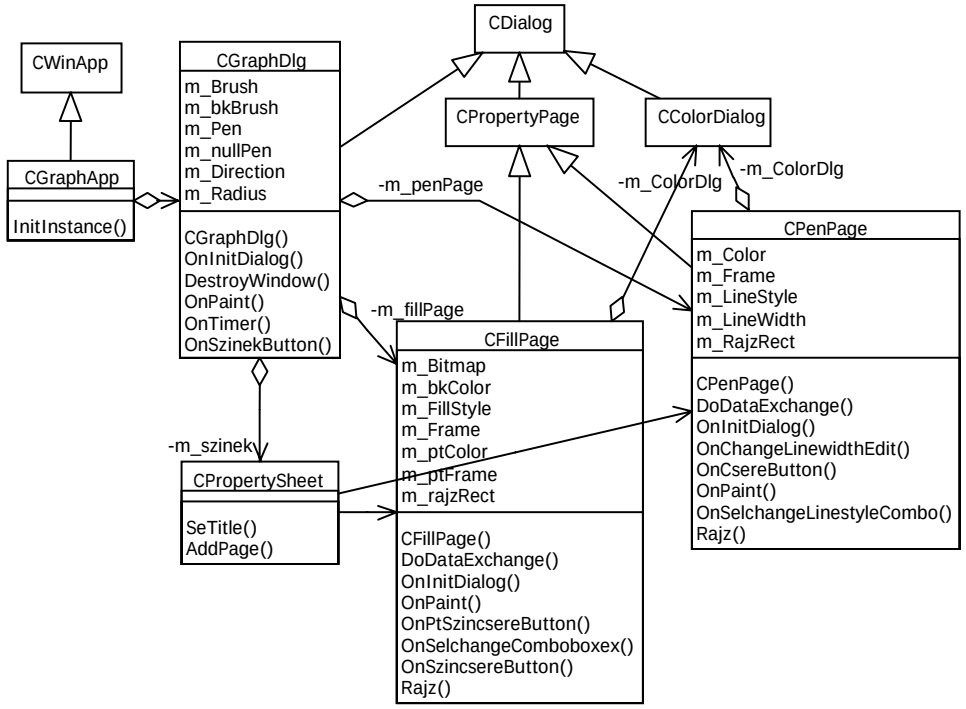
Ezután üresecset beállítással a következő ábrát is kaphatjuk:



5. Gyakorlat 31. ábra Ha az előző kört nem töröljük üresecset beállításnál

Megjegyzés:

A kombipaneleken szöveg helyett ábrák közül is kiválaszthatjuk a stílusokat. (Bár a minta nélküli és az üres ecset megkülönböztetése nem könnyű.)



5. Gyakorlat 32. ábra A 'Graph' alkalmazás osztálydiagramja

5.4. Az egér elkapása, téglalapba zárása

Húzzunk vonalat egy adott pontból oda, ahova az egér mutat, mialatt az egeret mozgatjuk!

Készítsünk párbeszédalapú alkalmazást MCapture néven! Szedjük ki belőle a vezérlőket!

Vegyünk fel egy CPoint típusú, m_OPoint adattagot a párbeszédosztályban! Adjunk neki kezdőértéket a konstruktorban! Ez lesz a vonalak kezdőpontja.

```
CMCaptureDlg::CMCaptureDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CMCaptureDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CMCaptureDlg)
     // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent
DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
  m_OPoint.x = 30;
  m_OPoint.y = 20;
}
```

Rendeljünk a párbeszédablak **WM_MOUSEMOVE** eseményéhez kezelőfüggvényt!

```
void CMCaptureDlg::OnMouseMove(UINT nFlags, CPoint point)
{
  CClientDC dc(this);
  dc.MoveTo(m_OPoint);
  dc.LineTo(point);
  CDialog::OnMouseMove(nFlags, point);

}
```

Teszteljük a programot! (Mivel nem mentjük a berajzolt vonalak adatait, az ablak frissítésekor nem látjuk a vonalakat! Ezzel törölhetünk tesztelés közben!) Mozgassuk az egeret az ablak felületén belül is és kívül is! Természetesen csak az ablak fölötti mozgásokra reagál a program. Próbáljuk meg mindezt lenyomott egérgombbal is!

➢ "Kapjuk el" az egeret!

A **WM_LBUTTONDOWN** eseményhez rendeljük hozzá a ***SetCapture()***, a **WM_LBUTTONUP**-hoz a ***ReleaseCapture()*** hívást!

```
void CMCaptureDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
  SetCapture();
  CDialog::OnLButtonDown(nFlags, point);
}

void CMCaptureDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
  ReleaseCapture();
  CDialog::OnLButtonUp(nFlags, point);
}
```

Teszteljük a programot az előzőkben leírt lépésekkel! Próbáljuk meg, hogy az egérgomb lenyomva tartásával az ablakon kívül mozgatva az egeret tovább rajzol a program.

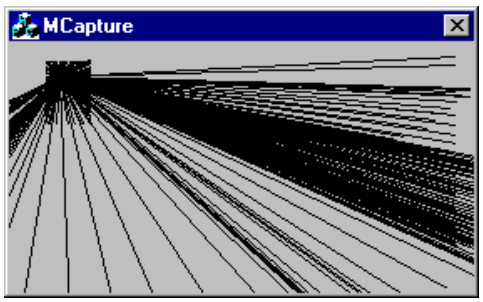
- **Ne engedjük ki az egeret a kliensablak egy téglalapjából!**

A bal oldali egérgomb duplakattintására hívódjon meg a korlátozás!

```
void CMCaptureDlg::OnLButtonDblClk(UINT nFlags, CPoint point)
{
 CRect rect;
 GetClientRect(&rect);      //Az ablak mérete
 ClientToScreen(&rect);     //Képernyő koordinátában
 ::ClipCursor(&rect);
 CDialog::OnLButtonDblClk(nFlags, point);
}
```

A jobbegér duplakattintás oldja fel! ***::ClipCursor(NULL)***

```
void CMCaptureDlg::OnRButtonDblClk(UINT nFlags, CPoint point)
{
 ::ClipCursor(NULL);
 CDialog::OnRButtonDblClk(nFlags, point);
}
```



5. Gyakorlat 33. ábra Az MCapture futás közben

Teszteljük más téglalap méretekkel is!

5.5. Feladatok:

5.5.1. Animált kurzor

Készítsünk az elmélet 5.4.3. szakaszában leírtak alapján animált kurzort! Használjuk hozzá az SDK AniEdit eszközét!

5.5.2. Pattogó pöttyös labda

Mozogjon ablakunkon keresztbe egy pöttyös labda! Az ablak széléhez érve a fizika törvényeinek megfelelően pattanjon vissza! Az ütközéskor be is lapulhat a labda!

5.5.3. Pöttyös labda görbült térben

Módosítsuk az előző programot úgy, hogy az ablak széléhez érkező labda, miközben kilép az ablakból, a túloldalon lépjen be! Mintha az ablak szélei henger alakban össze lennének ragasztva!

5.5.4. Tetszőleges alakú gomb*

Készítsünk ellipszis alakú vagy lekerekített sarkú, sőt tetszőleges alakú nyomógombot!

5.5.5. Besüllyedő nyomógomb*

Készítsünk nyomógombot, mely választás hatására benyomva marad, ill. visszaugrik!

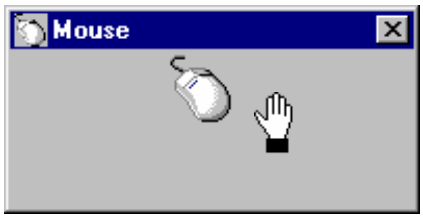
5. Gyakorlat 34. ábra Gombok

5.5.6. Számoló ovisok

Készítsünk számolni tanító programot ovisoknak! A képernyő felső részén legyenek képek, melyek tárgyak csoportját mutatják. Pl. 2 db meggy, 3 db mackó…(max. 9 db.). A képernyő alsó részén legyenek felsorolva a számjegyek! A játékos kattintson az egérrel az egyik képre, majd arra a számra ahány darab az adott figurából a képen látható! Ha talált, halljon egy dicséretet, ha nem, figyelmeztető hang következzen! Az egérkurzor legyen valamilyen játékfigura! Készíthetünk hasonlót betűkkel is!

5.5.7. Kapj el!*

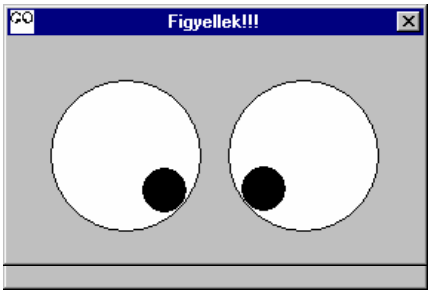
Egy számítógépes egér menekül (véletlen helyen bukkan föl) egy kéz formájú egér elől. Ha az egérrel sikerül rákattintani, akkor megáll. Újabb kattintásra újra elindul.



5. Gyakorlat 35. ábra Kapj el!

5.5.8. Figyellek!*

A képernyőn jelenjen meg egy ablak, melyen két szem látható! A szemek kövessék az egér irányát, ha az egér az ablak fölött van! Ha az egérgombot lenyomva tartjuk, kövessék az ablakon kívül is! (Ötlet: Baga, 1998, 96. old.)



5. Gyakorlat 36. ábra Figyellek!

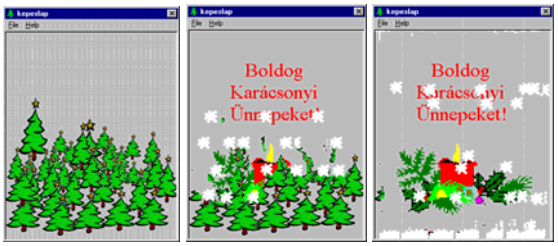
5.5.9. Kávé*

Készítsünk animációt! Pl. egy gőzölgő kávéscsésze képét!

5. Gyakorlat 37. ábra Kávé

5.5.10. Karácsonyi képeslap

Készítsen mozgó karácsonyi képeslapot! Hóesést, vagy animációt. Pl. Hóesés hatására az egyik képeslap átalakul a másikba.

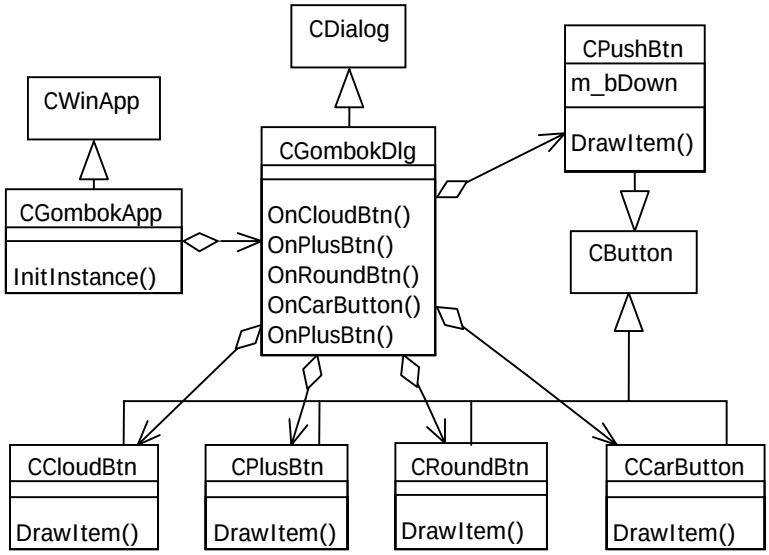


5. Gyakorlat 38. ábra Karácsonyi képeslap

5.6. Ötletek a feladatok megoldásához:

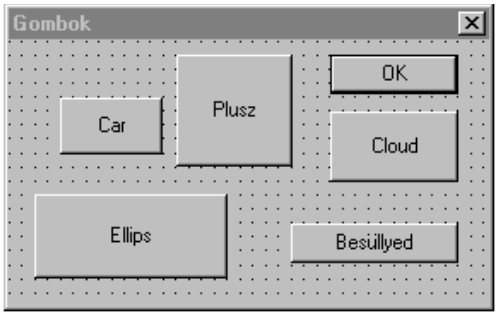
5.5.4. Tetszőleges alakú gomb

Készítsünk ellipszis alakú, lekerekített sarkú, sőt tetszőleges alakú nyomógombot! Készítsünk többfélét!



5. Gyakorlat 39. ábra A 'Gombok' alkalmazás osztálydiagramja

Helyzetüket az erőforrás-szerkesztőben határozhatjuk meg.



5. Gyakorlat 40. ábra Gombok az erőforrás-szerkesztőben

- Vegyünk fel a párbeszédfelületen egy nyomógombot, s a Properties Styles fülében állítsuk be az Owner draw választógombot!
- Készítsünk a projektünkhöz egy a CButton-ból származó, **CRoundBtn** nevű osztályt! Ebben fogjuk felülírni a nyomógomb rajzolását végző ***DrawItem()*** nevű virtuális függvényt!
- Rendeljünk a nyomógombhoz a ClassWizard segítségével egy CRoundBtn típusú m_Ellips adattagot a párbeszédablak osztályunkban! (Ne feledjük a CRoundBtn osztály fejlécfájlját beszerkeszteni!)

- Írjuk felül a CRoundBtn osztályban a CButton DrawItem virtuális függvényét! Amíg ezt nem tettük meg a CButton DrawItem üzenete hívódik meg, mely futtatáskor hibaüzenetet küld. Nem tudja kirajzolni a gombot. (A ClassWizard CRoundBtn-ben válasszuk ki a DrawItem üzenetet a Messages listáról, és rendeljünk hozzá kezelőfüggvényt!) A **LPDRAWITEMSTRUCT** lpDrawItemStruct paraméter segítségével lekérdezhetjük a párbeszédfelületen megszerkesztett nyomógomb adatait, pl. méretét, s ez segíthet a nyomógomb megrajzolásánál.

```
void CRoundBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 dc.Ellipse(&rect);
 dc.TextOut(rect.left+6, rect.top+rect.Height()/2-7,"Ellipszis
alakú");
}
```

Megjegyzés:

Mivel a párbeszédablak alapértelmezett színei az AFXWIN.H 4060. sora szerint:

```
void SetDialogBkColor(COLORREF clrCtlBk = RGB(192, 192, 192) ,
                          COLORREF clrCtlText = RGB(0, 0, 0));
      // set dialog box and message box background color
```

- Tehát ha a szokásos színű gombot szeretnénk, állítsuk mi is a nyomógombunk háttérszínét erre! Ha azonban más színt akarunk, azzal is dolgozhatunk.

```
void CRoundBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 CBrush redBrush;
 redBrush.CreateSolidBrush(RGB(255,0,0));
 CBrush* pOldBrush=dc.SelectObject(&redBrush);
 dc.Ellipse(&rect);
 dc.TextOut(rect.left+6, rect.top+rect.Height()/2-7,"Ellipszis
alakú");
 dc.SelectObject(pOldBrush);
}
```

A szöveg háttérszínét is át kell állítanunk!

```
void CRoundBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 CBrush redBrush;
 redBrush.CreateSolidBrush(RGB(255,0,0));
 CBrush* pOldBrush=dc.SelectObject(&redBrush);
 dc.Ellipse(&rect);
 COLORREF oldBkColor=dc.SetBkColor(RGB(255,0,0));
 dc.TextOut(rect.left+6, rect.top+rect.Height()/2-7,
                                          "Ellipszis alakú");
 dc.SetBkColor(oldBkColor);
 dc.SelectObject(pOldBrush);
}
```

Készítsünk hasonló módon további nyomógombokat! A képen láthatókhoz a DrawItem metódusok itt következnek.

Plusz alakú gomb:

```
void CPlusBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 // A minta kirajzolás háttere, tolla és ecsetje.
 COLORREF oldColor=dc.SetBkColor(RGB(192,192,192));
 CGdiObject* pOldPen=dc.SelectStockObject(NULL_PEN);
 CBrush hatchBrush;
 hatchBrush.CreateHatchBrush(5,RGB(255,255,255));
 CBrush* pOldBrush=dc.SelectObject(&hatchBrush);
 CRect hRect, vRect;
 vRect.SetRect(rect.left,rect.top+rect.Height()/2-10,
                     rect.right,rect.top+rect.Height()/2+10);

 // Hogy 3D gomb legyen!
 dc.DrawFrameControl(&vRect,DFC_BUTTON,DFCS_BUTTONPUSH);
 // Hogy látsszon a keret!
 vRect.SetRect(vRect.left+3,vRect.top+3,
                                  vRect.right-3,vRect.bottom-3);
 dc.Rectangle(&vRect);
 hRect.SetRect(rect.left+rect.Width()/2-10,rect.top,
    rect.left+rect.Width()/2+10,rect.top+rect.Height()/2-10);
 // Hogy 3D gomb legyen!
 dc.DrawFrameControl(&hRect,DFC_BUTTON,DFCS_BUTTONPUSH);
 // Hogy látsszon a keret!
 hRect.SetRect(hRect.left+3,hRect.top+3,
                                  hRect.right-3,hRect.bottom-3);
 dc.Rectangle(&hRect);
```

```
 hRect.SetRect(rect.left+rect.Width()/2-10,
              rect.top+rect.Height()/2+10,
              rect.left+rect.Width()/2+10,rect.bottom);
 // Hogy 3D gomb legyen!
 dc.DrawFrameControl(&hRect,DFC_BUTTON,DFCS_BUTTONPUSH);
 // Hogy látsszon a keret!
 hRect.SetRect(hRect.left+3,hRect.top+3,
                    hRect.right-3,hRect.bottom-3);
 dc.Rectangle(&hRect);

 // A DC visszaállítása.
 dc.SelectObject(pOldBrush);
 dc.SelectObject(pOldPen);
 dc.SetBkColor(oldColor);
}
```

Felhő formájú gomb:

A gomb formáját egy ikon adja meg. Ne feledkezzünk meg róla, hogy ikonban állíthatunk a háttérrel azonos és attól mindig eltérő színeket is!

Az ikon erőforrást természetesen el kell készíteni, azonosítója: IDI_ICON.

```
void CCloudBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 HICON ikon=AfxGetApp()->LoadIcon(IDI_ICON);
 dc.DrawIcon(0,0,ikon);
}
```

Autó formájú gomb:

A gomb formáját bitmap határozza meg. Ehhez is kell erőforrás, melynek azonosítója: IDB_BITMAP. A bitmap hátterét színezzük a felület hátterével azonos színűre!

```
void CCarButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 CBitmap bitmap;
 bitmap.LoadBitmap(IDB_BITMAP);
 CDC memDC;
 memDC.CreateCompatibleDC(&dc);
 memDC.SelectObject(&bitmap);
 dc.BitBlt(0,0,rect.Width(),rect.Height(),&memDC,0,0,SRCCOPY);
}
```

5. Gyakorlat 41. ábra A gombok futás közben

A besüllyedő gomb a következő feladat megoldásában szerepel.

Minden **owner draw** gomb kezelni tudja a **duplakattintás**t, más gomb pedig nem.

5.5.5. Besüllyedő nyomógomb

Készítsünk nyomógombot, mely ha kattintunk rajta benyomva marad, illetve visszaugrik!

- Az előzőek alapján készísünk owner draw stílusú nyomógombot, osztálya legyen **CPushBtn**, s a DrawItem tagfüggvényében rajzoljuk meg a nyomógombot!

Az ecset beállítása nem szükséges, de a szöveg kiírása miatt a háttérszínt állítanunk kell! Használjuk a **CDC** osztály ***DrawFrameControl()*** tagfüggvényének két különböző paraméterű hívását a rajzoláshoz! A példában szereplő gomb szövege is változik kattintás hatására.

- Először csak rajzoljuk ki a gombot!

```
void CPushBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 COLORREF oldColor=dc.SetBkColor(RGB(192,192,192));
    dc.DrawFrameControl(&rect,DFC_BUTTON,DFCS_BUTTONPUSH);
    dc.TextOut(5,3,"Benyomódó!");
 dc.SetBkColor(oldColor);
}
```

- Ahhoz, hogy a gomb benyomódva maradjon, vegyünk fel egy logikai adattagot az osztályban, mely a gomb állapotát őrzi!

```
// Attributes
public:
 BOOL m_bDown;
```

- A konstruktorban inicializáljuk!

```
CPushBtn::CPushBtn()
{
 m_bDown=FALSE;
}
```

- A kirajzolt nyomógomb képe függjön e változó értékétől!

```
void CPushBtn::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
 CClientDC dc(this);
 CRect rect=lpDrawItemStruct->rcItem;
 COLORREF oldColor=dc.SetBkColor(RGB(192,192,192));
 if (m_bDown)
 {
    dc.DrawFrameControl(&rect,
                    DFC_BUTTON,DFCS_BUTTONPUSH | DFCS_PUSHED);
    dc.TextOut(5,3,"Besüllyedt!");
 }
 else
 {
    dc.DrawFrameControl(&rect,DFC_BUTTON,DFCS_BUTTONPUSH);
    dc.TextOut(5,3,"Benyomódó!");
 }
 dc.SetBkColor(oldColor);
}
```

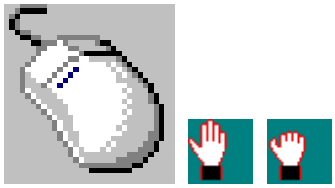
- A gomb választás hatására változzon az m_bDown értéke! Kezeljük le a nyomógomb BN_CLICKED üzenetét! Ez a Dialog osztályban történik. Ha a nyomógombhoz az CPushBtn::m_Push adattagot rendeltük, akkor a párbeszédosztály OnPushButton() tagfüggvényébe a következő kódot írhatjuk.

```
void CGombokDlg::OnPushButton()
{
 m_Push.m_bDown = !  m_Push.m_bDown ;
 m_Push.RedrawWindow();
}
```

5. Gyakorlat 42. ábra A besüllyedő gomb két állapota

5.5.7. Kapj el!

Először készítsük el az erőforrásokat! A menekülő egér egy bitmap, az egérkurzorhoz pedig két állapot kell, a mozgatás alatti és amikor megpróbálja elkapni. Nem árt, ha az alkalmazás ikonjába mindkét mérethez bemásoljuk a bitmapet. (A kicsihez kicsinyíteni lehet a kép kijelölt ablakának méretezésével.)



5. Gyakorlat 43. ábra A bitmap és a két egérkurzor

A CMouseDlg osztályban mindháromhoz tagváltozót rendelünk. Az m_move azt az állapotot adja meg, hogy az egér éppen mozog, vagy éppen megállította a felhasználó.

```
// Implementation
protected:
 HICON m_hIcon;
 HCURSOR m_hCursor;
 HCURSOR m_hFogCursor;
 CBitmap m_Bitmap;
 int m_Move;
```

A konstruktorban inicializáljuk őket. Az m_Size a kép bal fölső sarkának lehetséges maximális értékeit tartalmazza.

```
m_hCursor = AfxGetApp()->LoadCursor(IDC_CURSOR1);
m_hFogCursor = AfxGetApp()->LoadCursor(IDC_FOG_CURSOR);
m_Move = 1;
m_Bitmap.LoadBitmap(IDB_MOUSE);
m_Size=CSize(31,33); //m_Bitmap.GetBitmapDimension();
// Kliensméret-képméret.
m_Size.cx=GetSystemMetrics(SM_CXFULLSCREEN)-m_Size.cx;
m_Size.cy=GetSystemMetrics(SM_CYFULLSCREEN)-m_Size.cy;
```

Az OnInitDialog állítja be az ablakot teljes képernyősre, elindítja az időzítőt, ami majd mozgatja a bitmapet, és létrehozza a memória DC-t, hogy a bitmapet ki tudjuk tenni az ablakba. Ne feledjük az objektumok megszüntetését sem!

```
MoveWindow(0,0,GetSystemMetrics(SM_CXMAXIMIZED),
                          GetSystemMetrics(SM_CYMAXIMIZED));
if (SetTimer(PULSE, 500, NULL) = = 0)
   { MessageBox( „Nem tudtam létrehozni az időzítőt”); }
```

```
CClientDC dc(this);
m_memDC.CreateCompatibleDC(&dc);
m_memDC.SelectObject(m_Bitmap);
```

Az egér képét az OnMouseMove-ban állíthatjuk be a kézre:

```
SetCursor(m_hCursor);
```

A rajz mozgatását az időzítő végzi az OnTimerben:

```
if (m_Move)
{
   CClientDC dc(this);
   CRect rect(m_Point,m_Point+CSize(31,33));
   CBrush brush;
   brush.CreateStockObject(LTGRAY_BRUSH);
   dc.FillRect(&rect,&brush);

   m_Point.x= rand()*m_Size.cx/RAND_MAX;
   m_Point.y= rand()*m_Size.cy/RAND_MAX;
   dc.BitBlt(m_Point.x,m_Point.y,m_Size.cx,m_Size.cy,&m_memDC,
                                                  0,0,SRCCOPY);
   brush.DeleteObject();
}
```

A kattintáskor megjelenő egeret az OnLButtonDown-ban állítsuk be, míg a felengedéskor (OnLButtonUp) vissza kell állítani a kéz egérrajzát, ha nem akarjuk, hogy az eredeti nyíl formájú ikon is megjelenjen! A kattintáskor azt is ellenőriznünk kell, megfelelő helyen volt-e az egérkurzor, mert ha igen, meg kell állítanunk a kép mozgatását!

```
void CMouseDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
 SetCursor(m_hFogCursor);
 CRect rect(m_Point,m_Point+m_Size);
 if (PtInRect(&rect,point))// Ha eltalálta
 {
    if (m_Move)
        m_Move--;
    else
        m_Move++;
 }
 CDialog::OnLButtonDown(nFlags, point);
}
```

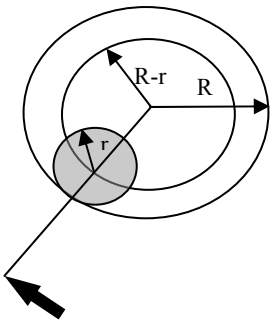
Megjegyzés:

Ablakhoz alapértelmezett kurzor hozzárendelésére láthatunk példát a 6.6.4. feladat megoldásánál.

5.5.8. Figyellek!

A képernyőn jelenjen meg egy ablak, melyen két szem látható. A szemek kövessék az egér irányát, ha az egér az ablak fölött van! Ha az egérgombot lenyomva tartjuk, kövessék az ablakon kívül is!

A fekete kör középpontja a fehér kör középpontja és a kurzor pontja által meghatározott egyenesen, valamint a fehér kör középpontú, R-r sugarú körön található. (R a fehér, r a fekete kör sugara.) Az egyenes és a kör egyenletét felírva, s mint egyenletrendszert megoldva megkapjuk a fekete kör középpontjának koordinátáit.



5. Gyakorlat 44. ábra A szem

A megoldás egyszerűbb, ha a koordinátarendszer origóját a fehér kör középpontjába helyezzük. Tehát transzformálnunk kell a pontok koordinátáit ennek megfelelően.

Az alkalmazás kódja:

A párbeszédosztály:

```
class CFigyDlg : public CDialog
{
// Construction
public:
  CFigyDlg(CWnd* pParent = NULL);  // standard constructor

// Dialog Data
  //{{AFX_DATA(CFigyDlg)
  enum { IDD = IDD_FIGY_DIALOG };
     // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CFigyDlg)
  public:
  virtual BOOL DestroyWindow();
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV
support
  //}}AFX_VIRTUAL
```

```
// Implementation
protected:
 HICON m_hIcon;

 // Generated message map functions
 //{{AFX_MSG(CFigyDlg)
 virtual BOOL OnInitDialog();
 afx_msg void OnPaint();
 afx_msg HCURSOR OnQueryDragIcon();
 afx_msg void OnMouseMove(UINT nFlags, CPoint point);
 afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
 afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
 //}}AFX_MSG
 DECLARE_MESSAGE_MAP()
private:
 CBitmap m_Bitmap;
 CDC m_memDC;
 void DrawCircle();
 void DeleteCircle();
 void MyToClient(CPoint O, CPoint& point);

 // Ahova a kurzor mutat.
 CPoint MyP;
 void ClientToMy(CPoint O, CPoint& point);

 // A körök sugarai.
 int r;
 int R;

 // A körök középpontjai.
 CPoint o2;
 CPoint O2;
 CPoint o1;
 CPoint O1;
};
```

Az inicializálás és a befejezés:

```
CFigyDlg::CFigyDlg(CWnd* pParent /*=NULL*/)
 : CDialog(CFigyDlg::IDD, pParent)
{
 m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
 O1.x = 80; O1.y = 80;
 O2.x = 200; O2.y = 80;
 R = 50;
 r = 15;
}
```

```
BOOL CFigyDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
 //...
 // TODO: Add extra initialization here
 // A szemgolyók, a fekete körök középpontja
 o1.x=80; o1.y=80;
 o2.x=200; o2.y=80;
 CClientDC dc(this);

 // Ide rajzolunk.
 m_memDC.CreateCompatibleDC(&dc);
 CRect rect;
 GetClientRect(&rect);
 // Erre a bitmapre rajzolunk.
m_Bitmap.CreateCompatibleBitmap(&dc,rect.Width(),rect.Height());
 m_memDC.SelectObject(&m_Bitmap);

 // Szürke háttér.
 CBrush brush;
 brush.CreateStockObject(LTGRAY_BRUSH);
 m_memDC.FillRect(&rect,&brush);
 brush.DeleteObject();
 return TRUE;  // return TRUE  unless you set the focus to a
control
}

BOOL CFigyDlg::DestroyWindow()
{
 // TODO: Add your specialized code here and/or call the base
class
 ReleaseDC(&m_memDC);
 m_Bitmap.DeleteObject();
 return CDialog::DestroyWindow();
}
```

Az osztályvarázsló által generált üzenettérkép:

```
BEGIN_MESSAGE_MAP(CFigyDlg, CDialog)
 //{{AFX_MSG_MAP(CFigyDlg)
 ON_WM_PAINT()
 ON_WM_QUERYDRAGICON()
 ON_WM_MOUSEMOVE()
 ON_WM_LBUTTONDOWN()
 ON_WM_LBUTTONUP()
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

A megjelenéskor és az ablak frissítésekor csak a nagy körök rajzolódnak ki:

```
void CFigyDlg::OnPaint()
{
 CPaintDC dc(this); // device context for painting
 if (IsIconic())
 {
 // ...
 }
 else
 {
    m_memDC.Ellipse(O1.x-R-1,O1.y-R-1, O1.x+R+1,O1.y+R+1);
    m_memDC.Ellipse(O2.x-R-1,O2.y-R-1, O2.x+R+1,O2.y+R+1);
    CRect rect;
    GetClientRect(&rect);
dc.BitBlt(0,0,rect.Width(),rect.Height(),&m_memDC,0,0,SRCCOPY);

    CDialog::OnPaint();
 }
}
```

Átkonvertálás a középpontok koordinátáihoz mint origóhoz és vissza.

```
void CFigyDlg::ClientToMy(CPoint O, CPoint & point)
{
 point.x -=O.x;
 point.y -=O.y;
}

void CFigyDlg::MyToClient(CPoint O, CPoint & point)
{
 point.x +=O.x;
 point.y +=O.y;
}
```

A kör letörlése és kirajzolása a memóriában:

```
void CFigyDlg::DeleteCircle()
{
 m_memDC.SelectStockObject(WHITE_BRUSH);
 m_memDC.SelectStockObject(WHITE_PEN);
 m_memDC.Ellipse(o1.x-r,o1.y-r,o1.x+r,o1.y+r);
 m_memDC.Ellipse(o2.x-r,o2.y-r,o2.x+r,o2.y+r);
}

void CFigyDlg::DrawCircle()
{
 m_memDC.SelectStockObject(BLACK_BRUSH);
 m_memDC.SelectStockObject(BLACK_PEN);
 m_memDC.Ellipse(o1.x-r,o1.y-r,o1.x+r,o1.y+r);
 m_memDC.Ellipse(o2.x-r,o2.y-r,o2.x+r,o2.y+r);
}
```

A szemgolyók követik az egérmozgást:

```
void CFigyDlg::OnMouseMove(UINT nFlags, CPoint point)
{
 // TODO: Add your message handler code here and/or call
default
 DeleteCircle();
 MyP = point;
 ClientToMy(O1,MyP);
 // 0-val nem oszthatunk!
 if (MyP.x == 0)
    MyP.x =1;
 if (MyP.y == 0)
    MyP.y =1;
 // (R-r)2=x2+y2
 // (R-r)2=x2(x2+y2)/x2
 // (R-r)2=x2(1+y2/x2)     h=x2/y2
 // (R-r)2/(1+h)=x2
    double h=(double)MyP.y /(double) MyP.x;
    h*=h;
    long x=(R-r)*(R-r);
     if (MyP.x > 0)
        o1.x = (long)sqrt(x/(1+h));
    else
        o1.x = -(long)sqrt(x/(1+h));
    if (MyP.y > 0)
        o1.y = (long)sqrt(x-o1.x * o1.x);
    else
        o1.y = -(long)sqrt(x-o1.x * o1.x);
 MyToClient(O1,o1);  // A kis kör kp. kliens koordinátában

//Az o2 kör középpontja
 MyP = point;
 ClientToMy(O2,MyP);
 if (MyP.x == 0)
    MyP.x =1;
 if (MyP.y == 0)
    MyP.y =1;
    h=(double)MyP.y /(double) MyP.x;
    h*=h;
    x=(R-r)*(R-r);
     if (MyP.x > 0)
        o2.x = (long)sqrt(x/(1+h));
    else
        o2.x = -(long)sqrt(x/(1+h));
    if (MyP.y > 0)
        o2.y = (long)sqrt(x-o2.x * o2.x);
    else
        o2.y = -(long)sqrt(x-o2.x * o2.x);
 MyToClient(O2,o2);  // A kis kör kp. kliens koordinátában
```

```
 DrawCircle();
 // A memDC-ről kirajzolja a dc-re:
 CClientDC dc(this);
 CRect rect;
 GetClientRect(&rect);
dc.BitBlt(0,0,rect.Width(),rect.Height(),&m_memDC,0,0,SRCCOPY);

 CDialog::OnMouseMove(nFlags, point);
}
```

Ha lenyomva tartjuk a bal oldali egérgombot, az ablakon kívül is követi a szem az egér mozgását. Felengedve az egérgombot, a követés megszűnik.

```
void CFigyDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
 SetCapture();
 CDialog::OnLButtonDown(nFlags, point);
}

void CFigyDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
 ReleaseCapture();
 CDialog::OnLButtonUp(nFlags, point);
}
```

A szem kezdetben is és frissítéskor is jelenjen meg!

Ha most teszteljük, akkor a szemek kezdetben üres fehér, míg frissítéskor fekete körként jelennek meg. Módosítsuk az OnPaint else ágát, ahol a kirajzolás történik!

```
 else
 {
    m_memDC.SelectStockObject(WHITE_BRUSH);     // Fehér szem.
    m_memDC.SelectStockObject(BLACK_PEN);
    m_memDC.Ellipse(O1.x-R-1,O1.y-R-1, O1.x+R+1,O1.y+R+1);
    m_memDC.Ellipse(O2.x-R-1,O2.y-R-1, O2.x+R+1,O2.y+R+1);
    SendMessage(WM_MOUSEMOVE);                  // Szemgolyók.
    CRect rect;
    GetClientRect(&rect);
    dc.BitBlt(0,0,rect.Width(),rect.Height(),&m_memDC,0,0,
                                                    SRCCOPY);

    CDialog::OnPaint();
 }
```

5.5.9. Kávé

A legnagyobb feladat a képek elkészítése. Ehhez másoljuk át az eredeti képet, majd módosítsuk a szükséges részt! Ha folyamatos lejátszást akarunk, ügyelni

kell rá, hogy az utolsó képkocka egyezzen meg az elsővel! (Ez az utolsó nem is kell!)

Az adattagok felvétele, inicializálása után az időzítő segítségével folyamatosan olvassuk be a képeket! Ne feledkezzünk meg az objektumok felszámolásáról sem!

```
void CCofeDlg::OnTimer(UINT nIDEvent)
{
 CClientDC dc(this);
 dc.BitBlt(30,20,47,47,&m_memDC,0,0,SRCCOPY);
 m_memDC.FillRect(CRect(30,20,47,47),&m_brush);
 m_bitmap.DeleteObject();
 m_bitmap.LoadBitmap(IDB_COFE1+m_ind++);
 m_memDC.SelectObject(&m_bitmap);
 if (m_ind>16) m_ind=0;//0-tól indexelünk
 CDialog::OnTimer(nIDEvent);
}
```

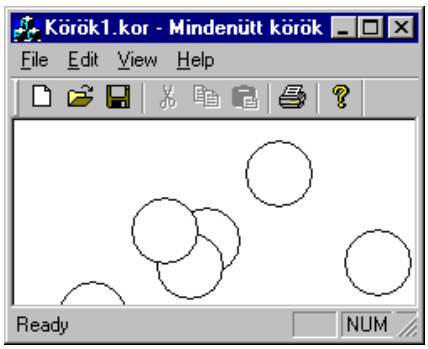
A frissítésnek itt nincs jelentősége, hisz az új ábra elég gyakori bekérésével a frissítés automatikusan megvalósul. Mégis írhatjuk rajzoló kódunkat az OnPaint metódusba is. Ekkor az OnTimer feladata csak az m_ind értékének változtatása, és a frissítés kikényszerítése (`InvalidateRect(CRect(30,20,47,47));`) lesz. Ennél a megoldásnál viszont kis villogás figyelhető meg a rajzunkon a képernyőn.

6. Gyakorlat SDI, MDI alkalmazások

6.1. 'Mindenütt körök' feladat

Készítsünk SDI alkalmazást!

1. A program elején egy kört láthatunk a képernyőn, az egérrel az alkalmazás kliensterületének különböző pontjain kattintva a kör a kattintás helyén jelenjen meg!
2. Kezdetben az alkalmazás üres, az egérrel az alkalmazás különböző pontjaira kattintva új kör jelenjen meg abban a pontban!



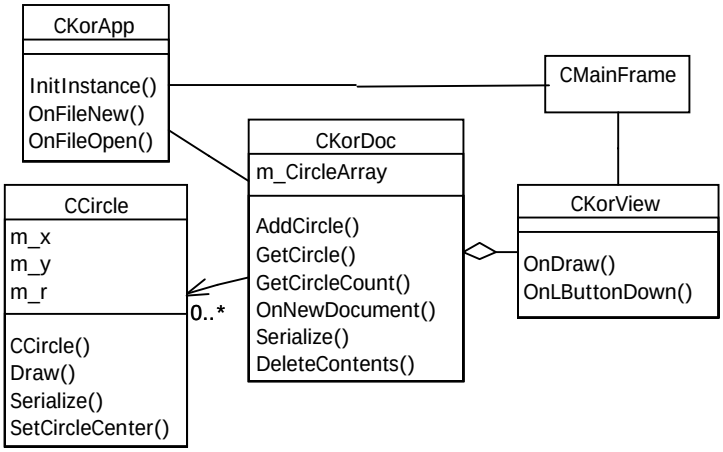
6. Gyakorlat 1. ábra A kész kör alkalmazás

Mentéskor a köröket az aktuális helyükön mentsük, s fájl megnyitásakor ott jelenjenek meg, ahonnét elmentettük őket!

A mentett fájlok kiterjesztése alapértelmezésben legyen '.kor', és ez a szűrés legyen beállítva nyitáskor a programban!

A fájlkezelőből **drag & drop** technikával áthúzva a '.kor' kiterjesztésű fájlokat az alkalmazás fölé, az alkalmazás nyissa meg azokat!

A fájlkezelőben a '.kor' kiterjesztésű fájlokra duplán kattintva azok hívják meg a kör alkalmazást, és töltsék be saját magukat!



6. Gyakorlat 2. ábra A kör alkalmazás osztálydiagramja

A tervezésnél végiggondoltuk, hogy **SDI** alkalmazássablont használunk.

- Az **alkalmazásosztály** (**CKorApp**) hozza létre a keretablak (CMainFrame), dokumentum (CKorDoc) és nézetosztályok (CKorView) objektumait (InitInstance), és a CWinApp osztály metódusait hívva kezeli a mentés ***OnFileSave()*** és megnyitás ***OnFileOpen()*** műveleteket. (Lásd Message Map!)
- A dokumentumosztály felel az adatok tárolásáért, CCircle objektumokat tárol egy dinamikus tömbben. A **CCircle osztály** adatként tárolja a kör középpontjának x és y koordinátáját és a sugarát. A konstruktor (CCircle) létrehozza a kört, a Draw kirajzolja, a SetCircleCenter a kör középpontját a paraméterében megadott pontba állítja. A köröket szerializációval mentjük, ehhez kell a Serialize() metódus.
- A **dokumentumosztály** (**CKorDoc**) tud új kört fölvenni (AddCircle), lekérdezni a tömb i-edik elemét (GetCircle), a körök számát. Képes új dokumentumot nyitni (OnNewDocument), törölni a dokumentumosztály adatait (DeleteContents). SDI alkalmazásnál ez fontos, hisz az osztály objektumát újrafelhasználjuk, és szerializációval (Serialize) mentjük az adatokat.
- A **nézetosztály** (**CKorView**) kirajzolja a dokumentumosztály adatait (OnDraw) és kezeli az egérkattintás eseményt (OnLButtonDown).
- A **keretablakosztály** (**CMainFrame**) kezeli a menüsor, eszköztár, címsor és státuszsor eseményeit.

Az osztálydiagram nem tartalmazza az alkalmazás összes osztályát (ezt összetettebb alkalmazások esetén nem is lehet egy diagramban ábrázolni), csak az alkalmazás szempontjából lényegeseket. Az áttekinthetőség kedvéért nem

tartalmazza a CSingleDocTemplate osztályt, mely a Document / View architektúra része, és az alkalmazásosztály egy példánya segítségével hozza létre a keretablak, a dokumentum és nézetablak példányait. Nem szerepel az ábrán a CPtrArray template osztály – mely az m_CircleArray tagváltozó típusa – és a CFileDialog osztály, amit a fájlnyitás és a fájlmentés során használunk. Nem mutatja az ábra a felhasznált osztályok őseit.

6.1.1. A kör alkalmazás standard tagváltozókkal

- **Az SDI keretprogram**

File menü **- New** menüpont

Projects fül

Név beírása: **Kor.**

MFC AppWizard(exe).

Location: A könyvtár meghatározása. **OK**

Az alkalmazás típusa: Single document.

Adatbázis beállítások: None.

Támogassa-e a komponensek használatát: None

Nem szükséges az ActiveX kontrolok támogatása. (Pipát ki!)

Jellemzők: Docking Toolbar. / Initial status bar. / Printing and print preview. / 3D controls. Hány file szerepeljen az aktuális file listán?: 4.

Advanced gomb:

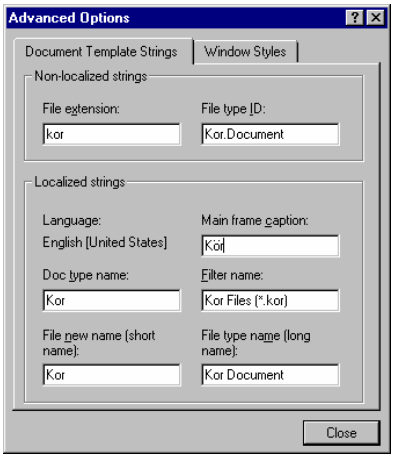
Document Template Strings fül:

File extension: kor.

Main frame caption:

Kör.

Nézzük meg a Window Styles fület is!



6. Gyakorlat 3. ábra Az alkalmazásvarázsló Advanced Options párbeszédablaka

A projekt stílusa: MFC Standard.

A tárgykódban megjegyzések: Yes, please.

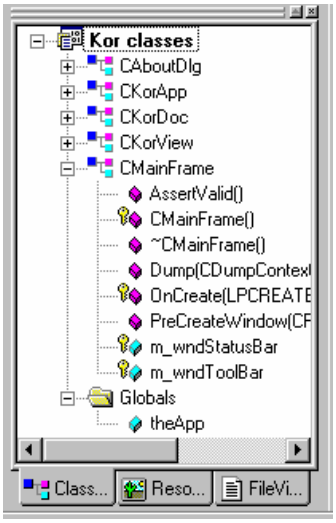
Dinamikusan szerkesztett. (As a shared DLL.)

Milyen osztályok és file-ok kellenek: Alapbeállítás.

Finish - A beállítások újra átnézhetők, majd OK.

➢ Nézzük meg a programot!

Nézzük meg a létrehozott új fájlokat, osztályokat! Az alkalmazásosztályon kívül megtaláljuk a Document / View architektúra fontosabb osztályait. A 6. Gyakorlat 4. ábrán a keretablak osztály adattagjai között láthatjuk a státuszsor és az eszköztár objektumait.



6. Gyakorlat 4. ábra A ClassView fül

Fordítsuk le, szerkesszük, majd futtassuk az üres alkalmazást!

Futtassuk a programot, és vizsgáljuk meg:

Az alkalmazás tartalmaz

- egy alapértelmezett menüt
- egy eszköztárat
- egy állapotsort

Vizsgáljuk meg mely menüpontok és milyen szinten működnek! Figyelem! A File menü Open hívása után ne hívjuk meg a Save menüpontot, mert amíg a mentés kódját nem írjuk meg, a megnyitott fájlba az alkalmazás nem ment semmit, vagyis Save után üres lesz a fájl! Próbáljuk ezt ki egy erre a célra létrehozott fájllal, pl. a ReadMe.txt másolatával!

Nyissuk ki a ReadMe.txt-t a Visual C++ fejlesztői környezetben, és mentsük el ReadMe1.Txt néven!

Ezután nyissuk meg a futó 'Kör' alkalmazásunkban a Readme1.txt-t. A megnyitáshoz az alkalmazásvarázslóban beállítottuk az alapértelmezett fájlkiterjesztést (.kor). Természetesen nincs még ilyen fájl, tehát az Open párbeszédablak Files of type listájából az 'All files (*.*)' beállítást válasszuk! (6.Gyakolat 4. ábra.) Majd az ablakban megtaláljuk a ReadMe1.txt fájlt. Nyissuk meg! Adatait az alkalmazás nem tudja értelmezni, megjeleníteni, de megnyitotta.

Mentsük a Save paranccsal! A fájl üres lesz. Megnézhetjük a tartalmát a Visual C++ fejlesztői környezetben.

6. Gyakorlat 5. ábra A kör alkalmazás Open ablaka

Vizsgáljuk meg az alkalmazás forráskódját is!

Az alkalmazásosztály implementációs fájljában (kor.cpp):

```
////////////////////////////////////////////////////////////////////
//////////////
// The one and only CKorApp object

CKorApp theApp;

////////////////////////////////////////////////////////////////////
//////////////
// CKorApp initialization

BOOL CKorApp::InitInstance()
{
//...
  // Change the registry key under which our settings are
  //stored. TODO: You should modify this string to be something
  //appropriate such as the name of your company or
  //organization.
  SetRegistryKey(_T("Local AppWizard-Generated Applications"));

  LoadStdProfileSettings();
  // Load standard INI file options (including MRU)
```

Az egyetlen alkalmazásobjektum

A registry kulcs létrehozása. Módosítsuk a saját adatainkra!

```
  // Register the application's document templates. Document
  //templates serve as the connection between documents, frame
  //windows and views.

  CSingleDocTemplate* pDocTemplate;
  pDocTemplate = new CSingleDocTemplate(
     IDR_MAINFRAME,
     RUNTIME_CLASS(CKorDoc),
     RUNTIME_CLASS(CMainFrame),       // main SDI frame window
     RUNTIME_CLASS(CKorView));
  AddDocTemplate(pDocTemplate);

  // Enable DDE Execute open
  EnableShellOpen();
  RegisterShellFileTypes(TRUE);
// ...
  // The one and only window has been initialized, so show and
update it.
  m_pMainWnd->ShowWindow(SW_SHOW);
  m_pMainWnd->UpdateWindow();

  // Enable drag/drop open
  m_pMainWnd->DragAcceptFiles();

  return TRUE;
}
```

A dokumentumsablon és az alkalmazásszerkezet objektumainak létrehozása.

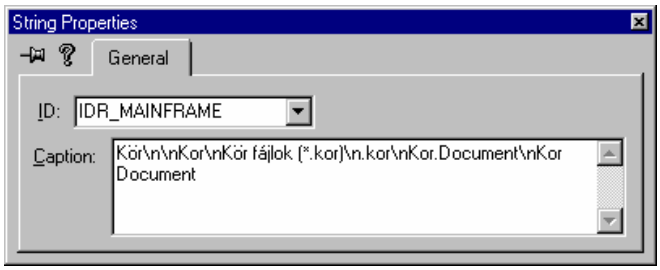
A fájlkiterjesztés regisztrálása. Lásd elmélet 7. fejezet!

Drag & drop engedélyezése a főablakra.

Nézzük meg a Resource View String Table **IDR_MAINFRAME**-ben a használt sztringeket! Javítsuk ki például a Kor Files szöveget Kör fájlokra!

A módosításhoz kétszer kattintsunk az IDR_MAINFRAME azonosítóra! A tulajdonságlap Caption ablakában végezhetjük el a módosítást.

Figyelmeztetés: A szöveg ne tartalmazzon Entert, a Visual C++ tördelni fogja a sort, ha szükséges!



6. Gyakorlat 6. ábra A kör String Table módosítása

➢ A dokumentumosztály adatmezői

A CKorDoc osztály új adatmezői tárolják a kör középpontjának koordinátáit:

```
class CKorDoc : public CDocument
// Attributes
public:
    int m_x;
    int m_y;
```

Itt azokat az adatokat tároljuk, amik majd el lesznek mentve a fájlba, amelyből később a program előhívhatja a régebben rajzolt kört.

Ha más adatokat is szeretnénk tárolni a körünkről, pl. a sugarát vagy a rajzoló színt, ezek számára is létrehozhatunk adatmezőket.

Az adatok inicializálása:

KorDoc.cpp / ***OnNewDocument()*** függvénymező. Az adatokat nem a konstruktorban inicializáljuk, mert a futó alkalmazás, ha új dokumentumot kér, akkor nem hívja a konstruktort (elmélet 6.2.2.2.szakasz), az inicializálást pedig akkor is végre kell hajtani.

```
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
m_x=200;
m_y=100;
```

Ez a tagfüggvény automatikusan meghívódik, amikor indítjuk a programot vagy ha egy új dokumentumot hozunk létre.

➢ A kód, amely megjeleníti a kört

Osztályvarázsló. / CKorView. / OnDraw()

```
void CKorView::OnDraw(CDC* pDC)
{
 CKorDoc* pDoc = GetDocument();
 ASSERT_VALID(pDoc);
 // TODO: add draw code for native data here
 RECT rect;
 rect.left  = pDoc->m_x-20;
 rect.top   = pDoc->m_y-20;
 rect.right = pDoc->m_x+20;
 rect.bottom= pDoc->m_y+20;
 pDC->Ellipse(&rect);
}
```

A rajzolt kör sugara 20 egység. A középpontja m_x, m_y.

A tagfüggvény mindig végrehajtódik, amikor a képernyőt újra kell rajzolni, vagy amikor a nyomtatást választjuk a File menüből.

Nyissunk ablakot a futó program fölé! Húzzuk le róla! Mozgassuk az alkalmazást addig, míg a képernyőn kívül nem kerül! Nézzük meg a nyomtatási képet, a nyomtató beállítást, s ha van nyomtatónk, nyomtassuk ki a képet!

➢ Az egérrel változtassuk a kör helyét!

Mivel a változtatás látszik a képernyőn, a CKorView osztályba kell írnunk az esemény kódját.

Az esemény: **WM_LBUTTONDOWN** / Add Function / Edit Code / CKorView::***OnLButtonDown()***

```
// TODO: Add your message handler code here and/or call default
CKorDoc* pDoc = GetDocument();
pDoc->m_x = point.x;
pDoc->m_y = point.y;
pDoc->SetModifiedFlag(TRUE);
// Kilépéskor innét tudja a rendszer: módosultak az adatok, s
// figyelmeztet a mentésre.
Invalidate(); // Előidézi az OnDraw() hívást.
```

A futó program az egérkattintásra átrajzolja a kört, de az új helyzetet nem menti el. Eddig sem mentette, a mentett fájl létrejött, de üres!

➢ Mentés és betöltés

Az adatok tárolása és a mentés a dokumentumosztály feladata.

KorDoc.cpp / ***Serialize()***;

Ez a függvény automatikusan végrehajtódik, ha a Save, Save As vagy Open menüpontokat választjuk a File menüből. Attól függően, hogy írunk a fájlba vagy olvasunk onnét, az ***IsStoring()*** elágazás igaz vagy hamis ága hajtódik végre.

```
void CKorDoc::Serialize(CArchive& ar)
{
 if (ar.IsStoring())
 {
    // TODO: add storing code here
    ar << m_x << m_y;
 }
 else
 {
    // TODO: add loading code here
    ar >> m_x >> m_y;
 }
}
```

Megjegyzés:

1. A mentés és a beolvasás esetén az adatok sorrendjének meg kell egyezni! Egyébként hibás lesz a beolvasás.
2. Az Exit menüpont előtt felsorolja a File menü az utolsó 4 **(Ennyit állítottunk be a keretprogramban)** mentett fájl nevét. Ezek bármelyikére kattintva, a mentett fájlt nyitja meg a rendszer. Az épp betöltöttel megegyező nevű fájlt nem tölti be a program, tehát teszteléskor ne várjuk a kép módosulását!
3. Próbálja ki a mentés és nyitás ikon működését is!

6.1.2. Tároljuk egy kör adatait egy CCircle osztályban!

Ha mi az adatokat egy általunk definiált osztályban tároljuk, akkor az előző, nem túl elegáns módszeren kívül `(ar << m_Circle.m_x << m_Circle.m_y;)`, adatainkat kétféleképpen menthetjük.

1. A CCircle osztályhoz tartozó inserter és extraktor operátorokkal, akkor a programnak tartalmaznia kell a << és >> operátorok új paraméterekkel történő átdefiniálását is, tehát ezt is el kell készítenünk.
2. Serialize függvény megírásával. Ezt a módszert kínálja nekünk az MFC, tehát használjuk ezt! Collection objektumok (tömb, lista) használata esetén e választás rendkívül kifizetődő lesz.

➢ Hozzuk létre a CCircle osztályt!

A Serialize szolgáltatást csak CObject utódosztályok tudják igénybe venni, így CCircle osztályunknak a CObject-ből kell származnia. Mivel ezt a lehetőséget a ClassView / New class párbeszédablakában nem kínálja föl (és az osztályvarázsló sem), így kénytelenek leszünk kézzel létrehozni az új osztályt. Lehetne az About Dialog-hoz hasonlóan egy már meglevő osztályban, mint a CKorDocument, vagy külön két fájlban, hátha még máskor is használni akarjuk.

Workspace

FileView fül

Header Files jobb egér

Add Files to Folder

File name: Circle.h does not exists **YES**

A header file-ok listájából válasszuk ki, és próbáljuk megnyitni! Az üzenet után, miszerint nem létezik a fájl, s hozza-e létre, igennel válaszolva, kezdhetjük a deklarációt!

```
// Circle.h header file
//
////////////////////////////////////////////////////////////////

class CCircle : public CObject
{
protected:
  int m_x;
  int m_y;
  int m_r;
public:
  CCircle(int x = 50, int y = 50, int r = 20)
        {m_x = x; m_y = y; m_r = r;}
              // Default konstruktor is!
  void Draw(CDC* pDC)
        { pDC->Ellipse(m_x-m_r, m_y-m_r,m_x+m_r,m_y+m_r);}
  CPoint SetCircleCenter(CPoint point);
};
```

Ne felejtsük az osztálydeklaráció végéről a ';'-t!!!

Az előzőhöz hasonlóan hozzuk létre a Source Files-ok közé a Circle.cpp-t, és implementáljuk benne a SetCircleCenter tagfüggvényt!

```
// Circle.cpp : implementation of the CCircle class
//
////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "Circle.h"

CPoint CCircle::SetCircleCenter(CPoint point)
{
  CPoint oldCenter;
  oldCenter.x = m_x;
  oldCenter.y = m_y;
  m_x=point.x;
  m_y=point.y;
  return oldCenter;
}
```

- **Az új osztálynak megfelelően módosítsuk a programunk kódját!**

CKorDoc.h-ban:

```
#include "Circle.h"
…
// Attributes
public:
  CCircle m_Circle;   //m_x és m_y helyett.
```

Az OnDraw-ban

```
RECT rect;
 rect.left  = pDoc->m_x-20;
 rect.top   = pDoc->m_y-20;
 rect.right = pDoc->m_x+20;
 rect.bottom= pDoc->m_y+20;

 pDC->Ellipse(&rect);
```

kód helyett, csak `pDoc->m_Circle.Draw(pDC);` kell.

Az OnLButtonDown-ban

```
void CKorView::OnLButtonDown(UINT nFlags, CPoint point)
{
 // TODO: Add your message handler code here and/or call
default
 CKorDoc* pDoc = GetDocument();
 pDoc->m_Circle.SetCircleCenter(point);
 Invalidate();

 CView::OnLButtonDown(nFlags, point);
}
```

A ***CKorDoc::Serialize()***-ban a következő kód szerepelne, de protected adattagokra nem hivatkozhatunk, így ez nem fordul le. Nem módosítjuk a CCircle osztályt (public tag, GetX metódus vagy friend kapcsolat), mert később úgyis szerializációval akarjuk menteni a kört. Tehát írjuk be a második kódrészletet, ami ugyan most még nem működik!

```
void CKorDoc::Serialize(CArchive& ar)
{
 if (ar.IsStoring())
 {
    // TODO: add storing code here
    ar << m_Circle.m_x << m_Circle.m_y;
 }
 else
 {
    // TODO: add loading code here
    ar >> m_Circle.m_x >> m_Circle.m_y;
 }
}
```

A Serialize függvény így módosul, amennyiben megvalósítjuk a CCircle-ben a Serialize-t. Töröljük ki a fölöslegessé váló, s a környezet által begenerált elágazást, majd írjuk be a vastagon szedett részt!

```
void CKorDoc::Serialize(CArchive& ar)
{
 m_Circle.Serialize(ar);
}
```

Ebben az esetben persze a CObject osztály szerializációja hívódik meg, mely nem csinál semmit. (Az IsSerializable() hívással tesztelhetnénk a szerializálhatóságot.)

Megjegyzés:

> Ha a dokumentumosztályunk az m_Circle tagváltozón kívül tartalmazna olyan adattagot is, ami nem szerializálható, természetesen továbbra is szükségünk lenne az elágazásra. Mivel általában ez a helyzet, az alkalmazásvarázsló ezért készíti elő nekünk.

CKorDoc.cpp-ben: Az OnNewDocument-ben nem szükséges az m_x, m_y-nak a kezdő értékadás, de az m_Circle adattagjait be kell állítanunk! Ezzel feleslegessé váltak a default konstruktor paraméterei.

Tesztelés:

Ha az m_Circle adatait nem állítjuk be itt, hanem a kezdőértékadást a default konstruktorra hagyjuk, tesztelhetjük, hogy az új dokumentum nyitásakor nem hívódik meg a konstruktor. Mentsük el a kört az új helyén, majd nyissunk új dokumentumot! A kör változatlanul az előző mentés helyén marad.

Magyarázat: Nem írtuk felül a ***DeleteContents()*** metódust, mely törölné az adatokat, így azok értéke megmaradt. (Nem is kell felülírnunk, mert nincs Clear parancs és nincs olyan adattag, melynek értéke nem írható egyszerűen felül, pl. dinamikus tagváltozó.) Másrészt nem hívódott meg a konstruktor, az ***OnNewDocument()*** pedig nem végezte el a kezdőértékadást.

Írjuk be az OnNewDocument-be a kezdőértékadást!

```
BOOL CKorDoc::OnNewDocument()
{
 if (!CDocument::OnNewDocument())
    return FALSE;
 // TODO: add reinitialization code here
 // (SDI documents will reuse this document)
 m_Circle.SetCircleCenter(CPoint(50,50));
 return TRUE;
}
```

A kör sugarát nem módosítjuk. Ha megírjuk a módosításhoz szükséges metódust, természetesen azt is hívni kell itt a kezdőértékadáshoz.

A tesztelés során láthatjuk, a mentés és beolvasás kivételével működik a program.

➢ A szerializáció megvalósítása a CCircle osztályban

Az osztály deklarációjában **DECLARE_SERIAL** (CCircle) sorral deklaráljuk. A paraméter az osztály neve. A .cpp fájl elején pedig implementáljuk az **IMPLEMENT_SERIAL** (CCircle, CObject, 1) sorral. Paraméterek: az osztály neve, az ős neve, 1: sémaszám. Mentéskor ez az adat a mentett fájlba kerül, s csak az azonos sémaszámú adatokat tudja majd beolvasni. Így ha később módosítunk az adatszerkezeten és más adatokat kérünk, régi fájl nyitásakor nem értelmezi hibásan az adatokat.

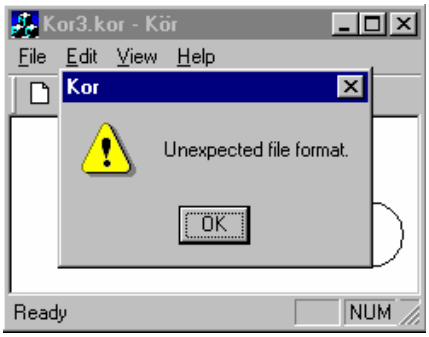
A deklarációban még meg is kell írnunk a szerializációt! A Circle.h-ban:

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

Implementációja a Circle.cpp fájlban:

```
void CCircle::Serialize (CArchive& ar)
{
 if (ar.IsStoring())
    ar << m_x << m_y << m_r;
 else
    ar >> m_x >> m_y >> m_r;
}
```

Teszteljük a programot! A szerializációval mentett esetekben jól működik, ha egy régebben mentett fájlt akarunk megnyitni, Unexpected file format hibaüzenetet kapunk. Tehát már most ellenőrzi az adatok helyes beolvasását a szériaszám.



6. Gyakorlat 7. ábra Nem szerializációval mentett fájl nyitása

6.1.3. Több kör tárolása

Ezt a szakaszt csak a template osztályok megismerése után dolgozza fel! A következő feladatok e szakasz végrehajtása nélkül is működnek.

➢ Szerializáció tömbben

Minden új egérkattintásra jelenjen meg az egérmutató által megjelölt középponttal és alapértelmezett sugárral egy új kör objektum! Az új ábrán tehát több kör is látható lesz, ezek mindegyikét mentéskor el kell tárolni, beolvasáskor a képernyőre kell rajzolni. Tároljuk az objektumokat egy **CArray** template osztályban!

Az osztály leírása az **afxtempl.h** fejlécfájlban található, így használatához a KorDoc.h elején be kell szerkesztenünk e fejlécfájlt!

```
#include <afxtempl.h>
```

Majd az osztály deklarációjába írjuk a vastagon szedett sorokat!

```
class CKorDoc : public CDocument
{
protected: // create from serialization only
 CKorDoc();
 DECLARE_DYNCREATE(CKorDoc)
    CTypedPtrArray<CObArray,CCircle*> m_CircleArray;
// Attributes
public:
//  CCircle m_Circle;

// Operations
public:
 void AddCircle(int x, int y, int r);
 CCircle* GetCircle(int i);
 int GetCircleCount();
//...
}
```

A függvények kódja:

```
void CKorDoc::AddCircle(int x, int y, int r)
{
 CCircle *pCircle=new CCircle(x,y,r);
 m_CircleArray.Add(pCircle);
}

CCircle* CKorDoc::GetCircle(int i)
{
 if (i < 0 || i > GetCircleCount()-1)
    return 0;
 return m_CircleArray.GetAt(i);
}

int CKorDoc::GetCircleCount()
{
 return m_CircleArray.GetSize();
}
```

Ezután a kattintásesemény kezelése is módosul:

A CKorView::OnLButtonDown-ban

```
 pDoc->AddCircle(point.x,point.y,20);
 Invalidate();
 pDoc->SetModifiedFlag();
```

A CKorView::OnDraw-ban:

```
for (int i=0; i<pDoc->GetCircleCount();i++)
pDoc->GetCircle(i)->Draw(pDC);
```

A szerializáció ilyen egyszerű:

```
void CKorDoc::Serialize(CArchive& ar)
{
    m_Circle Array.Serialize(ar);
}
```

A File menü New menüpontja hívásakor ugyan a File neve Untitled, de a kép nem lesz üres.

Ez a menüpont meghívja a dokumentumosztály virtuális ***DeleteContents()*** függvényét, melynek feladata a tartalom törlése. Írjuk ezt a függvényt fölül!

ClassWizard

Class Name :CKorDoc Object IDs: CKorDoc

Messages: DeleteContents

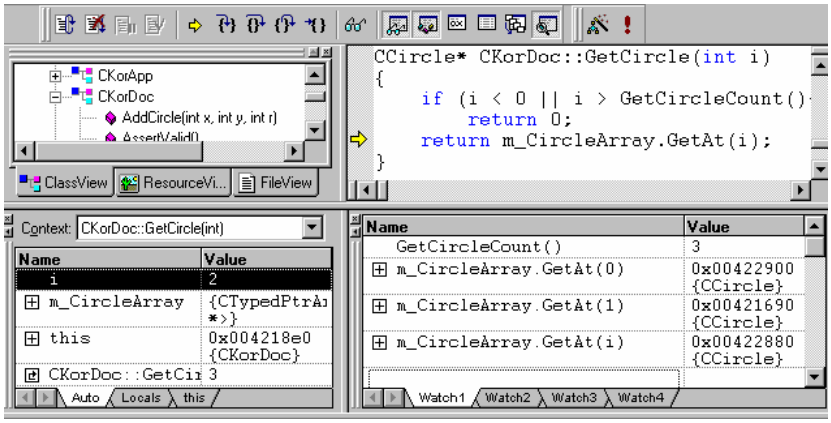
```
void CKorDoc::DeleteContents()
{
// TODO: Add your specialized code here and/or call the base
class
 int i = GetCircleCount();
 while (i)
    delete GetCircle(--i);
 m_CircleArray.RemoveAll();

 CDocument::DeleteContents();
}
```

Kövessük Debuggerrel a törlés folyamatát!

Állítsuk kurzorunkat az **int i = GetCircleCount();** sorra, majd Run to Cursor. A Watch ablakban kérdezzük le a GetCircleCount() értékét! (A kódban a függvényt kijelölve a Watch ablakba vonszolhatjuk.) Kezdetben 0. Tehát a while ciklusba nem lép be a program.

Állítsuk kurzorunkat ugyanerre a sorra, majd Run to Cursor-t választva a futó alkalmazásban vegyünk föl három kört! Majd válasszuk a File menü New menüpontját, hogy meghívjuk a függvényt!

Most a GetCircleCount hármat ad vissza. Figyeljük meg a Watch ablakban az m_CircleArray elemeit! (Ha nem létező elemet kérdezünk le, a debugger Assertion üzenetet küld. Ez természetes is, hiszen nem létező elemre hivatkoztunk. Ilyen esetben Ignore-ral menjünk tovább!) Lépjünk bele a delete végrehajtásába csak azért, hogy lássuk, a -- az i-re valóban még a törlés előtt végrehajtódott!



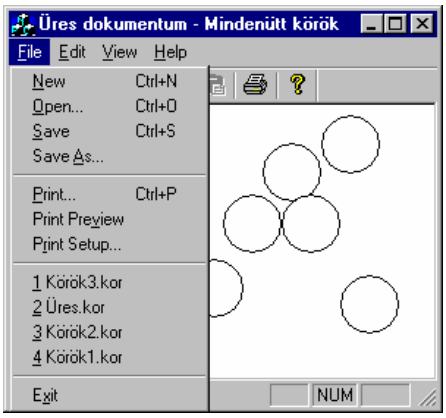
6. Gyakorlat 8. ábra A Kor alkalmazás a debuggerben

➢ Legyen a program címe: 'Mindenütt körök'!

Prject Workspace / Kor resources / Sring table / String table

Az első tartalmazza az ablak címét. Módosítsuk az ablak címét (Az **IDR_MAINFRAME**, Caption első eleme) "Mindenütt körök"-re!

Ha zavar benünket, hogy üres dokumentum esetén Untitled a felirat, módosítsuk az az OnNewDocument-ben egy `SetTitle("Üres dokumentum");` hívással! A tömbbe tárolt körök használata óta nincs szükségünk az m_Circle tagváltozóra a dokumentumosztályban, és nem kell kezdetben a képernyőre kitenni a kezdőkört, tehát az m_Circle.SetCircleCenter hívás fölöslegessé vált.



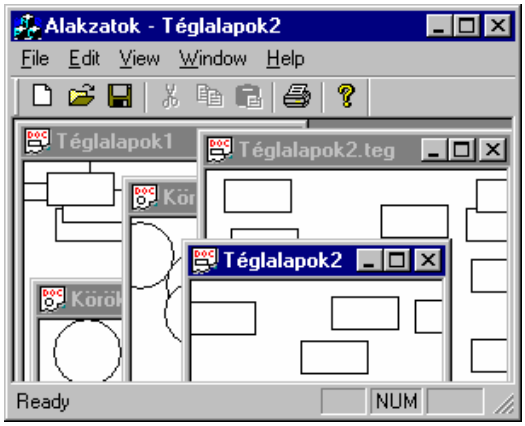
6. Gyakorlat 9. ábra A 'Kor' alkalmazás címsora

Próbáljuk ki, működik-e a drag & drop, nyissuk ki az explorert, és húzzuk bele az alkalmazásunk ablakába a .kor kiterjesztésű fájlokat! A .kor kiterjesztésű fájlokon duplán kattintva meghívódik az alkalmazás és betöltődnek a fájlban tárolt körök!

6.2. Mindenütt alakzatok

Készítsük el előző feladatunkat úgy, hogy különböző típusú fájljaink legyenek, s az alkalmazás mindegyiket tudja egyszerre megnyitni, kezelni! Ha köröket nyitunk meg, akkor kattintásra kört rajzoljon, ha téglalapokat, akkor azt!

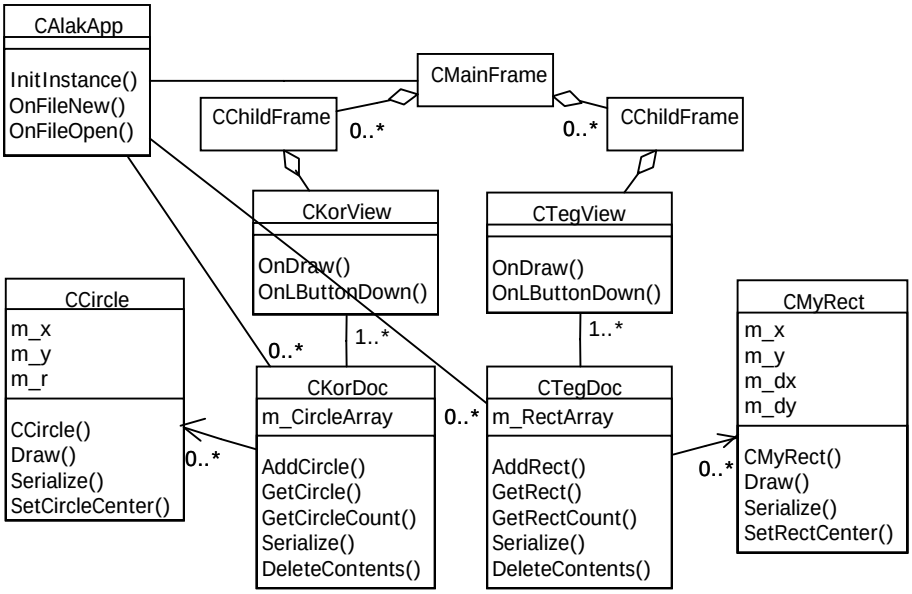
Ha a Window menü New Window menüpontját választjuk, az épp aktuális dokumentumot több példányban is megnyithatjuk, s a módosítások az összes példányon jelenjenek meg!



6. Gyakorlat 10. ábra Az ’Alakzatok’ alkalmazás, futás közben

Ehhez, mivel a dokumentumokat egyszerre kezeljük, MDI alkalmazásra lesz szükségünk!

➢ Az alkalmazás osztálydiagramja

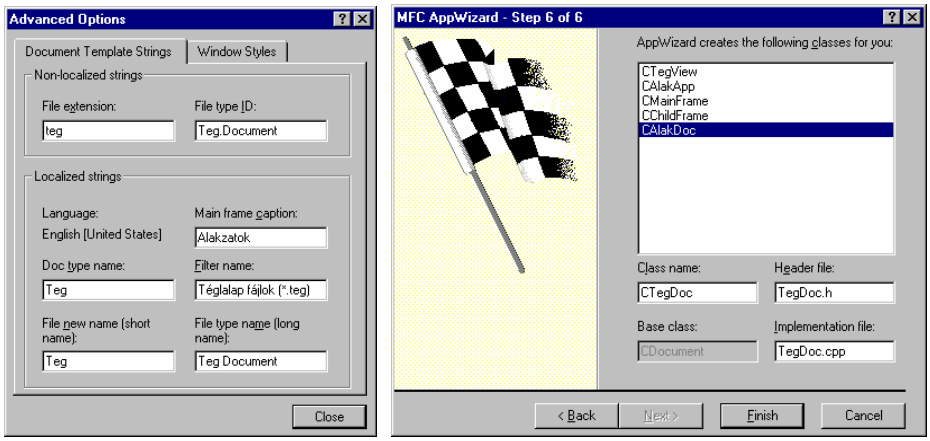


6. Gyakorlat 11. ábra Az Alakzatok alkalmazás osztálydiagramja

➢ A 'Mindenütt alakzatok' megvalósítása

Előző alkalmazásunk osztályait felhasználjuk a fejlesztéshez, és azok alapján hozzuk létre az új osztályokat is. Az osztályok metódusai is az előző feladathoz hasonló funkciót látnak el, így nem szükséges újra leírni őket. Az MDI sablon a fő keretablakon kívül gyermek-keretablakokkal is dolgozik. Minden egyes dokumentum megnyitásakor létrejön a keretablak - dokumentum - nézetablak hármas. Ha újabb ablakot nyitunk a dokumentumról, további keretablakok és nézetek jönnek létre.

Készítsünk egy 'Alak' nevű **MDI** alkalmazást, mely alapértelmezésben '.teg' kiterjesztésű 'Téglalap fájlok'-at hoz létre (4. lépés Advanced gomb)! A dokumentum- és nézetosztályának neve legyen TegDoc és TegView! Ezt az alkalmazásvarázsló 6. lépésében állíthatjuk be.



6. Gyakorlat 12. ábra A fájl és osztálynevek átállítása az alkalmazásvarázslóban

Teszteljük az üres alkalmazást! Egyszerre több ablakot is megnyithatunk, s a Window menü szokásos lehetőségei is rendelkezésünkre állnak.

➢ Több különböző dokumentumosztály kezelése

Ez egy **MDI alkalmazás**, egy dokumentumsablonnal, tehát minden dokumentum egy osztály példánya, de **egyszerre több dokumentumot is megnyithatunk**. A következőkben azon munkálkodunk, hogy alkalmazásunk többféle dokumentummal is dolgozni tudjon. Ezt egy SDI alkalmazásnál is megtehettük volna, mert az SDI alkalmazás is rendelkezhet több dokumentumsablonnal, csak egyszerre ezek közül egyet tud megnyitni. Az elv az új dokumentumsablon elkészítésére SDI alkalmazás esetén az itt következővel megegyező, csak a dokumentumsablon-objektum osztálya különbözik.

Másoljuk át az alkalmazásunk alkönyvtárába a KorDoc.cpp, KorDoc.h, a KorView.cpp, KorView.h, és a Circle.cpp, Circle.h fájlokat! Ha elkészítettük, akkor a res alkönyvtárból a KorDoc.ico fájlt is! (Eddigi tudásunkkal ez a saját osztályok újrafelhasználásának a módja.)

A Project menü / Add To Project / Files menüjében adjuk a projektünkhöz az előbb átmásolt .cpp és .h fájlokat! Az erőforrások közt az ikonoknál importáljuk be a KorDoc ikonunkat, és adjuk neki az IDR_KORTYPE azonosítót! Rajzoljuk meg az ikonokat!

Készítsünk újabb dokumentumsablon-osztályt! Ehhez szerkesszük be a fejlécfájlokat!

```
#include "KorDoc.h"
#include "KorView.h"
```

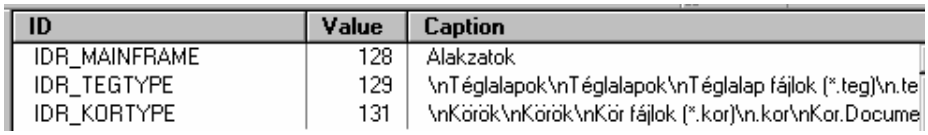
```
//...
BOOL CAlakApp::InitInstance()
{
//...
 CMultiDocTemplate* pDocTemplate;
 pDocTemplate = new CMultiDocTemplate(
    IDR_TEGTYPE,
    RUNTIME_CLASS(CTegDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CTegView));
 AddDocTemplate(pDocTemplate);
 pDocTemplate = new CMultiDocTemplate(
    IDR_KORTYPE,
    RUNTIME_CLASS(CKorDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CKorView));
 AddDocTemplate(pDocTemplate);
//...
}
```

A fordításkor a két hibaüzenetre, hogy nem tudja megnyitni a Kor.h include fájlt, mivel ezek az alkalmazásosztály fájljai, javítsuk a kódot: `#include "Alak.h"`-ra!

Készítsük el az IDR_KORTYPE erőforrásokat!

Hozzunk létre másolatot az IDR_TEGTYPE menüről (az azonosítót kijelölve, másolás beillesztés), majd nevezzük el IDR_KORTYPE-nak!

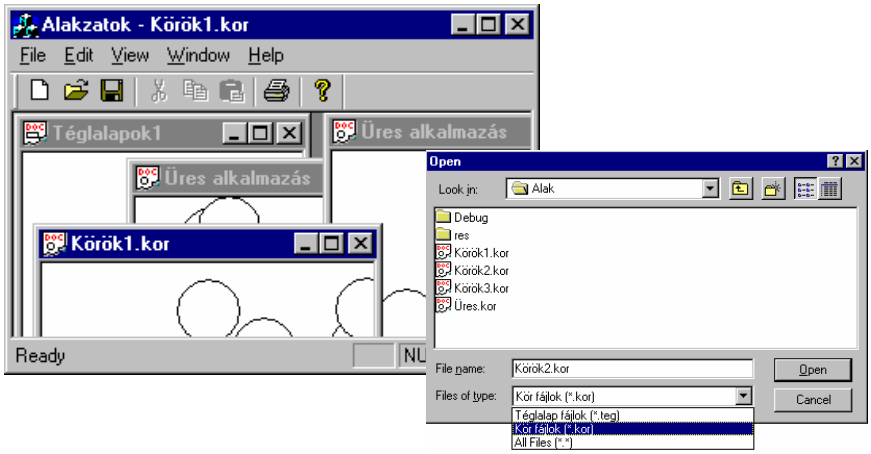
A String Table -be is készítsünk másolatot az IDR_TEGTYPE-ról, IDR_KORTYPE legyen a neve, és írjuk át a Teg szövegeket Körök-re! Az elmélet 6.2.1. szakasza alapján módosítsuk a szövegeket!



ID	Value	Caption
IDR_MAINFRAME	128	Alakzatok
IDR_TEGTYPE	129	\nTéglalapok\nTéglalapok\nTéglalap fájlok (*.teg)\n.te
IDR_KORTYPE	131	\nKörök\nKörök\nKör fájlok (*.kor)\n.kor\nKor.Docume

6. Gyakorlat 13. ábra A fájl és osztálynevek átállítása az alkalmazásvarázslóban

Ha futtatjuk az alkalmazást láthatjuk, hogy kör osztályaink választása esetén már működik is az alkalmazás, egérkattintás, mentés, beolvasás, csak most az üres dokumentum elnevezés nem a legszerencsésebb, hisz most egyszerre több dokumentum lehet nyitva ezzel a címsorral, ráadásul nem tájékoztat arról sem, hogy kört vagy téglalapot rajzol. (6. Gyakorlat 14. ábra) Szedjük ki a CKorDoc::OnNewDocument tagfüggvényből a SetTitle sort!

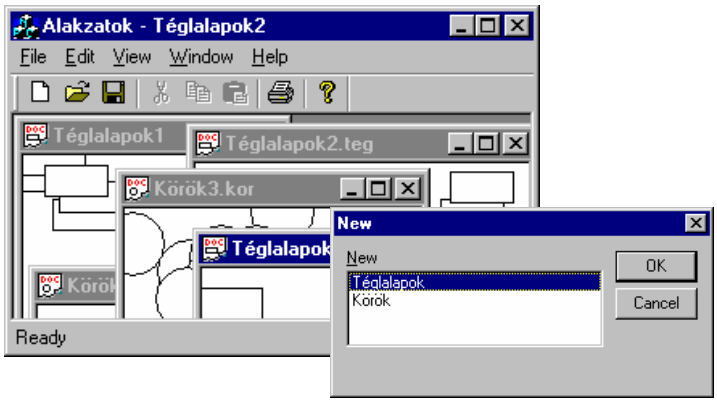


6. Gyakorlat 14. ábra A String Table beállításai és a kör dokumentumok kezelése működik

Most már nincs más teendőnk, minthogy a körök alapján elkészítsük a téglalap osztályainkat! Ne feledjük, CRect MFC osztály már van, nevezzük másként téglalap osztályunkat!

➢ Egy dokumentum több ablakban

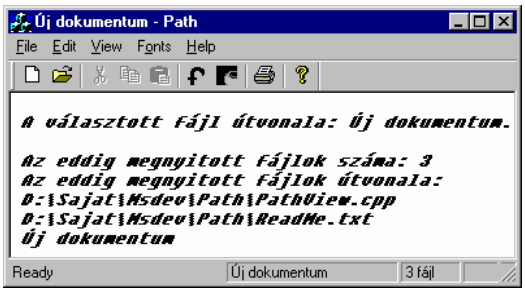
Teszteljük az alkalmazást! Próbáljuk meg ugyanazt a dokumentumot több ablakban megnyitni! Ez a **Window menü New Window menüpont**jával történhet. A megnyitott ablak a címsorában az ablak neve mögé írt ':2'-vel jelzi, ez a dokumentum második ablaka. Láthatjuk, hogy az új ablak mutatja a dokumentum adatait. Módosítsuk az adatokat! Az eredeti ablakon nem látjuk a módosítást! Ha újabb ablakot nyitunk a dokumentumra, az már a módosított adatokkal jelenik meg. Tehát a dokumentumobjektum tárolja a módosítást, csak a már megnyitott nézetek nem frissülnek. Az aktuális nézet kirajzolását a kattintás kezelése után az Invalidate() kényszeríti ki. Helyette frissítsük az összes nézetet az ***UpdateAllViews(NULL)*** metódushívással! (A módosítás az OnLButtonDown() kezelőjében történik mindkét nézetosztályban.)



6. Gyakorlat 15. ábra Az Alakzatok MDI alkalmazás

6.3. A keretablak interfészei

Készítsünk alkalmazást, mely kiírja az aktuálisan megnyitott fájl teljes elérési útvonalát a kliensterületre és az állapotsorba! Megszámolja az eddig megnyitott fájlok számát és kilistázza az indítás óta megnyitott fájlok teljes elérési útját. Legyen lehetőség a kiírás betűinek beállítására, a számolás és a lista újraindítására!



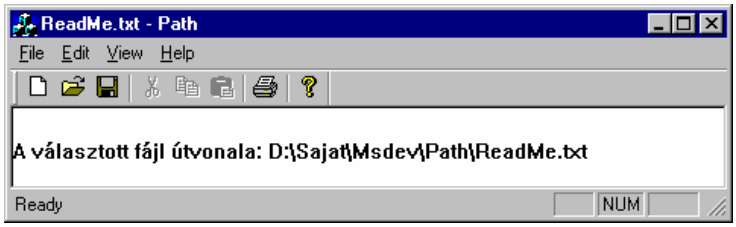
6. Gyakorlat 16. ábra A 'Path' alkalmazás futás közben

➢ **Készítsünk egy SDI alkalmazást!**

Neve legyen Path! Az első lépésben Single document, a továbbiak alapbeállításúak.

Írjunk szöveget az ablakba! ***CPathView::OnDraw()*** tagfüggvénye:

```
// TODO: add draw code for native data here
CString st=pDoc->GetPathName();
pDC->TextOut(0,20,"A választott fájl útvonala: " + st);
```



6. Gyakorlat 17. ábra A Path alkalmazás

Az st változó csak akkor kap értéket, ha választunk egy fájlt, mondjuk a Readme.txt-t a File menü Open menüpontjában. Ha azt akarjuk, hogy új dokumentum esetén az legyen kiírva: "Új dokumentum", módosítsuk a kódot!

```
// TODO: add draw code for native data here
CString st=pDoc->GetPathName();
if (st=="")
{
   st = "Új dokumentum.";
   AfxGetMainWnd()->SetWindowText("Új dokumentum - Path");
   // Címsor se legyen Untitled!
}
pDC->TextOut(0,20,"A választott fájl útvonala: "+st);
```

Próbáljuk ki, hogy a szöveg a nyomtatóra is nyomtatható!

! Ha megnyitunk egy fájlt, és mentjük is, akkor mivel a Serialize metódust nem írtuk át, az semmit nem ír a fájlba, tehát üres lesz az addig adatokkal feltöltött fájlból. Ezt kipróbálhatjuk a ReadMe.txt fájllal! Nézzük meg a tartalmát a Visual C++ fejlesztői környezetben, majd bezárása után nyissuk meg a 'Path' alkalmazásban! Mentsük el, és nézzük meg az eredményt a Visual C++ szövegszerkesztőjével! A fájl üres lett. Ezért tegyük lehetetlenné a mentést!

6.3.1. Menüpontok

- **Készítsünk egy új menüpontot, és töröljük a fölöslegeseket!**

Resource View

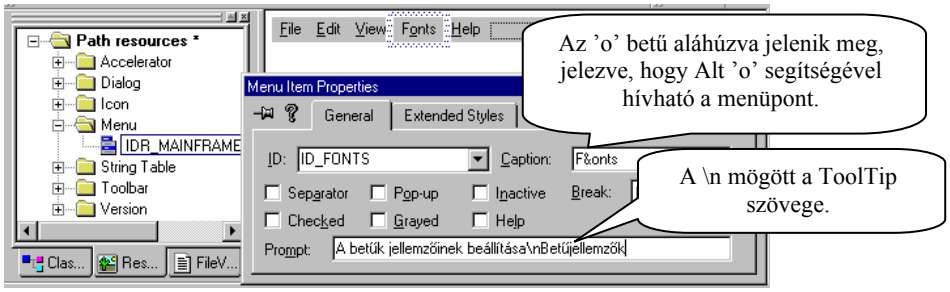
Menu

IDR_MAINFRAME

A File menü Save és Save as menüpontját kijelölve a Delete gomb segítségével töröljük őket! (A fenti figyelmeztetés miatt.)

Szeretnénk egy font típust módosító menüpontot! A főmenü új menüpontját a bal oldali egérgomb segítségével húzzuk az általunk választott helyre!

Kezdjük el írni a menü címét! Hatására lenyílik a Properties ablak, s mi épp a Caption beviteli mezőbe gépelünk. Az új menüpont neve legyen Fonts! Ne legyen legördülő menü (pop-up), és töltsük ki a Prompt mezőt is, hogy legyen információ a menüről az alsó üzenetsorban!



6. Gyakorlat 18. ábra A Menü szerkesztése

Rendeljünk a menüpontunkhoz egyenlőre egy üzenetablakot az osztályvarázsló segítségével! (Pl. Ctrl Dupla klick a menüponton.)

El kell döntenünk, melyik osztály kezelje a menüpontot! Mivel a fontokra a megjelenítéskor a View osztályban van szükség, ezt az osztályt állítsuk be az osztályvarázslóban!

Class name: **CPathView** class,

ID: ID_FONTS,

Messages: COMMAND

Add Function: OnFonts,

Edit Code: MessageBox("Fonts");

Teszteljük a programot! Vizsgáljuk meg Alt 'o' hatására is megjelenik-e az üzenetablak!

6.3.2. Az eszköztár

- **Módosítsuk az eszköztárat!**

Töröljük a Save eszközgombot úgy, hogy ikonját egyszerűen húzzuk le az eszköztárról!

Resource View

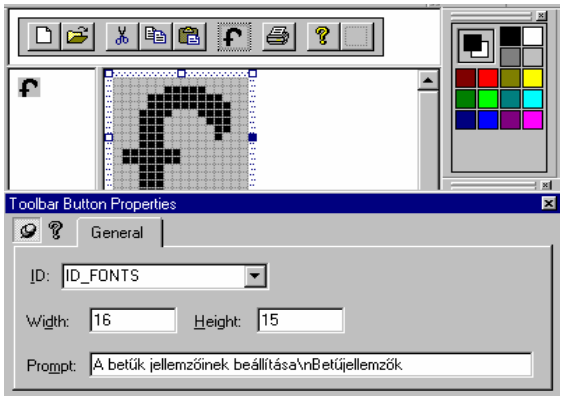
Toolbar

IDR_MAINFRAME

Helyezzünk el egy Fonts gombot az eszköztárban! Azt akarjuk, hogy az ikon választása esetén a Fonts menüponthoz rendelt parancs hajtódjon végre!

Duplán kattintva az új ikonon az eszköztárban, hozzá rendelhetjük a Properties ablak ID legördülő listájából az ID_FONTS azonosítót! Amint a rendszer azonosította az ID-t, ha újra kinyitjuk a tulajdonságablakot, már a Prompt dobozában ott a Fonts menüpont üzenete.

Az új ikont választva képe szerkeszthető. Drag & drop technikával tegyük a kívánt helyre az eszköztárban!



6. Gyakorlat 19. ábra Az eszköztár szerkesztése

➢ Működjön a Fonts menüpont nevének megfelelően!

Mely osztály feladata a font tárolása? A View osztályé. Vegyünk fel egy **CFont** típusú adattagot a CPathView osztályba! Nevezzük m_Font-nak! Ez a változó őrzi majd az aktuális font típusát. Az adattagok kezdő-értékadásának helye az osztály konstruktora. Tehát:

```
CPathView::CPathView()
{
 m_Font.CreateStockObject( SYSTEM_FONT );
}
```

A CFont osztály a **CGdiObject** utódosztálya, így hívhatjuk a ***CreateStockObject()*** tagfüggvényt **SYSTEM_FONT** argumentummal. Alapértelmezésben a Windows a SYSTEM_FONT-ot használja menük, párbeszédablak vezérlők és egyéb feliratok

elkészítéséhez. Ezzel hozzájuthatunk az operációs rendszer által biztosított fonthoz (system font).

Most írjuk meg a menühöz tartozó parancskezelőt!

```
void CPathView::OnFonts()
{
//  MessageBox("Fonts");
  CFontDialog fontDlg;
  if (fontDlg.DoModal()==IDOK);
}
```

A **CFontDialog** osztály a CDialog utódosztálya, így hívható a ***DoModal()*** tagfüggvény, mely OK gomb választása esetén **IDOK**-val tér vissza. A CFontDialog osztály biztosítja a font kiválasztásának minden kényelmét. Próbáljuk ki a működő programot!

Természetesen még nem tároltuk és vettük figyelembe a font típusát!

Nézzük meg a **CFontDialog** konstruktorát a súgóban! Átadható neki egy **LPLOGFONT** típusú argumentum, tehát egy **LOGFONT** típusú mutató. (A LOGFONT struktúra tartalmazza a font jellemzőit.) Ezen a változón dolgozik a párbeszédablak.

Vegyünk fel egy LOGFONT típusú lokális változót, majd írjuk bele a meglevő m_Font adattagunkban tárolt fontot! Ezt a CFont osztály ***GetLogFont()*** tagfüggvényével tehetjük.

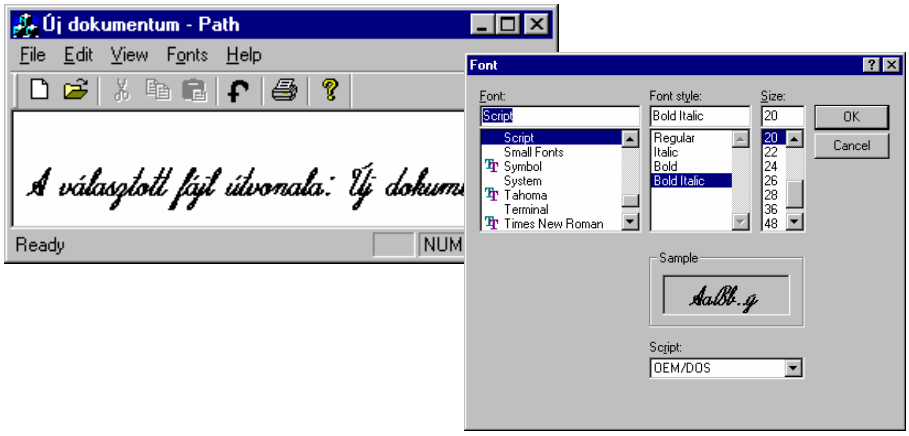
Az adatok visszamásolása a ***CFont::CreateFontIndirect()*** tagfüggvénnyel történik.

```
void CPathView::OnFonts()
{
// MessageBox("Fonts");
  LOGFONT logFont;
  m_Font.GetLogFont(&logFont);
  CFontDialog fontDlg(&logFont,CF_SCREENFONTS |
                                    CF_INITTOLOGFONTSTRUCT);
  if (fontDlg.DoModal()==IDOK)
  {
     m_Font.DeleteObject();
     m_Font.CreateFontIndirect(&logFont);
     Invalidate();
  }
}
```

Az itt beállított fontot használjuk kiírásunkkor!

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

```
CFont* pOldFont = pDC-> SelectObject(& m_Font);
TEXTMETRIC tm;
pDC->GetTextMetrics(&tm);
int lineHeight=tm.tmHeight+tm.tmExternalLeading;

CString st=pDoc->GetPathName();
if (st=="")
{
   st = "Új dokumentum.";
   AfxGetMainWnd()->SetWindowText("Új dokumentum - Path");
   // Címsor se legyen Untitled.
}
pDC->TextOut(5,lineHeight,"A választott fájl útvonala: "+ st);
pDC-> SelectObject(pOldFont);
}
```



6. Gyakorlat 20. ábra A font kiválasztása működik

➢ Az eddig megnyitott fájlok száma

Ahhoz, hogy e feladatot megvalósítsuk, tárolnunk kell az eddig megnyitott fájlok számát. Ez a dokumentumosztály felelőssége! Vegyünk fel a Class View gyorsmenüje segítségével egy private int m_FileCount adattagot a CPathDoc osztályba! Mivel a kiírásnál a View osztályból hozzá szeretnénk férni ehhez az értékhez, írjunk hozzá egy public GetFileCount metódust! Ha a megnyitott fájlok számát más osztályból nem engedjük módosítani, SetFileCount tagfüggvényt ne írjunk!

```
class CPathDoc : public CDocument
{
//…
// Implementation
public:
 int GetFileCount() const { return m_FileCount; };
//...
private:
 int m_FileCount;                //a fájlok száma
};
```

Állítsuk be a kezdőértéket a konstruktor felülírásával! Mivel fájl megnyitásakor és új dokumentum létrehozásakor fogjuk növelni a megnyitott fájlok számát, így az ***OnNewDocument()*** és az ***OnOpenDocument()*** metódusokban tesszük ezt. Ha előrelátóak vagyunk a feladat továbbfejlesztése érdekében nem az ***Invalidate()*** hívással frissítjük az ablakot, hanem az ***UpdateAllViews(NULL)*** segítségével, így ha később több nézetablakunk lesz, azok mindegyike frissülni fog.

```
BOOL CPathDoc::OnNewDocument()
{
 if (!CDocument::OnNewDocument())
    return FALSE;

 // TODO: add reinitialization code here
 // (SDI documents will reuse this document)
 m_FileCount++;
 UpdateAllViews(NULL);
 return TRUE;
}
```

Az OnOpenDocument virtuális metódust az osztályvarázsló segítségével tudjuk felüldefiniálni.

```
BOOL CPathDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
 if (!CDocument::OnOpenDocument(lpszPathName))
    return FALSE;
 // TODO: Add your specialized creation code here
 m_FileCount++;
 UpdateAllViews(NULL);
 return TRUE;
}
```

A kezdőértékadás a konstruktor feladata. Ha az alkalmazás elindítását nem tekintjük fájlnyitásnak, akkor az m_FileCount kezdőértékét -1-re kell állítanunk, mert nyitáskor az OnNewDocument növeli ezt az értéket. Tehát így lesz kezdetben a megnyitott fájlok száma 0.

```
CPathDoc::CPathDoc()
{
 // TODO: add one-time construction code here
 m_FileCount=-1;
 // Indításkor az OnNewDocument végrehajtódik és növeli az
értékét!
}
```

Megjelenítés a képernyőn. Ez a View osztály OnDraw feladata.

```
void CPathView::OnDraw(CDC* pDC)
{
 //…
 pDC->TextOut(5,lineHeight,"A választott fájl útvonala: "+ st);
 CString sCount;
 sCount.Format("Az eddig megnyitott fájlok száma: %d",
                                        pDoc->GetFileCount());
 pDC->TextOut(5, 3*lineHeight, sCount); // Egy sor kimarad.
 pDC-> SelectObject(pOldFont);
}
```

➢ Az útvonalak tárolása

Az útvonalak tárolása egy **CStringList** típusú konténerben történik. A CPathDoc osztály tagváltozói közé vegyünk fel egy CStringList m_PathList private adatmezőt, és írjunk hozzá egy GetPathList() public metódust!

```
class CPathDoc : public CDocument
{
// ...
// Implementation
public:
 int GetFileCount() const {return m_FileCount;};
 CStringList& GetPathList() {return m_PathList;};
// ...
private:
 CStringList m_PathList;
 int m_FileCount;
};
```

A konstruktor létrehozza az üres listát, melyet feltölteni új dokumentum létrehozásakor (File menü New menüpont), illetve új dokumentum nyitásakor kell (File menü Open menüpont). Tehát ezek kezelőit kell módosítanunk.

Az OnNewDocument()-ben:

Az m_FileCount++; után, csak akkor adjuk hozzá a listához az Új dokumentum szöveget, ha nem az első dokumentumot nyitottuk meg.

```
BOOL CPathDoc::OnNewDocument()
{
 if (!CDocument::OnNewDocument())
    return FALSE;

 // TODO: add reinitialization code here
 // (SDI documents will reuse this document)
 m_FileCount++;
 if (m_FileCount) m_PathList.AddTail("Új dokumentum");
 UpdateAllViews(NULL);
 return TRUE;
}
```

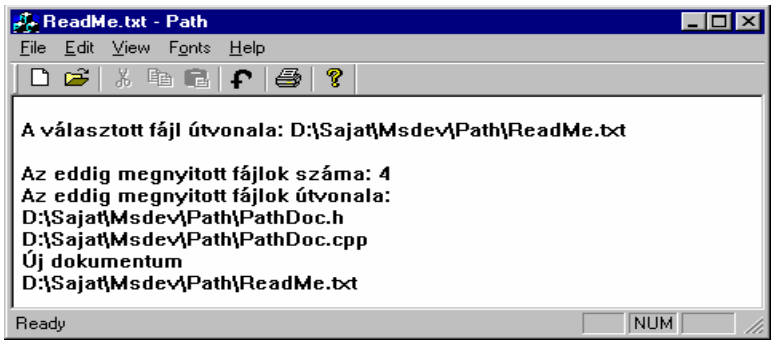
Az OnOpen Document() tagfüggvényben:

```
m_FileCount++;
m_PathList.AddTail(lpszPathName);
 //A GetPathName() még az előzőt adja!
```

Az adatok megjelenítése:

```
void CPathView::OnDraw(CDC* pDC)
{
 CPathDoc* pDoc = GetDocument();
 ASSERT_VALID(pDoc);
 // TODO: add draw code for native data here
 CFont* pOldFont = pDC-> SelectObject(& m_Font);
 TEXTMETRIC tm;
 pDC->GetTextMetrics(&tm);
 int lineHeight=tm.tmHeight+tm.tmExternalLeading;

 CString st=pDoc->GetPathName();
 if (st=="")
 {
    st = "Új dokumentum.";
    AfxGetMainWnd()->SetWindowText("Új dokumentum - Path");
    // Címsor se legyen Untitled.
 }
 pDC->TextOut(5,lineHeight,"A választott fájl útvonala: "+ st);
 CString sCount;
 sCount.Format("Az eddig megnyitott fájlok száma: %d",
                                        pDoc->GetFileCount());
 pDC->TextOut(5, 3*lineHeight, sCount); // Egy sor kimarad.
 pDC->TextOut(5, 4*lineHeight, "Az eddig megnyitott fájlok
                                              útvonala:");
```

```
  if (!pDoc->GetPathList().IsEmpty())
  {
     int i=5;   // Az 5. sorban kezdjük kiírni.
     for (POSITION pos=pDoc->GetPathList().GetHeadPosition();
                                                  pos != NULL;)
     {
         st=pDoc->GetPathList().GetNext(pos);
         pDC->TextOut(5,lineHeight*i++,st);
     }
  }
  pDC-> SelectObject(pOldFont);
}
```



6. Gyakorlat 21. ábra A lista a képernyőn

➢ Clear All menüpont

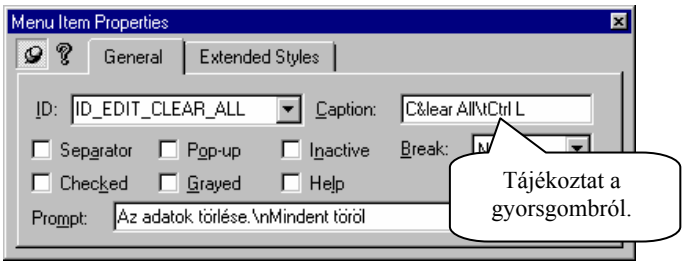
Tegyünk új almenüpontot az Edit menü legördülő almenüjébe! Neve legyen Clear All!

Resource View

Menu

IDR_MAINFRAME

A menüerőforrásból válasszuk az Edit menüpontból legördülő almenü új mezőjét! Tegyünk be egy elválasztót (Separator)! A szöveg írása automatikusan tölti a Properties Caption mezőjét. Az azonosítók listájából válasszuk az ID_EDIT_CLEAR_ALL-t!



6. Gyakorlat 22. ábra A Clear All menü tulajdonságlapja

A menüpont, amíg nincs kezelő metódus csatolva hozzá, nem választható.

Az osztályvarázsló segítségével adjuk meg a parancskezelőt! Ctrl duplaklikk / **CPathDoc** osztály, mert annak adatain dolgozunk / **COMMAND** / OnEditClearAll().

```
void CPathDoc::OnEditClearAll()
{
 OnNewDocument();
 m_PathList.RemoveAll();
 m_FileCount=0;
 UpdateAllViews(NULL);
}
```

6.3.3. Gyorsgombok

- **Rendeljünk gyorsgombot (accelerator key) a menüponthoz!**

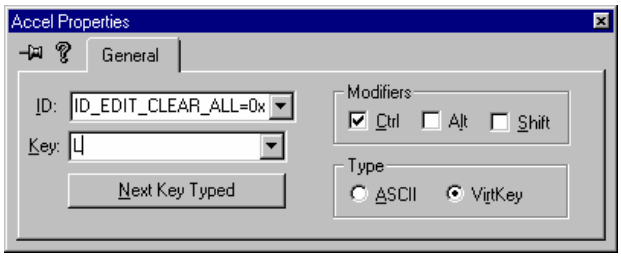
Almenükben szokás – a menüpont gyorsabb elérése érdekében – a menüponthoz gyorsgombot rendelni. A menüpont mellé a Properties Caption-be egy tabulátorjel (\t) után be szokás írni tájékoztatásul a billentyűkombinációt, pl. Ctrl L!

Resource View

Accelerator

IDR_MAINFRAME

Az új mezőre duplán klikkelve: Kitöltjük a Properties ablakot. Az azonosítót a legördülő listából választhatjuk, az Alt, Ctrl, Shift gombokat kijelölve, csak a betűt kell beírnunk.

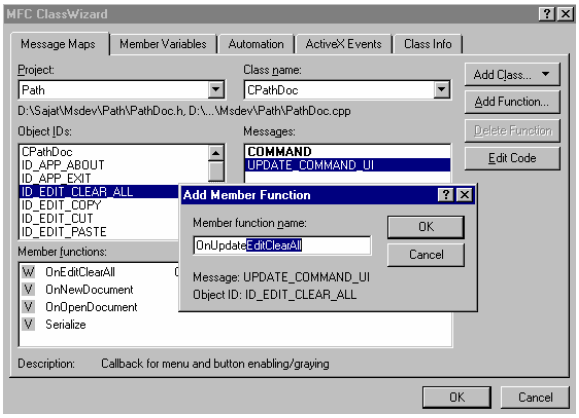


6. Gyakorlat 23. ábra Gyorsgomb a Clear All menüponthoz

Teszteljük a gyorsgombot!

➢ Az Edit-Clear All menüpont ne legyen választható, ha nincs mit töröljünk!

Mivel a dokumentumosztály adata alapján dönthető el engedélyezett-e a menüpont, ebbe az osztályba írjuk a kezelőjét!



6. Gyakorlat 24. ábra Tiltás és engedélyezés

```
void CPathDoc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
  pCmdUI->Enable(GetFileCount());
}
```

➢ Új eszközgomb az új menüponthoz!

Készítsünk az eszköztárba egy új gombot, mely ezt a parancsot hajtja végre! Nem kell mást tennünk, csak az ikon azonosítóját az ID_EDIT_CLEAR_ALL-ra kell állítani. A gomb a menüponttal együtt lesz engedélyezve vagy tiltva.

Használjunk egy meglevő bitmapet a gomb létrehozásához! A bitmap egy részét kijelölve a vágólapon át az eszközgombra másolhatjuk. Az eszközgomb szokásos mérete: 16x15. Ha nagyobb a kijelölt kép, csak egy része kerül a gombra, vagy a kép összenyomható. A gombok az egyik eszköztárból a másikba másolhatók.

6.3.4. Az állapotsor

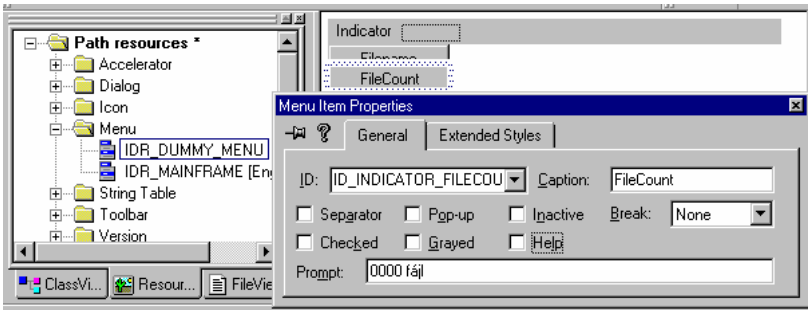
➢ Új ablak az állapotsorban!

A megnyitott fájlok számát mutassa az **állapotsor** (status bar)!

A **státuszsor** ablakainak számát és sorrendjét a MainFrm.cpp **indicators** tömbje határozza meg. Vegyünk fel két új elemet, az aktuális fájlnevet kiíró ablaknak és a fájlok számát tartalmazó ablaknak!

```
static UINT indicators[] =
{
  ID_SEPARATOR,           // status line indicator
  ID_INDICATOR_FILENAME,
  ID_INDICATOR_FILECOUNT,
  ID_INDICATOR_CAPS,
  ID_INDICATOR_NUM,
  ID_INDICATOR_SCRL,
};
```

Az azonosítókhoz adatokat is rendelnünk kell! Az osztályvarázsló nem biztosítja az állapotsor (status bar) közvetlen kezelését. Ezt vagy a String Table-ben tehetjük meg, vagy egy dummy menüt hozunk létre. Mivel az ablakokba a kiírást más osztályokból kell megvalósítanunk, a legegyszerűbb az üzenetkezelő mechanizmus útján elérni a szövegek kiírását. Ehhez pedig a legkényelmesebben a statiszta menü segítségével juthatunk el. (A dummy menü is létrehozza a String Table-ben a saját azonosítóit. Nézzük meg!)



6. Gyakorlat 25. ábra Dummy menü

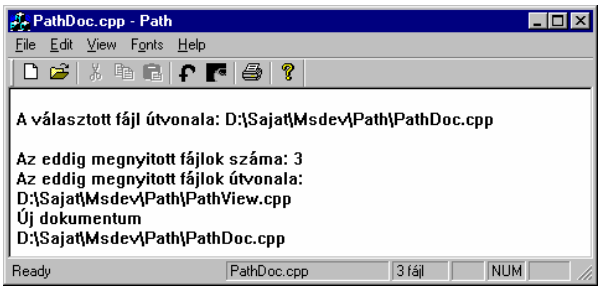
Ha az osztályvarázslóval kezelőt szeretnénk rendelni a menüpontokhoz, az Adding a Class ablak nyílik meg először, mely tájékoztat, hogy a DUMMY_MENU egy új erőforrás. Ne rendeljünk hozzá osztályt! Cancellel kiléphetünk az ablakból.

Vegyük fel a statisztamenüben a menüpontokat, a megfelelő azonosítóval (6. Gyakorlat 25. ábra)! Ne feledjük kitölteni a prompt sort! Ez adja az ablak méretét.

A kezelőfüggvényeket mi döntjük el, melyik osztályba helyezzük. Mivel az adatok tárolását a dokumentumosztály végzi, tegyük abba az **UPDATE_COMMAND_UI** kezelőket!

```
void CPathDoc::OnUpdateIndicatorFilecount(CCmdUI* pCmdUI)
{
 CString sFileCount;
 sFileCount.Format("%d fájl", m_FileCount);
 pCmdUI->SetText(sFileCount);
}

void CPathDoc::OnUpdateIndicatorFilename(CCmdUI* pCmdUI)
{
 if (GetTitle()=="Untitled")      // Magyarul írja ki!
    pCmdUI->SetText("Új dokumentum");
 else
    pCmdUI->SetText(GetTitle());
 pCmdUI->Enable(m_FileCount);
 // Csak ha van megnyitott akkor működjön!
}
```

A tesztelés során ellenőrizzük az Edit menü Clear All parancsának hatására is helyesen működik az állapotsor!



6. Gyakorlat 26. ábra Működik az állapotsor

Megjegyzés:

Ha a feladat futtatása során a 'Failed to create empty document' hibaüzenettel találkozunk, először nézzük meg, van-e mérete a a státuszsor ablakainak. Vagyis írtunk-e értéket az azonosítóhoz

tartozó menüpont tulajdonságlapján a prompt sorba. Ha igen, akkor az is előfordulhat, hogy az adott operációs rendszer (pl. Win 98) nem tudja kezelni a Dummy menüt. Ez esetben kézzel kell megírnunk a kezelőfüggvényeket.

Töröljük ki a menüpontokhoz rendelt UPDATE_COMMAND_UI kezelőket, (a kód maradhat) és az erőforrások közül a dummy menüt.

A CPathView osztályban deklaráljuk a két üzenetkezelő függvényt, írjuk be őket az //}}AFX_MSG és a DECLARE_MESSAGE_MAP() közé!

PathView.h:

```
//}}AFX_MSG
afx_msg void OnUpdateIndicatorFilecount(CCmdUI*pCmdUI);
afx_msg void OnUpdateIndicatorFilename(CCmdUI* pCmdUI);
DECLARE_MESSAGE_MAP()
```

Ezután a frissítés üzenethez rendeljük hozzá a kezelőfüggvényeket az üzenettérképben!

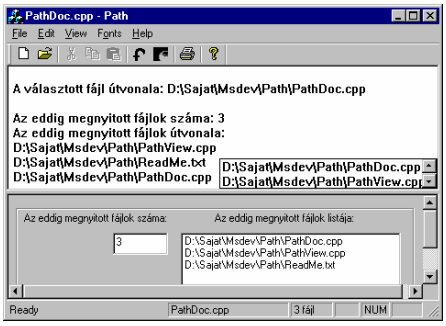
PathView.cpp:

```
//}}AFX_MSG_MAP
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_FILECOUNT,
      OnUpdateIndicatorFilecount);
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_FILENAME,
      OnUpdateIndicatorFilename);
END_MESSAGE_MAP()
```

Ha a függvények kódját nem töröltük ki, készen vagyunk.

6.4. Egy dokumentum, több nézet

Dokument / View architektúra esetén lehet több ablakkal is dolgozni! Erre kényelmes, a framework által szolgáltatott lehetőséget nyújtott az MDI alkalmazás (6.2.Gyakorlat). Lehetőségünk van, bár nem olyan kényelmesen, mint a párbeszédablaknál, hogy a View ablakunkba további ablakok is kerüljenek és arra is, hogy egy adott dokumentumosztályhoz több azonos vagy különböző típusú nézetablakot hozzunk létre.



6. Gyakorlat 27. ábra Több nézetablak

Folytassuk az előző programot a 6.3. gyakorlat feladatát!

Vegyünk fel a nézetablakunkba egy listadobozt, mely az eddig megnyitott fájlok listáját ábécérendben tárolja! Készítsünk dinamikus splitView-t, majd vegyünk fel egy dinamikus splitView-t különböző nézetekkel! Végül több különböző nézetben jelenítsük meg adatainkat!

6.4.1. Egy nézet több ablak

A listát egy listadobozban is meg szeretnénk jeleníteni. A CListBox a CWnd utódosztálya.

➢ A listadoboz létrehozása

A CPathView osztályban hozzunk létre egy **CListBox** típusú adatmezőt! (*CListBox m_ListBox)* Ne felejtsük el, a konstruktor csak a C++ objektumot hozza létre, a system ablak a ***Create()*** hívásakor keletkezik.

A listadoboz gyermekablaka lesz a nézetablaknak. A View osztály system ablaka a **WM_CREATE** üzenet hatására az ***OnCreate***()-ben leírt módon jön létre. Ebben kell meghívnunk az m_ListBox.Create() függvényt! Vagyis csatoljunk a CPathView osztályhoz egy WM_CREATE üzenetkezelőt!

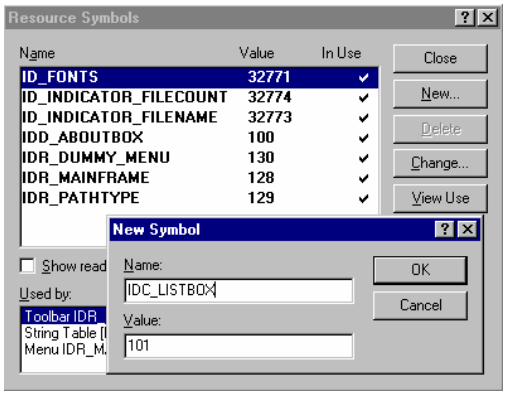
```
int CPathView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
 if (CView::OnCreate(lpCreateStruct) == -1)
    return -1;

 // TODO: Add your specialized creation code here
 m_ListBox.Create(WS_CHILD|WS_VISIBLE|WS_CLIPSIBLINGS|
    LBS_STANDARD|LBS_NOINTEGRALHEIGHT|LBS_DISABLENOSCROLL,
                    CRect(0,0,10,10),this, IDC_LISTBOX);
 return 0;
}
```

A CListBox::Create paraméterei:

- **Első paraméter a stílus**, mely részben az ablakosztály, részben a ListBox stílusa lehet. Javaslat: WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS (Az átlapolt szomszédos ablakok területére ne rajzoljon rá!) | LBS_STANDARD | LBS_NOINTEGRALHEIGHT | LBS_DISABLENOSCROLL (Mutatja a gördítősávot akkor is, ha nincs elég elem a görgetéshez. Nem engedélyezi a scrollozást míg meg nem telik.)
- **Második paraméter a listadoboz mérete** és helyzete a főablak kliens területéhez viszonyítva. Használjunk egy tetszőleges téglalapot, mert a

méretet majd igazítani akarjuk a változó ablakmérethez, s ezt a **WM_SIZE** kezelője teszi majd meg! CRect(0, 0,10,10).

- **Harmadik paraméter a szülőablak,** ebben az esetben a view objektum, tehát **this**.
- **Negyedik paraméter a listadoboz azonosítója**, az ID. Hozzunk létre egy új szimbólumot View menü, Resource Symbols parancs! New gomb, és gépeljük be a nevet! Javaslat: IDC_LISTBOX, mert a gyermekablakok szokásos ID kezdése IDC (control / window). A számértéket ne módosítsuk!



6. Gyakorlat 28. ábra Új azonosító felvétele

Az új ablak mérete változzon együtt a főablakkal! Írjunk kezelőt a View osztály WM_SIZE üzenetéhez! Az ***OnSize()*** kezelőnek van két paramétere: cx és cy. Megadják a kliensterület méretét. Számítsuk ebből az m_ListBox objektumra a ***MoveWindow()*** tagfüggvényének harmadik és negyedik paraméterét!

```
m_ListBox.MoveWindow(230,100,cx-230,cy-100);
```

Ellenőrizzük, a listadoboz a kerettel együtt változtatja-e méretét! (Látszik-e a gördítősáv?)

➢ A lista megjelenítése a listadobozban

Az **OnDraw()** tagfüggvény ciklusa előtt: m_ListBox.***ResetContent()***; üríti a listadobozt, a ciklusban az m_ListBox.***AddString***(st); pedig újra felveszi az elemeket.

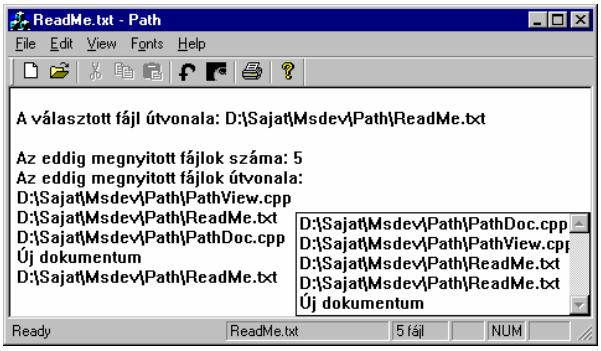
```
void CPathView::OnDraw(CDC* pDC)
{
 //...
 if (!pDoc->GetPathList().IsEmpty())
 {
    int i=5;   // Az 5. sorban kezdjük kiírni.
    m_ListBox.ResetContent();
    for (POSITION pos=pDoc->GetPathList().GetHeadPosition();
                                                pos != NULL;)
    {
        st=pDoc->GetPathList().GetNext(pos);
        pDC->TextOut(5,lineHeight*i++,st);
        m_ListBox.AddString(st);
    }
 }
 else m_ListBox.ResetContent();

 pDC-> SelectObject(pOldFont);
}
```

A megjelenítés a nézetosztály feladata, ezért itt töltjük fel a listadobozt a tárolt adatokkal. (Nem tehetjük meg, hogy csak az utolsó elemet adjuk a listához, mert az OnDraw máskor is hívódik, nem csak az ***UpdateAllViews()*** hatására, s így torz kijelzést adna.) Egy másik, sok listaelem esetén hatékonyabb megoldás: az ***OnUpdate()***-ben feltölteni a listát. Erre látunk példát később a FormViewnál.

Felmerül az a megoldás is, hogy a CPathDoc::***OnNewDocument()*** és ***OnOpenDocument()*** metódusában hozzáadni mindig a következő elemet, és az ***OnEditClearAll()*** metódusban kiüríteni. Ez utóbbi eset viszont feltételezi, hogy egy ilyen nézetünk van, s az az első. Ha több nézetet veszünk fel, és nem az első vagy nem csak az első nézet lesz ez az ablak, akkor az összes nézetre módosítni kell a két függvény kódját. Tehát kifizetődő lehet fejben tartani. A megjelenítés a nézetosztály feladata.

A listadoboz rendezetten jeleníti meg az adatokat.

A listadoboz fontbeállítása: CPathView::OnFonts() IDOK elágazásában: m_ListBox.SetFont(&m_Font);



6. Gyakorlat 29. ábra A listadoboz rendezetten tárolja a fájlneveket

➢ Írás átlapolt gyermekablakokban

Ha két egymás fölötti gyermekablakunk van, s egymás után írunk rájuk, mindkét ablakot felülírja a rendszer.

Ha olyan fontot állítunk be, hogy a View szövege a listadoboz alá csúszik, és mozgatjuk az ablakméretezőt, egy kis villogás most is látszik, de módosítsuk a CPathView::OnDraw kódját a következőre:

```
for (POSITION pos=pDoc->GetPathList().GetHeadPosition();
                                                pos != NULL;)
{
    st=pDoc->GetPathList().GetNext(pos);
    pDC->TextOut(5,lineHeight*i++,st);
    m_ListBox.AddString(st);
}
    Sleep(1000);
```

Futáskor (különösen, ha a betűméret elég nagy) jól látható az egymás után egymás fölé írás.

Megoldás: Az ablak létrehozása előtt állítsuk be a **WS_CLIPCHILDREN** jellemzőt!

```
BOOL CPathView::PreCreateWindow(CREATESTRUCT& cs)
{
  // TODO: Modify the Window class or styles here by modifying
  //  the CREATESTRUCT cs
  cs.style |= WS_CLIPCHILDREN;
  return CView::PreCreateWindow(cs);
}
```

Teszteléskor jól látható a különbség! A takart felületet nem írja felül az alsó ablaknak küldött szöveg. A tesztelés után a Sleep(1000); sort kitörölhetjük az OnDrawból!

6.4.2. Legyen több megosztott nézet!

➢ Dinamikusan megosztott nézet

Több nézetet a splitterwindow-val is létrehozhatunk. A nézetek helye a keretablak kliensterülete. A CMainFrame osztályban vegyünk tehát fel egy új adattagot:

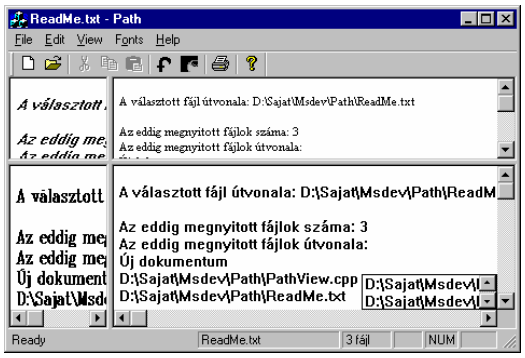
```
public:
   CSplitterWnd m_wndSplitter;
```

A view-kat az ***CMainFrame::OnCreateClient()*** hozza létre. Az osztályvarázsló segítségével ezt a virtuális metódust kell felüldefiniálnunk!

Ha dinamikus splitwindow-t készítünk, csak azonos view osztályhoz tartozhatnak a nézetek.

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
 // TODO: Add your specialized code here and/or call the base
class
 return m_wndSplitter.Create(this,2,2,CSize(10,10),pContext);
 //return CFrameWnd::OnCreateClient(lpcs, pContext);
 return TRUE;
}
```

Az ősosztály OnCreateClient hívását le kell tiltanunk, mert az csak egy nézetosztályt hoz létre!



6. Gyakorlat 30. ábra Dinamikusan megosztott nézet

Teszteljük a programot! Sajnos a gördítősávok nem működnek, de a fontot külön változtathatjuk a négy nézetben!

Ha különböző típusú nézeteket akarunk, azt statikus splitwindowval valósíthatjuk meg. Ez a ***CreateStatic()*** függvényhívással lehetséges. Első paramétere a

szülőablak, második az egymás melletti ablakok száma, harmadik az egymás alatti ablakok száma.

Statikus SplitWindow esetén meg kell adnunk a view-ablakokat is külön-külön.

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
 // TODO: Add your specialized code here and/or call the base
class
 m_wndSplitter. CreateStatic(this,2,1);
 m_wndSplitter.CreateView(0,0,RUNTIME_CLASS(CPathView),
                                  CSize(100,100),pContext);
 m_wndSplitter.CreateView(1,0,RUNTIME_CLASS(CPathView),
                                  CSize(100,100),pContext);
 return TRUE;
}
```

Ahhoz, hogy a fordító felismerje az osztályok neveit, szerkesszük be a mainframe.cpp elejére a PathView header fájlt, míg a PathView.h elejére a PathDoc.h-t!

➢ Több különböző típusú osztott nézet

Tesztelésképpen hozzunk létre egy új osztályt az osztályvarázslóval!

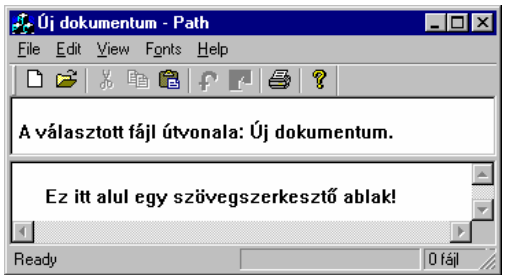
Class Wizard

Add Class

CPathEdit a neve

CEditView az őse.

A MainFrame.cpp-ben az m_wndSplitter.***CreateView*** harmadik paramétere ez legyen! Ehhez #include “PathEdit.h” kell a fájl elején.



6. Gyakorlat 31. ábra Statikusan megosztott különböző nézetek

Feladat:

Írjuk meg a hiányzó funkciókat!

➢ Legyen Form alapú a második nézet!

Készítsünk egy párbeszéd-erőforrást (Form View)

ResourceView

Insert

Dialog kinyitva

Form View (IDD_FORMVIEW)

New

Szerkesszük meg a felületet a következő ábra alapján!

Class Wizard hívásra: Nincs osztálya! Legyen New class.

Neve CPathForm

Őse a CFormView.

Azonosító hozzá az IDD_FORMVIEW

Legyen ez a második nézet! Ahhoz, hogy **RUNTIME_CLASS** makrót hívni tudjuk, ellenőrizzük a következő kódsorokat a fájlokban:

A PathForm.h-ban:

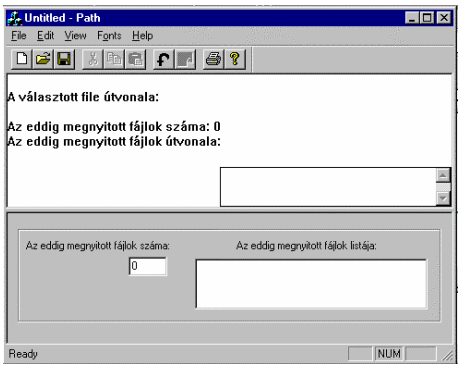
```
// Construction
public:
  CPathForm(); // protected constructor used by dynamic creation
  DECLARE_DYNCREATE(CPathForm);
```

A PathForm.cpp-ben:

```
// CPathForm
IMPLEMENT_DYNCREATE(CPathForm, CFormView)
```

A MainFrame.cpp-ben szükségünk lesz az #include PathForm.h sorra, s az ***OnCreateClient()***-be:

```
m_wndSplitter.CreateView( 1, 0, RUNTIME_CLASS(CPathForm),
CSize(100,100), pContext);
```



6. Gyakorlat 32. ábra A FormView már látszik

➢ **Rendeljünk változókat a vezérlőkhöz!**

Az IDC_FILECOUNT_EDIT-hez az int m_FileCount-ot, az IDC_LIST-hez a CListBox m_List control objektumot!

A változóknak értéket kell adnunk! Gondolkozzunk el, milyen metódusba tegyük az értékadást! Amikor a fájlnyitás esemény bekövetkezik, akkor kell értéket adnunk! Tehát a File menü Open menüpontjának kiválasztásakor! Azonban ahhoz az eseményhez már rendeltünk egy metódust. A rendszer csak az első kezelőfüggvényig megy el, mi most a másodikat akarjuk megírni. A következő szakaszt csak akkor próbálja ki, ha tesztelni akarja ezt az állítást!

ClassWizard

Message Maps fül

Class name: CPathForm

Object IDs: ID_FILE_OPEN

Messages: Command

A metódus az OnFileOpen nevet kapta:

A GetDocument() visszatérési értéke CDocument típusú. A CDocument osztálynak nincs GetFileCount metódusa, így konvertálnunk kell.

```
void CPathForm::OnFileOpen()
{
  // TODO: Add your command handler code here
  m_FileCount = ((CPathDoc*)GetDocument())->GetFileCount();
  UpdateData(FALSE);
}
```

Ahhoz, hogy fölismerje a CPathDoc osztályt: #include "PathDoc.h"

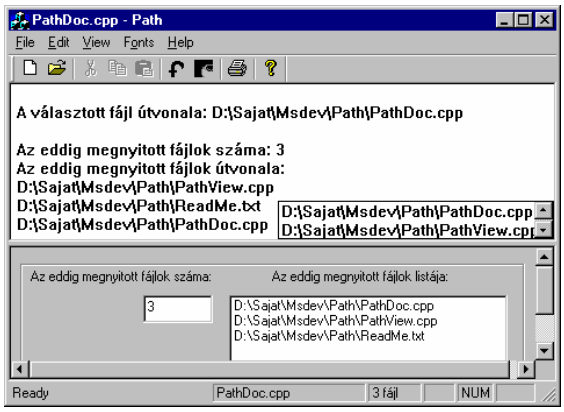
Próbáljuk ki, hogy ez a metódus valóban nem hívódik meg!

A megnyitott fájlok számát és listáját a dokumentumosztály OnOpenDocument vagy az OnNewDocument metódusa módosítja. Mindkettő hívja az ***UpdateAllViews()*** metódust, ami a nézetek ***OnUpdate()*** metódusát hívja meg. Tehát a nézetablak OnUpdate metódusában kell a vezérlőhöz rendelt változóknak értéket adnunk.

```
#include "PathDoc.h"
//...
void CPathForm::OnUpdate(CView* pSender, LPARAM lHint, CObject*
pHint)
{
 CPathDoc* pDoc=(CPathDoc*)GetDocument();
 m_FileCount=pDoc->GetFileCount();
 if (m_List && m_FileCount>m_List.GetCount())
 {
    CString st;
    POSITION pos=pDoc->GetPathList().GetTailPosition();
    st=pDoc->GetPathList().GetPrev(pos);
    m_List.AddString(st);
 }
 else if (!(m_FileCount) && (m_List))
    m_List.ResetContent();
 UpdateData(FALSE);
}
```

Mivel az OnUpdate is többször (kétszer) hívódik meg, mielőtt az elemet felfűznénk, ellenőrizzük, hogy nem került-e már föl a listára! Ezt a listadoboz elemszámának lekérdezésével tehetjük meg. Mivel az OnUpdate meghívódik a listadoboz ablakának létrehozása előtt is, mielőtt rákérdeznénk az elemszámra, meg kell tudnunk, hogy készen van-e már a listadoboz.

Most már mindkét View kijelzi az adatokat.



6. Gyakorlat 33. ábra Mindkét nézet működik

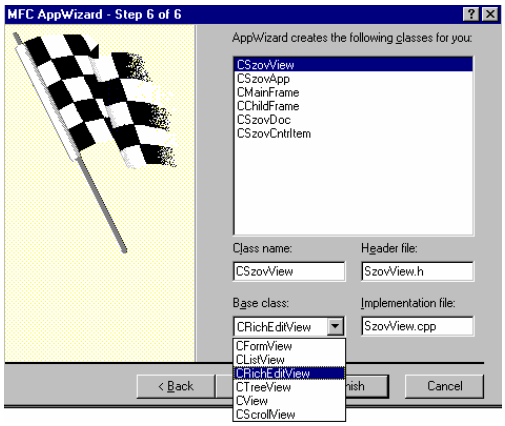
6.5. Szövegszerkesztő alkalmazás

Készítsünk szövegszerkesztő alkalmazást, melybe képet is be lehet szúrni!

6. Gyakorlat 34. ábra A szövegszerkesztő a kereső ablakkal

➢ Készítsünk egy új keretprogramot!

Neve legyen Szov. Az első lépésben Multiple document. A harmadik lépésben válasszuk a Container választógombot a None helyett! Ezzel tesszük lehetővé, hogy más alkalmazásokban készített elemeket (képet) is beszúrhassunk. A hatodik lépésben változtassuk meg a CSzovView ősosztályát **CRichEditView**-ra!



6. Gyakorlat 35. ábra A nézetosztály kiválasztása

Megjegyzés:

A CRichEditView választás esetén kötelező a Container választása. Ha ezt nem tettük meg, a végén figyelmeztet az alkalmazásvarázsló és felkínálja a beállítást.

➢ Teszteljük az alkalmazást!

↳ A File menü

Mint minden MDI alkalmazás, a File menü New menüpontjára és az eszközsor New ikonjára is egy SzovX nevű új dokumentumot nyit, ahol X a megnyitott dokumentum sorszáma.

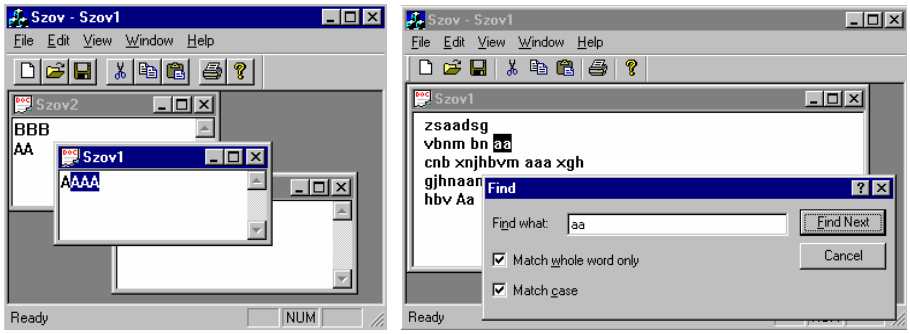
A megszokott módon működik a Save és a Save as parancs és az Open is! (Figyelem! Ha a Szov1 dokumentumot Save as segítségével Szov2 néven mentettük, majd New segítségével újabb dokumentumot nyitunk, előfordulhat az a furcsa helyzet, hogy két különböző tartalmú, de azonos nevű dokumentum lesz nyitva! A dokumentumosztályunkhoz kiterjesztést rendelve megkülönböztethetjük ezt a két dokumentumot.)

Próbáljuk ki, hogy a szöveg a nyomtatóra is nyomtatható!

↳ Edit menü

Az összes menüpont működik, a szokásos gyorsgombok is. A menüpontok és az eszközsor ikonjai mutatják a kijelölés és a vágólap állapotát.

Lehet az alkalmazásban keresni, a teljes szó és a kis- és nagybetű megkülönböztetését is figyeli. A szövegek cseréjét is elvégzi.



6. Gyakorlat 36. ábra Az eszközgombok mutatják a kijelölést és a vágólap állapotát, a keresőablak is működik

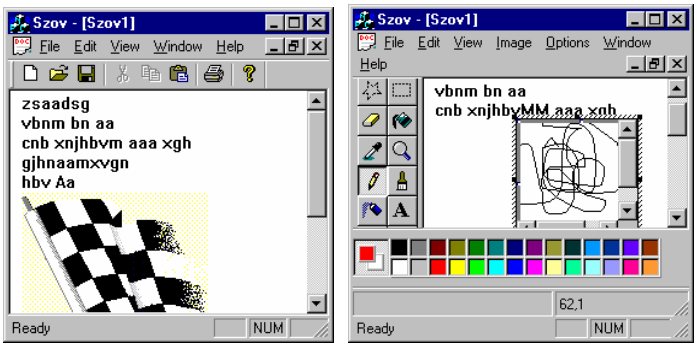
A Paste Special parancs a Container választásnak köszönhetően került a menübe, a párbeszédablakban a Bitmap-et választva a vágólapon található képet szúrja be. Az Insert New Object segítségével új, a felkínált objektum típusának megfelelő objektumot szerkeszthetünk vagy nyithatunk meg és szúrhatunk be a dokumentumba.

A View menü

Benne mind a két menüpont működik.

A Window menü

Cascade, Tile és ablakválasztó menüpontjai a vártnak megfelelőek, de a New Window menüpont, bár a címsorban kiírja, hogy az aktuális ablak egy másik nézetét látjuk, valójában a tartalmat nem mutatja, s nem is menti az adott helyre.



6. Gyakorlat 37. ábra Kép beszúrása a dokumentumba

Help menü

A szokásos About ablakot tartalmazza.

➢ Időnként új, működő menüpontokat vehetünk fel

Vegyünk fel a főmenüben egy Format nevű menüpontot, majd rendeljünk hozzá egy Font nevű almenüt! Ha a Font menüpont azonosítójának a legördülő listából az ID_FORMAT_FONT azonosítót választjuk, menüpontunk egyetlen sor kód írása nélkül is működni fog.

6.6. Feladatok

6.6.1. Arc

Ne kört, hanem arcot rajzoljunk ki! Színezzük ki! Nevessen, szomorkodjon! Beszéljen!

6.6.2. SDI pulzáló kör*

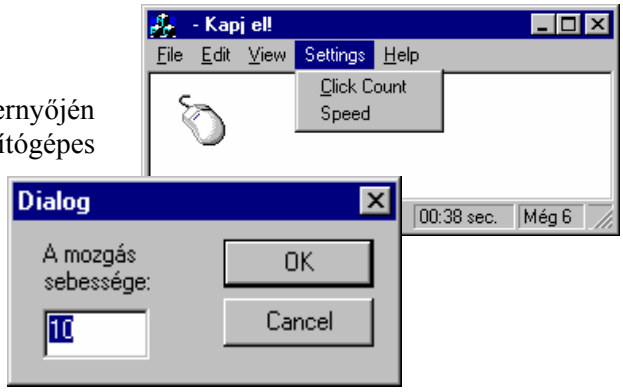
Készítsünk pulzáló kört SDI felületen!

6.6.3. Képrajzolás*

Készítsünk a rajzoláshoz bitmapeket felkínáló alkalmazást! A kép elkészítése közben választhatunk a bitmapek közül, majd az egér kattintására az aktuális helyen megjelenik a kiválasztott kép. Más bitmapet választva az újonnan választott ábrát tehetjük az ablakba az egérrel.

6.6.4. Kapj el!*

Az alkalmazás képernyőjén jelenjen meg egy számítógépes egér képe, bizonyos időközönként más-más helyen. Kéz formájú kurzorral kapjuk el az egeret! Ha sikerült rákattintanunk, álljon meg a mozgás! Újabb kattintásra a játék kezdődjön elölről!



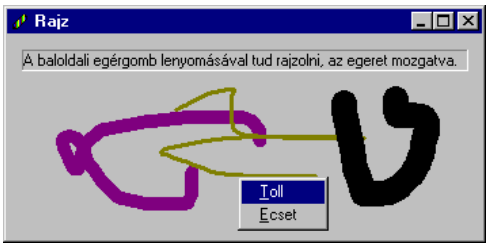
6. Gyakorlat 38. ábra Kapj el!

Az alkalmazásban legyen lehetőség az ablakméret és a számítógépes egér képének mozgatásai közt eltelt idő beállítására!

A játék az egér többszöri elkapása után álljon csak le! Legyen lehetőség e szám beállítására! A státuszsor tájékoztasson arról, hányszor kell még elkapnunk az egeret a mozgás leállításáig és mennyi ideje próbálkozunk!

6.6.5. Rajz gyorsmenüvel*

Bővítsük ki az 5.2. gyakorlat rajz alkalmazását egy a jobb oldali egérgomb lenyomása hatására felbukkanó gyorsmenüvel, amiben beállíthatjuk a rajzoláshoz használt toll és a kitöltéshez használt ecset tulajdonságait.



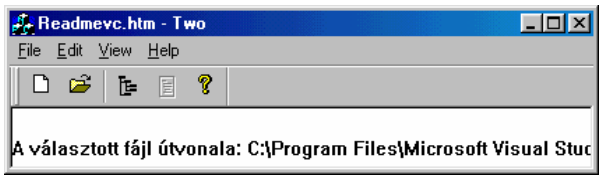
6. Gyakorlat 39. ábra Rajz gyorsmenüvel

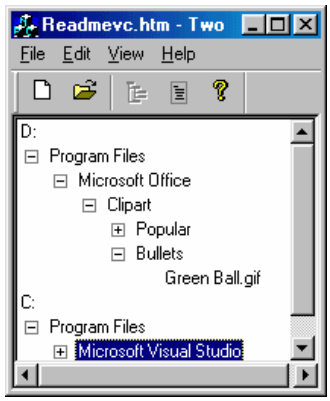
6.6.6. Eseménykezelés

Készítsünk egy esemény nevű menüpontot. Kezeljük le az üzenetét minden osztályban ahol lekezelhető úgy, hogy üzenetet küld, most éppen melyik osztályban van! Nézzük meg, milyen sorrendben kezelik le a parancsüzeneteket az osztályok!

6.6.7. Egy dokumentum, több önálló nézettel*

Készítsük el a Path alkalmazásunkat két nézettel (PathView és TreeForm) úgy, hogy egy menüpont vagy egy hozzá tartozó eszközgomb segítségével válthassunk a nézetek között!





6. Gyakorlat 40. ábra Két különböző nézet

6.6.8. Mindenütt körök több nézettel

Készítsünk a 6.1. gyakorlatban szereplő 'Mindenütt körök' alkalmazásunkhoz olyan nézetet, mely egy táblázatban kiírja a kirajzolt körök adatait!

6.7. Ötletek a feladatok megvalósításához

6.6.2. SDI pulzáló kör

Készítsünk pulzáló kört SDI felületen!

A Timert ablakhoz kell hozzárendelni, legyen ez a View! Az ***OnInitialUpdate***-ben inicializáljuk, és az ***OnDestroy()***-ban szüntessük meg!

A **WM_TIMER** esemény ***OnTimer()*** kezelőjében változtassuk az irányt, s az ***OnDraw()***-ban rajzoljuk ki a kört! Nézzük meg, mi történik, ha a kört az OnTimer rajzolja ki!

6.6.3. Képrajzolás

Készítsünk a rajzoláshoz bitmapeket felkínáló alkalmazást! A kép elkészítése közben választhatunk a bitmapek közül, majd az egér kattintására az aktuális helyen megjelenik a kiválasztott kép. Más bitmapet választva az újonnan választott ábrát tehetjük az ablakba az egérrel.

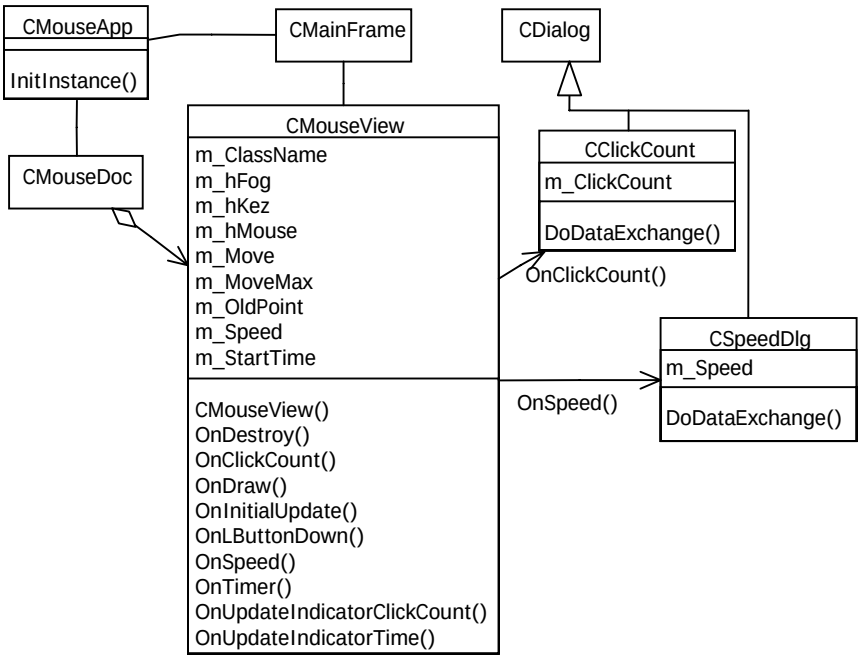
A feladatot elkészíthetjük párbeszédalapú alkalmazásként, bitmap gombokat választógombként kezelve, vagy SDI alkalmazásként, a bitmapeket eszközgombként és menüpontként (A SetRadio beállítással) kezelve.

6.6.4. Kapj el!

Az alkalmazásban legyen lehetőség az ablakméret és a számítógépes egér képének mozgatásai közt eltelt idő beállítására!

A játék az egér többszöri elkapása után álljon csak le! Legyen lehetőség e szám beállítására! A státuszsor tájékoztasson arról, hányszor kell még elkapnunk az egeret a mozgás leállításáig és mennyi ideje próbálkozunk!

- **A terv:**



6. Gyakorlat 41. ábra A 'Kapj el!' osztálydiagramja

A megoldás SDI felületen és az egér képét ikonban tárolva történik.

Az erőforrás-szerkesztőbe importáljuk a kéz alakú kurzorokat (IDC_CURSOR_KEZ, IDC_CURSOR_FOG) és az egér bitmap-et a párbeszédalapú alkalmazásból. Ha nem készítettük el, akkor szerkesszük meg őket most! Készítsünk egy új ikont IDI_ICON_MOUSE azonosítóval, és másoljuk bele az egér képét, majd színezzük a hátteret a háttérrel megegyezőre! (Zöld monitor.)

Szükségünk lesz a kurzorok és az ikon kezelőjére, valamint egy osztálynévre, mellyel a kurzort már az ablak létrehozása előtt beállíthatjuk a sajátunkra. A nézetosztályunkba tehát vegyük fel őket:

```
HCURSOR m_hKez;
HCURSOR m_hFog;
CString m_ClassName;
HICON m_Mouse;
```

A ***PrecreateWindow()*** tagfüggvényben állítsuk be a nézetosztályra az egérkurzort és az osztály ikonját!

```
BOOL CMouseView::PreCreateWindow(CREATESTRUCT& cs)
{
 // TODO: Modify the Window class or styles here by modifying
 //  the CREATESTRUCT cs
 m_hKez=AfxGetApp()->LoadCursor(IDC_CURSOR_KEZ);
 m_hFog=AfxGetApp()->LoadCursor(IDC_CURSOR_FOG);
 m_hMouse=AfxGetApp()->LoadIcon(IDI_MOUSE);

 m_ClassName = AfxRegisterWndClass
    (CS_HREDRAW | CS_VREDRAW, m_hKez,
                              // Itt állítjuk be a saját kurzort.
    (HBRUSH)::GetStockObject(WHITE_BRUSH),
                              // Szürke háttérhez LTGRAY_BRUSH.
    m_hMouse);
 cs.lpszClass = m_ClassName;

 return CView::PreCreateWindow(cs);
}
```

Írjuk fölül a View osztály OnInitialUpdate tagfüggvényét, indítsuk el az időzítőt!

```
void CMouseView::OnInitialUpdate()
{
 CView::OnInitialUpdate();

 // TODO: Add your specialized code here and/or call the base
 //class
 SetTimer(MOVE,1000,NULL);
}
```

A program végén, még az ablak megszűnése előtt állítsuk le az időzítőt. Ehhez a **WM_DESTROY** üzenetkezelőjét az ***OnDestroy()***-t kell elkészítenünk.

```
void CMouseView::OnDestroy()
{
 CView::OnDestroy();
 // TODO: Add your message handler code here
 KillTimer(MOVE);
}
```

- **Csatoljuk az időzítő eseményéhez az egér kirajzolását!**

Egyenlőre mindig ugyanoda, hogy ellenőrizni tudjuk az eddigieket!

```
void CMouse1View::OnTimer(UINT nIDEvent)
{
    int cx =100;
    int cy=100;
    CClientDC dc(this);
    dc.DrawIcon(cx,cy,m_hMouse);
 CView::OnTimer(nIDEvent);
}
```

Tegyük véletlen helyre az egér képét! Az int cx = 100; int cy =100; helyett:

```
    CRect clientRect;
    GetClientRect(clientRect);
    int cx=clientRect.right;
    int cy=clientRect.bottom;
    //Az Ikon mérete 32x32!
    //Ezt vonjuk ki, hogy ne lógjon ki a képből!
    cx-=32;    cy-=32;
    cx =rand()*cx/RAND_MAX;
    cy =rand()*cy/RAND_MAX;
```

Az előző helyről nem törlődik az ikon. Töröljük le úgy, hogy téglalapot rajzolunk fölé! Ha a vonala is fehér, a téglalap nem fog látszani a képernyőn. Ehhez persze tárolnunk kell a bal felső sarokpontját az utolsó ikonrajznak. Tároljuk egy CPoint m_OldPoint adattagban a View osztályunkban!

```
void CMouseView::OnTimer(UINT nIDEvent)
{
 //...
    CClientDC dc(this);
              //Törli az előző ikont
              dc.SelectStockObject(WHITE_PEN);
               dc.Rectangle(m_OldPoint.x,m_OldPoint.y,
                     m_OldPoint.x+32,m_OldPoint.y+32);
    dc.DrawIcon(cx,cy,m_hMouse);
              m_OldPoint.x=cx;
              m_OldPoint.y=cy;
 CView::OnTimer(nIDEvent);
}
```

Természetesen az m_OldPoint adattagnak kezdőértéket kell adnunk. Erre a legalkalmasabb hely a konstruktor.

➢ Változzon a kurzor képe!

A mikor lenyomjuk az egérgombot, a kurzor változzon az m_Fog képre!

```
void CMouseView::OnLButtonDown(UINT nFlags, CPoint point)
{
 // TODO: Add your message handler code here and/or call
default
 SetCursor(m_hFog);
 CView::OnLButtonDown(nFlags, point);
}
```

➢ Nehezítsük a feladatot!

Vegyünk fel egy int típusú változót, melyben tárolni fogjuk, hogy elkaptuk-e az egeret, illetve hányszor kell még elkapnunk ahhoz, hogy az egér mozgása leálljon! (m_Move) Kezdetben mozogjon az egér, így kezdőértéke legyen 1!

```
CMouseView::CMouseView()
{
 // TODO: add construction code here
 m_OldPoint.x = 100;
 m_OldPoint.y = 100;
 m_Move = 1;
}
```

Az OnTimer csak akkor mozgassa az egeret, ha m_Move értéke nem 0!

```
void CMouseView::OnTimer(UINT nIDEvent)
{
 if (m_Move)
 {
    CRect clientRect;
    ….
    m_OldPoint.y=cy;
 }
 CView::OnTimer(nIDEvent);
]
```

A feladat szövege szerint m_Move értéke nemcsak 0 vagy 1 lehet, hanem a felhasználó beállíthatja azt (m_MoveMax). Ha elkaptuk az egeret, csökkentsük m_Move értékét! Tehát akkor, ha épp az ikon fölött nyomtuk le a bal egérgombot. Ha viszont áll az egér, akkor újraindul a játék, tehát m_Move értéke a beállításnak megfelelő (m_MoveMax) lesz. Az m_MoveMax értéke kezdetben 1 (konstruktor), a felhasználó mondjuk a Settings menü Click Count menüpontjában egy párbeszédablakkal állíthatja be.

```
void CMouseView::OnLButtonDown(UINT nFlags, CPoint point)
{
 SetCursor(m_hFog);
 CRect rect(m_OldPoint.x,m_OldPoint.y,m_OldPoint.x+32,
            m_OldPoint.y+32);   // Az ikon helye.
 if (rect.PtInRect(point))       // Fölötte van-e az egér?
 {
    if (m_Move)
        m_Move--;
    else
        m_Move=m_MoveMax;
 }
 CView::OnLButtonDown(nFlags, point);
}
```

Nehezíthetjük a feladatot úgy, hogy minden sikertelen próbálkozás után eggyel többször kelljen elkapni az egeret ahhoz, hogy leálljon.

```
 if (rect.PtInRect(point))
 {
    if (m_Move)
        m_Move--;
    else
        m_Move=m_MoveMax;
 }
 else
    m_Move++;
 CView::OnLButtonDown(nFlags, point);
}
```

A Settings menü megvalósítása következik.

Vegyük fel az erőforrás-szerkesztőben az új menüpontokat! Settings / Click Count, Settings / Speed. A Click Count menüpont hatására egy párbeszédablak mutassa m_MoveMax értékét! A módosítás után OK-val kilépve az ablakból m_MoveMax értéke vegye fel a beállítottat! A Click Count kezelőjét a CMouseView osztályba célszerű megírni, mert ott az m_MoveMax értéke elérhető.

```
#include "ClickCount.h"
...
void CMouseView::OnClickCount()
{
 CClickCount dlg;
 dlg.m_ClickCount=m_MoveMax;
 if (dlg.DoModal()==IDOK)
 {
    m_MoveMax=dlg.m_ClickCount;
    m_Move=m_MoveMax;
 }
}
```

A Speed menüpont az egér gyorsaságát, tehát az időzítőt állítja. Ennek értékét is tárolni kell a nézetosztályban (m_Speed). Ha kezdetben a sebesség mondjuk 10 egység, 20 egységet beírva kétszeres sebességet akar a felhasználó. Tehát az új érték/10-zel kell osztanunk az időzítő második paraméterét, az 1000-et. A Speed menüparancs kezelőjét szintén a CView osztályba célszerű tenni, mert ez az osztály kezeli az időzítőt.

```
#include "SpeedDlg.h"
#define MOVE 1

void CMouseView::OnSpeed()
{
 CSpeedDlg dlg;
 dlg.m_Speed=m_Speed;
 if (dlg.DoModal()==IDOK)
    m_Speed=dlg.m_Speed;
 KillTimer(MOVE);
 SetTimer(MOVE,1000*m_Speed/10, NULL);
}
```

➢ A státuszsor tájékoztassa a játékost!

A **státuszsor** írja ki a még szükséges kattintások számát! Statiszta menüben és az **indicators** tömbben (MainFrm.cpp) vegyük fel a két azonosítót (ID_INDICATORS_TIME, ID_INDICATORS_CLICKCOUNT), valamint a nézetosztályban az m_StartTime az időmérés kezdetét tartalmazó tagváltozót! Az OnInitialUpdate-ben inicializáljuk, majd rendeljünk az azonosítókhoz **UPDATE_COMMAND_UI** kezelőket!

```
void CMouseView::OnUpdateIndicatorClickcount(CCmdUI* pCmdUI)
{
 CString str;
 str.Format("Még %d",m_Move);
 pCmdUI->SetText(str);
}

void CMouseSDIView::OnUpdateIndicatorTime(CCmdUI* pCmdUI)
{
 CTimeSpan time=CTime::GetCurrentTime()-m_StartTime;
 CString sTime = time.Format("%M:%S sec.");
 pCmdUI->SetText(sTime);
}
```

A játék újraindításakor állítsuk m_StartTime értékét!

Ha a sebességet kicsire állítjuk, a státuszsor időt kijelző ablaka nem frissül, csak ha mozgatjuk az egeret. Magyarázat: Nincs üzenet az üzenetsorban, nincs pihenőidős feldolgozás. Kell egy másik időzítőt is indítanunk, hogy elég

gyakran (másodpercenként) küldjön üzenetet az üzenetsorba a státuszsor kijelzőjének frissítéséhez. Mivel mindkét időzítő az OnTimert hívja, az általunk írt kódrészletet, mely az egér képét mozgatja, tegyük egy elágazásba, mely teszteli, melyik időzítő fut.

```
if (nIDEvent == MOVE)
{
   if (m_Move)
```

Az ablak alapértelmezésben méretezhető (thick frame).

Ha már teszteltük az alkalmazást, és helyesen működik, be kell állítanunk, hogy a "véletlen" számokat véletlen helyen kezdje generálni.

```
srand((unsigned)time(NULL)); // RANDOMIZE
```

A feladat megoldása során a Document / View szerkezet Document osztályát nem használjuk. A tárolásra váró adatokat (időzítő és a szükséges kattintások száma) a nézetosztályban tároljuk, mert csak a nézetosztály dolgozik rajtuk. Az adatok inkább a felhasználó beállításai kategóriába tartoznak, mint az adatokéba. Ha a tervet megnézzük is látszik a nézetosztály, mint ablak meghatározó szerepe. A menüsor és a státuszsor kezelése miatt választottuk ezt a szerkezetet, de ezeket a párbeszédablakban is felvehetjük! Döntsük el magunk, hogy a kényelmünk miatt érdemes-e SDI szerkezettel megoldani a feladatot!

 A beállítások tárolása a 7. gyakorlatban szerepel.

6.6.5. Rajz gyorsmenüvel

Rajzoljunk az egérrel! A jobb oldali egérgomb lenyomása hatására felbukkanó gyorsmenüben beállíthatjuk a toll tulajdonságait és a háttér színét, mintáját.

A megoldáshoz használjuk az 5.2. gyakorlat rajz alkalmazásának kódját!

Szúrjunk be az alkalmazás **CRajzDlg** osztályába egy 'Pop-up Menu' komponenst! (Project menü / Add to Project / Components and Controls / Developer Studio Components vagy Visual C++ Components / Pop-up Menu, Insert.) Létrejön az ***OnContextMenu()*** kezelőmetódus a **WM_CONTEXTMENU** üzenethez és a menüerőforrás.

A menü toll és ecset beállításainál használhatjuk az 5.3. gyakorlaton elkészített PropertyPage osztályokat és erőforrásokat.

```
void CRajzDlg::OnToll()
{
 CPropertySheet sheet("A toll tulajdonságlapja");
 CPenPage penPage;
 sheet.AddPage(&penPage);
 if (sheet.DoModal() == IDOK)
 {
    m_Pen.DeleteObject();
    m_Pen.CreatePen(penPage.m_LineStyle,penPage.m_LineWidth,
                                          penPage.m_Color);
    m_memDC.SelectObject(&m_Pen);
 }
}
```

Mivel az eredeti Rajz alkalmazás az m_Bitmap-ben tárolja a háttérszínt is, ezt, ha a hátteret állítani akarjuk, meg kell szüntetni, a hátteret mindig az aktuális ecsettel lefesteni, az m_Bitmap-ben pedig csak a rajzot tudjuk tárolni.

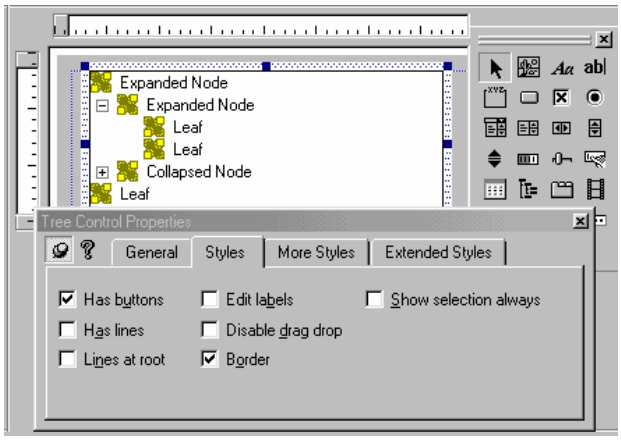
Az OnPaint IsIconic elágazás else ágában:

```
else
{
   dc.SetBkColor(m_bkColor);
   dc.FillRect(&m_rect, &m_Brush);
   dc.BitBlt(0,0, m_rect.Width(), m_rect.Height(),&m_memDC,
                                                  0,0,SRCAND);
   CDialog::OnPaint();
}
```

Az m_rect a maximalizált ablak kliensterületének méretét adja.

6.6.7. Egy dokumentum, több önálló nézettel

Az első nézet legyen a Path alkalmazás nézetablaka, míg a második egy FormView egy **CTreeCtrl** vezérlővel! Az erőforrás létrehozása után rendeljünk vezérlő típusú változót a TreeCtrl-hoz! A hozzárendelés során az osztályvarázsló segítségével létrehozhatjuk a formhoz tartozó CTreeForm nézetosztályt, melynek őse a CFormView. Az erőforrás-szerkesztőben beállíthatjuk a TreeCtrl vezérlőnk tulajdonságlapján a 'Has buttons' stí1ust, melynek hatására az olyan elemek elé, melyek további elemeket tartalmaznak a szokásos négyzet kerül, benne a + jellel jelezve, hogy tovább nyitható, illetve a – jellel jelezve, ha nyitva van.



6. Gyakorlat 42. ábra A TreeCtrl a szerkesztőben.

Azt akarjuk, hogy a vezérlő az egész nézetet foglalja el, így méretét az ablak méretének változtatása során is állítanunk kell! (WM_SIZE)

```
void CTreeForm::OnSize(UINT nType, int cx, int cy)
{
 CFormView::OnSize(nType, cx, cy);

 // TODO: Add your message handler code here
 if (m_TreeCtrl) m_TreeCtrl.MoveWindow(0,0,cx,cy);
}
```

Ha csak két nézetet kell készítenünk, egyszerűbb velük bánni, ha a CMainFrameben felveszünk két mutató típusú tagváltozót, melyek a két nézetre mutatnak! Több vagy változó számú nézet esetén a dokumentumosztály nézetlistáját kell használnunk elérésükhöz. A ***CMainFrame::OnCreateClient()*** metódusában érdemes létrehozni azokat a nézeteket melyeket az alkalmazás futása alatt végig használunk, mert itt kényelmesen elérhetjük a CCreateContext* pContext paramétert! A nézetek a keretablakosztály gyermekablakai. A függvény neve is utal rá, itt hozzuk létre őket. A létrehozás során a pContext paraméter megadásával határozzuk meg, hogy a nézetobjektum melyik dokumentumhoz tartozik. A ***CFrameWnd::CreateView()*** tagfüggvénye egyben fel is fűzi a létrehozott nézetet a dokumentumobjektum nézetlistájára.

Megjegyzés:

Ha nem mutató típusú a nézetobjektum adattag, akkor a keretablak megszüntetése során helye felszabadul. Mivel a dokumentumosztály megszüntetése során a nézetlista felszabadul, így másodszor is megpróbálkuzunk ugyanannak a területnek a felszabadításával. Ha mutató típusú a tagváltozónk, és a keretablak felszámolásakor nem szabadítjuk fel az általa mutatott területet, nem lesz ugyanaz a memóriacím kétszer felszabadítva.

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
CCreateContext* pContext)
{
 // TODO: Add your specialized code here or call the base class
 m_pView = (CTwoView*)CreateView(pContext);
 CCreateContext* pTreeContext = new CCreateContext;
 pTreeContext->m_pNewViewClass = RUNTIME_CLASS(CTreeForm);
 pTreeContext->m_pCurrentDoc = pContext->m_pCurrentDoc;
 pTreeContext->m_pCurrentFrame = pContext->m_pCurrentFrame;
 pTreeContext->m_pLastView = m_pView;
 pTreeContext->m_pNewDocTemplate = pContext->m_pNewDocTemplate;

 m_pTreeForm =
     (CTreeForm*)CreateView(pTreeContext,AFX_IDW_PANE_FIRST+1);
 return 1;//CFrameWnd::OnCreateClient(lpcs, pContext);
}
```

Mivel a nézeteket a keretablakból kényelmesen elérhetjük, a nézetváltást megvalósító menügombot ez az osztály kezelje! Figyelve melyik nézet aktuális, csak a másik legyen választható!

```
void CMainFrame::OnViewTreeview()
{
 m_pTreeForm->SetDlgCtrlID(AFX_IDW_PANE_FIRST);
 m_pView->SetDlgCtrlID(AFX_IDW_PANE_FIRST+1);
 m_pTreeForm->ShowWindow(SW_SHOW);
 m_pView->ShowWindow(SW_HIDE);
 SetActiveView(m_pTreeForm);
 RecalcLayout();
}

void CMainFrame::OnUpdateViewTreeview(CCmdUI* pCmdUI)
{
 pCmdUI->Enable( GetActiveView()==m_pView);
}
```

Mivel az OnInitialUpdate minden dokumentum megnyitása után meghívódik, ez a legalkalmasabb tagfüggvény az új elemek felfűzéséhez.

```
void CTreeForm::OnInitialUpdate()
{
 CFormView::OnInitialUpdate();

//TODO:Add your specialized code here andor call the base class

 if (!GetDocument()) return;
 CTwoDoc* pDoc = (CTwoDoc*)GetDocument();
 CString st=pDoc->GetPathName();
 if (st == "" )      return;

 HTREEITEM current=TVI_ROOT;
 HTREEITEM parent=m_TreeCtrl.GetRootItem();
 st=st+"\\";
 CString itemText = st.SpanExcluding("\\");    // \-ig halad.
 if(itemText!="")
 {
    // Keresi az azonos nevű gyökeret.
    while (parent != NULL && itemText !=
                           m_TreeCtrl.GetItemText(parent))
    {
        parent = m_TreeCtrl.GetNextItem(parent, TVGN_NEXT);
    }
    if (parent == NULL) // Nincs azonos nevű gyökér.
    {
        parent = TVI_ROOT;  // Új gyökérelem.
    }

    //A gyökér megegyezik egy már létezővel.
    else if (itemText == m_TreeCtrl.GetItemText(parent))
    {
        // Veszi a következő elemet.
        st.Delete(0,itemText.GetLength()+1);
        itemText = st.SpanExcluding("\\");
        // Keresi a 'parent' gyermekei közt az 'itemText' nevűt
        while (itemText!="" && SeekChilds(itemText, parent))
        {
        // Ha talált, az lett a parent, és veszi a következőt.
              st.Delete(0,itemText.GetLength()+1);
              itemText = st.SpanExcluding("\\");
        }
    }
```

```
    // A jelenlegi parent gyermekei közt nincs 'itemText'-el
    //megegyező. Tehát a hátralevő elemeket fel kell fűzni.
    while (itemText!="")
    {
     current= m_TreeCtrl.InsertItem(itemText,parent, TVI_LAST);
     m_TreeCtrl.Select(current,TVGN_CARET);
     parent = current;
     st.Delete(0,itemText.GetLength()+1);
     itemText = st.SpanExcluding("\\");
    }
  }
}

// A 'parent' paraméter gyermekei között keresi
// az 'itemText'-ben megadott szöveget.
BOOL CTreeForm::SeekChilds(CString itemText, HTREEITEM& parent)
{
  if (m_TreeCtrl.ItemHasChildren(parent))
  {
    HTREEITEM child=m_TreeCtrl.GetChildItem(parent);
    while (child != NULL && itemText!=
                          m_TreeCtrl.GetItemText(child))
    {
        child = m_TreeCtrl.GetNextItem(child, TVGN_NEXT);
    }
    // Megtalálta az azonos szövegűt a gyermekek között.
    if (itemText==m_TreeCtrl.GetItemText(child))
    {
        parent = child; // A következő keresésnél ő a parent.
        return TRUE;
    }
  }
  // Ha nincs gyereke, vagy nincs közte 'itemText'-el egyező.
  return FALSE;
}
```

7. Gyakorlat A registry használata

7.1. Beállítások tárolása a registryben

A 6.6.4. feladatban (Kapj el!) beállítottuk az egérmozgás sebességét, az ablakméretet és annak értékét, hányszor kell az egeret elkapni ahhoz, hogy megálljon a mozgása.

Ezeket a beállításokat szeretnénk a program két futása közt is megőrizni a registryben.

Az első függvény, amit meg kell hívnunk a **registry** használatához, az alkalmazásosztályunk ***SetRegistryKey()*** metódusa az ***Initinstance()*** tagfüggvényben. Ezt az alkalmazásvarázsló megtette nekünk, a gyártó neveként a "Local AppWizard-Generated Applications"-t adva meg. A függvény hatására, ha még nincs ilyen, akkor létrejön egy kulcs a gyártó nevével ("A Visual C++ és az MFC") és egy alkulcs az alkalmazás nevével. A későbbiekben ez alá az alkulcs alá írunk kulcsokat (7. Gyakorlat 1. ábra).

A SetRegistryKey függvényt meghívhatjuk a String Tableben megadott sztring erőforrás paraméterrel is.

➢ A sebesség és a kattintások számának megőrzése

A nézetosztály (ő tárolja a beállításra vonatkozó adatokat) destruktorában kiírjuk, konstruktorában beolvassuk az adatokat.

```
CMouseView::~CMouseView()
{
 CWinApp* pApp;
 pApp=AfxGetApp();
 pApp-> WriteProfileInt("Settings", "MoveMax", m_MoveMax);
 pApp-> WriteProfileInt("Settings", "Speed", m_Speed);
}

CMouseView::CMouseView()
{
 // TODO: add construction code here
 m_OldPoint.x=100;
 m_OldPoint.y=100;
 CWinApp* pApp;
 pApp=AfxGetApp();
 m_MoveMax= pApp->GetProfileInt("Settings", "MoveMax", 1);
 m_Move= m_MoveMax;
 m_Speed= pApp->GetProfileInt("Settings", "Speed", 10);
 srand((unsigned)time(NULL)); // RANDOMIZE
}
```

A GetProfileInt harmadik paramétere a visszatérési érték, ha nem találja a registryben a bejegyzést.

Az időzítő kezdeti beállításánál is vegyük figyelembe az m_Speed értékét!

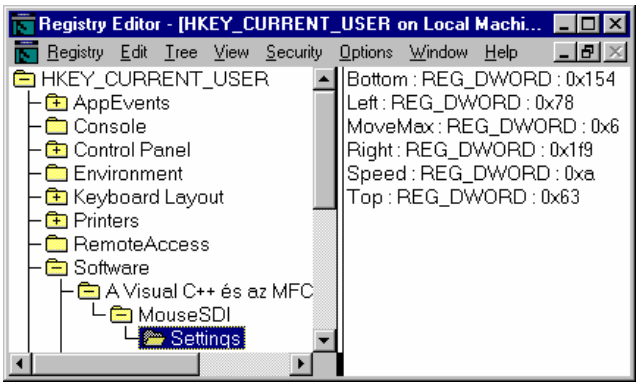
```
void CMouseView::OnInitialUpdate()
{
 CView::OnInitialUpdate();
 // TODO: Add your specialized code here and/or call the base
class
 SetTimer(MOVE,1000*10/m_Speed, NULL);
 SetTimer(TIME,1000,NULL);
 m_StartTime=CTime::GetCurrentTime();
}
```

➢ Az ablak méretének beállítása

Az ablakméretet a keretablak határozza meg. Tehát ennek a bezárásakor (***DestroyWindow()*** virtuális metódus) kell mentenünk az ablak téglalapjának adatait, s létrehozásakor (OnCreate) pedig beolvasni azokat. (Maximalizált ablak esetén negatív értékek is lehetnek!)

```
BOOL CMainFrame::DestroyWindow()
{
 CWinApp* pApp;
 pApp=AfxGetApp();
 CRect rect;
 GetWindowRect(rect);
 pApp-> WriteProfileInt("Settings", "Left", rect.left);
 pApp-> WriteProfileInt("Settings", "Top", rect.top);
 pApp-> WriteProfileInt("Settings", "Right", rect.right);
 pApp-> WriteProfileInt("Settings", "Bottom", rect.bottom);

 return CFrameWnd::DestroyWindow();
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
//...
 CWinApp* pApp;
 pApp=AfxGetApp();
 CRect rect;
 rect.left= pApp->GetProfileInt("Settings", "Left", 100);
 rect.top= pApp->GetProfileInt("Settings", "Top", 100);
 rect.right= pApp->GetProfileInt("Settings", "Right", 500);
 rect.bottom= pApp->GetProfileInt("Settings", "Bottom", 500);
 MoveWindow(rect);

 return 0;
}
```

Futtassuk a **regedt32** regisztrációs adatbázis-szerkesztőt! A **HKEY_CURRENT_USER** ablakban a Software mappa A Visual C++ és az MFC kulcs alatt ott találjuk az alkalmazásunkat, és a Settings kulcs alatt az általunk megadott változók adatait nézhetjük végig (7. Gyakorlat 1. ábra).



7. Gyakorlat 1. ábra A registry adatai a Regedt32 szerkesztőben

7.2. Feladatok

7.2.1. Az utoljára használt dokumentum beolvasása indításkor

Az 5. gyakorlat 5.1.3. feladatában állítsuk be az utoljára használt dokumentum automatikus megnyitását, ha az alkalmazás nyitásakor nem adtunk meg a parancssorba a dokumentum nevét!

7.2.2. Készítsünk ini fájlt az alkalmazásunkhoz!

A ***SetRegistryKey()*** hívás törlésével nem kapcsolódik alkalmazásunk a registryhez. Létrehoz a Windows alkönyvtárunkban egy My.ini fájlt. A regisztrációs adatbázis helyett ebbe a fájlba ír, ebből a fájlból olvas. Készítsük el a My.ini fájlt! Nézzük meg tartalmát! Ellenőrizzük, hogy valóban az ini fájl segítségével is megőrzi a beállításokat! Például nyitja-e az utolsó dokumentumot automatikusan?

7.2.3. Tanulmányozzuk a registry bejegyzéseket

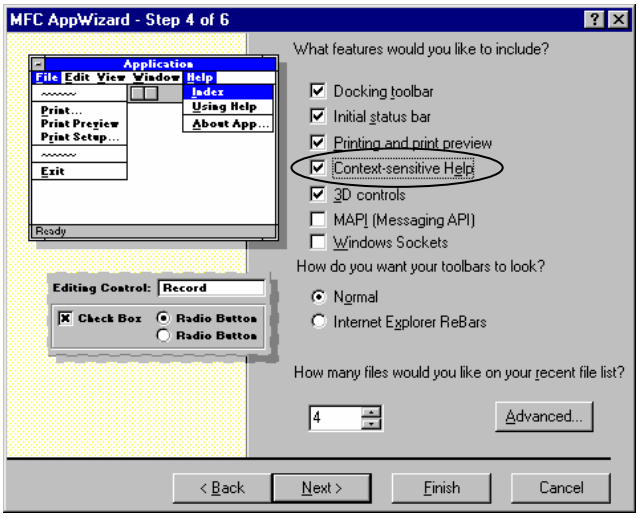
A 'Mindenütt alakzatok' alkalmazásunkban (6.2. gyakorlat) különböző fájlkiterjesztésű dokumentumokkal dolgoztunk. Keressük meg a regisztrációs adatbázisban a **HKEY_CLASSES_ROOT** kulcs alatt a Teg.Document, és a Kor.Document bejegyzéseket! (**Find Key**)

8. Gyakorlat Helyzetérzékeny súgó készítése

A legkényelmesebben úgy készíthetünk súgót, hogy az alkalmazásvarázslóban kiválasztjuk a context-sensitive help jelölőnégyzetet.

8.1. Súgó készítése a projektből

Készítsünk egy SDI alkalmazást FormView-val és súgóval! A projekt neve legyen MyHelp!



8. Gyakorlat 1. ábra A súgó-támogatás beállítása az alkalmazásvarázslóban

Ne feledjük az utolsó lépésben a View ősosztályának a CFormView-t választani!

8.1.1. Futó alkalmazásunk súgóval

Nézzük meg, mi történt a jelölőnégyzet választás hatására a kész alkalmazásban!

A futó alkalmazásunk bővült egy menüponttal, mely a Help főmenü Help Topics almenüje. Az eszköztárunk pedig a helyzetérzékeny súgó eszközgombbal. Próbáljuk ki őket!

A Help Topics hatására kinyílik a más alkalmazásokból ismerős súgó ablak a három füllel.

- A Contents tartalmazza a már készen generált menüpontok angol nyelvű leírását.
- Az Index-ben is találunk néhány kulcsszót.
- A Find fül választás hatására megkeresi a rendelkezésre álló adatokat, és bizonyos szavakra már rá is tudunk keresni. Gyakran a súgó szövege – mint pl. Replace Dialog választás esetén – csak a következő: << Write application-specific help here. >>

A helyzetérzékeny súgó eszközgomb választás hatására, ha a kérdőjeles kurzorunkkal pl. a nyomtató ikonra kattintunk, megjelenik a nyomtatáshoz tartozó súgó szövege.

A státuszsorban a következő sort olvashatjuk, ha a kurzorunk a kliensterületre mutat: For Help, press F1! És valóban a kurzort például a Save eszközgomb fölött

tartva F1 hatására megjelenik a mentéshez tartozó súgó szöveg. Shift F1 hatására pedig a kurzorunk a helyzetérzékeny súgó ikonját veszi föl, s a keresett vezérlőt kiválasztva megkapjuk a súgó szövegét, feltéve, hogy megírtuk azt.

8.1.2. Projekt súgóval

Most nézzük meg a project fájljai hogyan változtak!

Természetes módon a menü erőforrásban megjelent a Help Topics, az eszköztárban a helyzetérzékeny súgó eszközgomb és a gyorsgombok között a VK_F1, és a Shift+VK_F1.

A String Table AFX_IDS_IDLEMESSAGE szövege Ready-ről For Help, press F1-re módosult. Új erőforrásunk az AFX_IDS_HELPMODEMESSAGE, mely akkor jelenik meg, ha helyzetérzékeny súgó üzemmódban van alkalmazásunk. "Select an object on which to get Help"

A CMainFrame osztály MESSAGE_MAP-jében megtalálhatjuk a súgó kezelését támogató függvényeket.

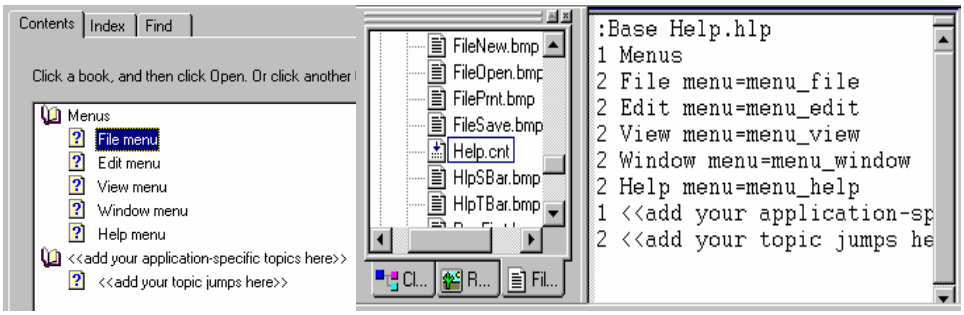
```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
     // NOTE - the ClassWizard will add and remove mapping
macros here.
     //    DO NOT EDIT what you see in these blocks of generated
code !
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
  // Global help commands
  ON_COMMAND(ID_HELP_FINDER, CFrameWnd::OnHelpFinder)
  ON_COMMAND(ID_HELP, CFrameWnd::OnHelp)
  ON_COMMAND(ID_CONTEXT_HELP, CFrameWnd::OnContextHelp)
  ON_COMMAND(ID_DEFAULT_HELP, CFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()
```

Jelentésük:

- ID_HELP_FINDER: Téma megjelenítése,
- ID_HELP: F1 gomb,
- ID_CONTEXT_HELP: Shift+F1 gomb,
- ID_DEFAULT_HELP: Ha nincs téma rendelve az azonosítóhoz.

Az elmélet 8.1.2. szakasza részletesen ismerteti a fájlok feladatát.

Ha megnyitjuk a .cnt fájlt megtaláljuk benne a Contents fül (tartalomjegyzék) leírását, melyet futáskor már megnéztünk az alkalmazásban (8. Gyakorlat 2. ábra).



8. Gyakorlat 2. ábra A Contents fül futás közben, és leírása a .cnt fájlban

A projekt mappájában megtaláljuk az alkalmazásvarázsló által generált MakeHelp.bat fájlt is, melyet az Open segítségével megnyithatunk.

A projekt első felépítésekor létrejön a Workspace / FileView / External Dependencies közt található MyHelp.hm fájl, mely a MyHelp projekt help map fájlja, tehát az erőforrásokat csatolja a súgóhoz. A fájlt a MakeHelp.bat által hívott makehm.exe készít el, mely az eszközök közt található.

Másoljunk egy mindenféle bejegyzést tartalmazó oldalt (pl. a Print Setup command (File menu)) az AfxPrint.rtf fájl végére. Írjuk át a szövegét a mi Saját témánk súgójának szövegére! Adjuk meg a saját téma és a my topic kulcsszavakat, és címként az Első saját téma szöveget!

sajat1

K saját téma; my topic

$ Első saját téma

Bezáráskor rákérdez, hogy menteni kívánjuk-e a fájlt, igen.

A fordításkor egy pillanatra megjelenik a Help Workshop.

Futáskor már működik is az Index fül, benne a két kulcsszóval, s megtalálja az általunk leírt témát (topicot).

Megjegyzés:

Ha a mentéskor a magyar ékezetes betűk eltűnnek, ne csodálkozzunk! A magyarázat, hogy a súgónál beállított betűtípusokban ezek a karakterek nem léteznek (ő).A megoldás: állítsuk át a súgó szöveges részén használt 'Helv' betűtípust mondjuk Ariel-re, a lábjegyzet Tms Rmn betűtípusát pedig mondjuk Times New Romanra! A legegyszerűbb, ha a Lábjegyzetszöveg stílus és a Normál stílus beállításait módosítjuk.

Készítsünk még egy témát második saját téma szöveggel!

Javítsuk a .cnt fájlunk szövegét, hogy az új oldalak is megtalálhatók legyenek benne!

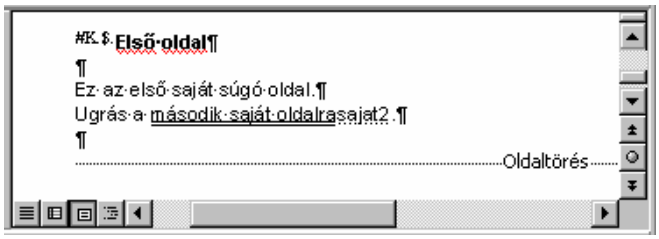
```
:Base Help.hlp
1 Menus
2 File menu=menu_file
2 Edit menu=menu_edit
2 View menu=menu_view
2 Window menu=menu_window
2 Help menu=menu_help
1 Saját témák
2 Első saját téma=sajat1
2 Második saját téma=sajat2
2 <<add your topic jumps here>>=main_index
```

A súgónk tartalomjegyzékében megjelent a saját témák főcím, s alatta a két új alcím.

A Help Workshop eszköz segítségével kényelmesebben szerkeszthetjük meg súgófájljainkat!

➢ Hiperhivatkozások a szövegben

Ha egyik témakörünkből a másikra akarunk hivatkozni, a szöveget, melyre az egérrel kattintva az új téma ablaka kinyílik kétszeres aláhúzással jelöljük. (A szöveg kijelölése után: Formátum, Betűtípus, az Aláhúzás kombipanel listájából a Dupla választása.) Közvetlen ezután írjuk a kívánt téma azonosítóját rejtett stílusú szöveggel. (Kijelölés, Formátum, Betűtípus, Rejtett jelölőnégyzet.) Ha a rejtett szöveget nem látjuk, az Eszközök menü, Beállítások, Megjelenítés fül, Rejtett szövegrész jelölőnégyzet választással láthatóvá tehetjük.



8. Gyakorlat 3. ábra Az első saját súgóoldal forrása hiperhivatkozással

8.2. Helyzetérzékeny súgó készítése

Vegyünk föl egy új menüpontot ID_MY azonosítóval! Fordításkor a MyHelp.hm-be egy új azonosító kerül HID_MY néven. Készítsünk súgó oldalt, melynek HID_MY az azonosítója! A Help  ikon kiválasztása után a My menüpontra kattintva a súgó HID_MY azonosítójú oldala töltődik be.

Ha a My menüpont az aktuális, F1 hatására is a hozzá csatolt oldal súgója jelenik meg. Sőt, ha eszközgombot csatolunk a menüponthoz, a helyzetérzékeny súgó az eszközgombra is működik.

Lehetőség van arra is, hogy egy azonosítóhoz már megírt témakört csatoljunk. Erre szolgál a .hpj fájl ALIAS kategóriája. Nincs más dolgunk, mint az azonosítót egyenlővé tenni a kívánt oldalhoz tartozó context ID-vel.

A következő sor hatására a My menühöz a sajat2 oldalt csatoljuk:

[ALIAS]

HID_MY=sajat2

➢ Helyzetérzékeny súgó a fom felületen

A párbeszédfelületen található vezérlőkhöz is szeretnénk helyzetérzékeny súgót írni! Ehhez először a vezérlő tulajdonságlapján kell bejelölnünk a Help ID jelölőnégyzetet! A következő fordításkor létrejön a resource.hm, mely értelemszerűen az általunk létrehozott erőforrások helyzetérzékeny-súgójához tartalmazza a súgóazonosítókat. Azokhoz a vezérlőkhöz, melyeknél bejelöltük a HelpID-t kapunk súgóazonosítókat. pl. IDC_BUTTON1->HIDC_BUTTON1

Az előbbiek alapján készítsünk súgó oldalt az azonosítóhoz!

A help projekt fájlban (.hpj) tudnunk kell az azonosítókról, ha dolgozni akarunk velük. Ezért a [MAP] szakaszba be kell szerkesztenünk a resource.hm fejlécfájlt. Mivel ez a fájl nem a debug alkönyvtárban, hanem a forrásfájlok között található, így a kód a következő lesz:

```
[MAP]
#include <..\resource.hm>
#include <C:\Program Files\Visual Studio\VC98\MFC\include\
                                                afxhelp.hm>
#include <Help.hm>
```

Ha a Help eszközgombot kiválasztjuk, a párbeszédfelületre kattintva a súgó kezdőoldala hívódik meg. Az F1 gombra nem reagálnak a vezérlők.

Ahhoz, hogy F1 hatására is elinduljon a súgó, a CWinApp::WinHelp metódusának át kell adnunk a vezérlőhöz tartozó súgótéma azonosítóját. Ehhez természetesen a fájl elején be kell szerkesztenünk a resource.hm help map fájlt.

A kívánt súgóoldalt a nézetosztály OnHelpInfo (virtuális) metódusának felülírása segítségével érhetjük el.

```
BOOL CHelpView::OnHelpInfo(HELPINFO* pHelpInfo)
{
 // Csak akkor hívja meg, ha F1-et ütünk az aktuális vezérlőn.
 CRect rect;
 GetWindowRect(&rect);
 if (rect.PtInRect(pHelpInfo->MousePos)) //View terület
 {
    AfxGetApp()->WinHelp(pHelpInfo->dwContextId);
    return TRUE;
 }
 return CFormView::OnHelpInfo(pHelpInfo); //Keret terület
}
```

Ekkor F1 hatására a fókuszban levő vezérlő azonosítójához tartozó súgóoldal nyílik ki. Ha a vezérlőnél nem jelöltük a Help ID-t, vagy nincs ilyen azonosítójú súgó oldal, akkor hibaüzenetet kapunk. Tehát vagy a WinHelp hívás előtt kell tesztelnünk a vezérlő azonosítót, vagy minden azonosítóhoz rendelni kell súgóoldalt, például a főoldalt.

Egér segítségével még mindig a főoldal nyílik ki a nézet fölött, a menüpontokon duplán kell kattintani. A View fölött nem működik a Shift F1, csak a keret fölött.

8.3. Feladatok

8.3.1. Válasszuk ki egy már megírt alkalmazásunkat, és készítsünk súgót hozzá!

Az elkészített súgó legyen helyzetérzékeny!

Irodalomjegyzék

1. Ackermann (1996) Philip: Developing Objec-Oriented Multimedia Software, dpunkt, Heidelberg, 1996.
2. Alexander (1997) Christopher: A Pattern Language: Towns/Buildings/Construction, Oxford University Press, Inc. 1977.
3. America (1987) P.:POOL-T: A paralell object-oriented language. Object-Oriented Concurrent Programming, MIT Press, Camebridge, Mass, 1987.
4. Ágoston (2000) György: Windows a gyakorlatban, LSI Oktatóközpont, Budapest, 2000.
5. Baga (1998) Edit: Delphi másképp, Magán kiadás, Budapest, 1998.
6. Benkő (1995) Tiborné, Kiss Zoltán, Kuzmina Jekatyerina, Dr. Tamás Péter, Tóth Bertalan: Könnyű a Windows-t programozni!?, ComputerBooks, Budapest, 1995.
7. Benkő (1996) Tiborné, Benkő László, Poppe András: Bevezetés a Borland C++ programozásba, ComputerBooks, Budapest, 1996.
8. Benkő (1998) Tiborné, Benkő László, Poppe András: Objektum-orientált programozás C++ nyelven, ComputerBooks, Budapest, 1998.
9. Bennett (1997) David: Visual C++ 5 Developer's Guide, Sams Publishing, Indianapolis, 1997.

10. Bernstein (1997) Philip A., Newcomer Eric: Principles of Transaction Processing, Morgan Kaufmann Publishers Inc. San Francisco, 1997.
11. Biddle (1998) Robert, Mercer R, Wallingford E: Evaluating Object-Oriented Design Workshop, Proceedings, Educator's Symposium, OOPSLA'98, Vancouver, ACM, New York, 1998.
12. Bodor (1998) László: C/C++ programozás, LSI Oktatóközpont, Budapest, 1998.
13. Booch (1998) Grady, Rumbaugh J, Jacobson I: The Unified Modeling Language User Guide, Addison Wesley Longman Inc., Reading Massachusetts, 1998.
14. Brown (1998) Alan W., Wallnau Kurt C: The Current State of CBSE, IEE Software, 1998 Szeptember / Októberi száma, IEEE Computer Society http://computer.org.
15. Demeter (1999) Ibolya: Visual Basic 6.0, Panem Könyvkiadó, Budapest, 1999.
16. Eddon (1998) Guy, Eddon Henry: Inside Distributed COM, Microsoft Press, Redmond Washington, 1998.
17. Gamma (1995) Erich, Helm R, Jonson R, Vlissides J: Design Patterns, Addison Wesley Longman Inc., Reading Massachusetts, 1995.
18. Gurewich (1996) Ori, Gurewich Natham: Teach yourself Visual C++ 4, Sums Publishing, Indianapolis, 1996.
19. Hargittai (1993) Péter, Kaszanyitczki László: Visual C++ fejlesztői rendszer Windows alatt. LSI kiadó Budapest, 1993.
20. Illés (1993) Zoltán: Programozási nyelvek: C++ Az ELTE μrológia sorozatának 23. kötete. ELTE TTK 1993.
21. Jacobson (1997) I, Griss M, Jonsson P: Software Reuse ACM Press, New York, 1997.
22. Jacobson (1999) Ivar, Rumbaugh J, Booch G,: The Unified Software Development Process, Addison Wesley Longman Inc., Reading Massachusetts, 1998.
23. Kernighan (1988) B. W. Ritchie D. M.: A C programozási nyelv, Műszaki Könyvkiadó, Budapest, 1988.
24. Kirtland (1999) Mary: Designing Component-Based Applications, Microsoft Press, Redmond, 1999.
25. Kis (1999) Balázs: Windows NT 4.0, SZAK Kiadó Kft., Bicske, 1999.
26. Kondorosi (1997) Károly, László Zoltán, Szirmai-Kalos László: Objektum-orientált szoftverfejlesztés, ComputerBooks, Budapest, 1997.

27. Ledgard (1996) Henry F.: Az objektumorientált programozás alapjai, Műszaki Könyvkiadó, Budapest, 1996.

28. Lippmann (1992) Stanley B.: C++ először, Novotrade Kiadó kft, Budapest, 1992.

29. Madsen (1998) Ole Lehrmann Madsen, Henric Ron, Kresten Krab Thorup & Mads Torgersen, A Conceptual Approch to Teaching Object-Orientation to C Programmers, Proceedings, Educator's Symposium, OOPSLA'98, Vancouver, ACM, New York, 1998.

30. Microsoft (1995) Official Curriculum: Mastering Visual C++ 4, Microsoft Press, 1995.

31. Microsoft (1997): InteractiveTraining Mastering MFC Fundamentals, Microsoft Press, 1997.

32. Microsoft (1997) VB: Official Curriculum: Mastering Microsoft Visual Basic 5, Microsoft Press, 1997.

33. Microsoft (1998): Official Curriculum Microsoft SCL Server 7.0, Microsoft Corporation, 1998.

34. Microsoft (1998) DA: Mastering Distributed Application Design Using Microsoft Visual Studio, Microsoft Corporation, 1998.

35. Microsoft (2000) Mastering MFC Fundamentals Using Microsoft Visual C++, Microsoft Press, Redmond, 2000.

36. Microsoft (1998) WA: Mastering Web Application Development Using Visual InterDev 6. Microsoft, 1998.

37. MSDN (1998): Developer Days 98, Áttérés xBase-ről, Konferencia kiadvány, Microsoft Corporation, Budapest/Redmond, 1998.

38. MSDN (1999): Windows DNA fejlesztői konferenciák, Developer Day, Budapest, 1999.

39. MSDN Library: Microsoft Developer Network Library 1-3 CD, Microsoft Corporation Corpotation 1992-2000.

40. MSDN Online: http://msdn.microsoft.com

41. OOP: Sipos Mariann: Objektumorientált programozás a C++ nyelv lehetőségeivel, A Gábor Dénes Főiskola jegyzete, 2000.

42. Pethő (1991) Ádám: abC, Számalk Könyvkiadó, Budapest, 1991.

43. QA (1997) Training: Windows Programming with Visual C++ and the MFC Library, Gloucestershire, England, 1997.

44. Quatrani (1998) Terry: Visual Modeling with Rational Rose and UML, Addison Wesley Longman Inc., Reading Massachusetts, 1998.
45. Rogerson (1997) Dale: Inside COM, Microsoft Press, Redmond Washington, 1997.
46. Rumbaugh (1998) James, Booch G, Jacobson I: The Unified Modeling Language Reference Manual, Addison Wesley Longman Inc., Reading Massachusetts, 1998.
47. Sipos (2000) Mariann: Objektumorientált programozás a C++ nyelv lehetőségeivel, Gábor Dénes Főiskola, főiskolai jegyzet, Budapest, 2000.
48. Sodhi (1992) Jag: Software Requirements Analysis and Specifications, McGraw-Hill Inc. 1992.
49. Stroustrup (2000) Bjarne: A C++ programozási nyelv, I. II. kötet, Addison-Wesley, Kiskapu, Budapest, 2001.
50. Yung (1998)/I Michael J.: Visual C++ 6 Mesteri Szinten I. kötet, Kiskapu Kiadó, Budapest, 1998.
51. Yung (1998)/II Michael J.: Visual C++ 6 Mesteri Szinten II. kötet, Kiskapu Kiadó, Budapest, 1998.
52. Vég (1999) Csaba: Alkalmazásfejlesztés a Unified Modeling Language szabványos jelöléseivel, Logos 2000 Kiadó, 1999.
53. Vég (1998) Csaba, dr. Juhász István: Objektumorientált világ, IQSOFT Budapest, 1998.
54. Zolnay (1998) András: CPP segédlet, www.aut.bme.hu/studinfo/targyinfo/jegyzet/szofttech/cpp_c.htm, 1998.

Tárgymutató

Á

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

Ü

V

W

Könyvajánló

Sipos Marianna: Programozás élesben, C#

A Visual C++ és az MFC szerzőjének új könyve a programozás alapjait mutatja be. A szerző szerint azonban ezek az alapfogalmak mára jelentős mértékben átalakultak. Mások az elvárások a ma programozójával szemben, mint korábban. Fejlődtek a fejlesztői környezetek és a programozási nyelvek is. A C# nyelv is e fejlődés eredménye. Segítségével .NET környezetben lehet az új objektumorientált fogalmak és a .NET osztálykönyvtár felhasználásával kényelmesen Windows alkalmazásokat fejleszteni. A .NET programozás előkészíti a Windows utódának a Vistának a programozását is. A könyv szerkezete a Visual C++ kötetéhez hasonlóan elméleti és gyakorlati részből áll, lehetővé téve az objektumorientált programozáson kívül a feladatok mögötti filozófia és a Visual Studio fejlesztőeszköz által kínált technikák megismerését is.

A könyv feladatmegoldásai olyan a részletesek, hogy a forráskódra nincs szükség, de letölthető a http://www.nik.hu/aii/oktatok/siposmarianna.html honlapról.

Sipos Marianna – a Visual C++ és az MFC könyv szerzője – a BMF Neumann Informatikai Karán programozást és szoftvertechnológiát tanít. Jelentős szakmai és oktatásmódszertani tapasztalatokkal rendelkezik. Doktori (PhD) disszertációját programozásoktatás témakörben írta. A könyv feladatait a BMF NIK műszaki informatika szakos és az ELTE IK programozó matematikus hallgatóival próbálta ki.

Adatok:
ISBN: 963 216 652 3
Oldalszám: 354 oldal
Kategória: Programozás, C#
Felhasználói szint: Kezdő-haladó
Kapható: a könyvesboltokban.

...önálló tanuláshoz...

Sipos Marianna

Programozás élesben

KÉNYELMES, GYORS ÉS HATÉKONY
.NET ALKALMAZÁSFEJLESZTÉS,
MODERN FEJLESZTŐESZKÖZ,
KOMPONENSALAPÚ PROGRAMOZÁS,
EGYSZERŰEN FELDOLGOZHATÓ,
NAPRAKÉSZ ISMERETEK

Deutsch:  In English:  Magyarul: 

Iványi Antal Miklós

Oktatás

Kutatás

Kedvenc könyvek

Személyes adatok

TELEK (Tony Elektronikus Könyvtára)

Budapest, Szeptember 30.

A honlap kezel•je: tony@inf.elte.hu (a honlapot átalakítom, a hibákért elnézést kérek)

Vissza **Iványi Antal Miklós** magyar nyelv• honlapra